DIBBOlib

A Data-Intensive Black-Box Optimization Library for Apache Spark

Project Report

cs-23-dt-10-03

Aalborg University
Electronics and IT

Department of Computer Science
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
DIBBOlib: A Data-Intensive Black-Box Optimization Library for Apache Spark

Theme:
Prescriptive Analytics on Big Data

Project Period:
Spring Semester 2023

Project Group:
cs-23-dt-10-03

Participant(s):
Martin Moesmann

Supervisor(s):
Torben Bach Pedersen

Copies: 1

Page Numbers: 214

Date of Completion:
June 16, 2023

Abstract:

Motivated by the advance of mathematical optimization within contemporary analytics, this project develops a *sample-efficient black-box optimization library*, extending the *Apache Spark* platform for data-intensive analytics. Named *DIBBOlib (Data-Intensive Black-Box Optimization library)*, this new tool enables a *data-driven*, *simulation-based* approach to problem solving, which unlike other black-box methodologies copes with non-trivial data-intensive workloads. DIBBOlib technically forms an extension of *Spark MLlib*, and is designed to feel as such from a usability standpoint. It offers an extensible standard suite of optimization algorithms and generic constraint handling methods, fully integrated with Spark SQL. Mainline features include an *algorithmic wizard*, global support for *vertical transfer learning*, a novel *constraint handling method*, load-balanced *trial parallelism*, as well as *dynamic search space partitioning* based on a hybrid dynamic/greedy programming approach and e.g. cooperative game theory. Compared to alternatives, the library inhabits a special niche as a *general-purpose solution for data-intensive analytics*, while having unique features in its own right. Experiments demonstrate the usefulness of novel library features on a set of example problems.

Summary

Mathematical optimization, combined with predictive technologies such as Machine Learning, is increasingly popular within contemporary data analytics research, falling under the moniker of *Prescriptive Analytics*. Researched application domains so far span from education and healthcare to logistics and manufacturing. Researchers have however observed how a general lack of dedicated tool support and a predominance of expert-driven methodologies hinder the broader applicability of Prescriptive Analytics outside of a research setting.

This project takes its onset in an idea of how to provide a more data-driven approach to Prescriptive Analytics within data-intensive applications, which have practically no dedicated tools for it. Inspired by the successful democratization of Machine Learning, i.e. *black-box prediction*, the proposed methodology is based on *Black-Box Optimization*, which arguably brings with it some similar user-centered benefits. These methods are all *simulation-based*, in that their search strategies are formed around simulating the outcome of particular solution choices, trial-and-error style.

To accommodate a data-driven approach within a Big Data setting efficiently, we focus on a particular subset of these methods, which can be deemed *sample-efficient* when compared to e.g. *population-based* methods such as Genetic Algorithms. So far, sample-efficient methods have mostly found use within scientific simulation and various engineering design settings, in which running high-fidelity simulations during optimization might take hours or even days in extreme cases. The underlying optimization algorithms are therefore designed to form quite deliberate search strategies, with some of them even interpolating the entire search space to decide on which solution to trial next.

With the aspiration of creating a scalable, usable, and sample-efficient black-box optimization library for the Apache Spark platform, the solution developed in this project is a named *DIBBOlib (Data-Intensive Black-Box Optimization library)*.

Written in Scala, DIBBOlib extends Spark MLlib, and thus integrates with existing advanced analytics functionality on the platform. At its core, the library simply extends the MLlib Transformer class with a subclass named BlackBoxOptimizer. This subclass implements a generalized way of translating a black-box

optimization model to and from Spark SQL queries, solving the specified problem with a particular optimization algorithm. For better usability and easier onboarding, DIBBOlib Transformers support the same API as MLlib Transformers.

DIBBOlib offers an extensible suite of optimization algorithms as well as generic constraint handling methods, covering all broadly useful approaches within sample-efficient BBO. It furthermore offers an *algorithmic wizard* for automatic algorithm selection and chaining for given input problems.

On top of this, it supports universal *vertical transfer learning*, meaning that it offers a transparent way to checkpoint progress made on a particular problem, and continue progressing across several optimization runs, using different optimization algorithms or subproblem configurations in each.

To support *generally constrained optimization* throughout the library, efforts were made on finding a generic way to integrate constraint definition and handling with Spark SQL, and offering a consistent baseline level of support for all algorithms, while accommodating constraint handling methods more or less limited to individual algorithms or constraint types.

As a byproduct of the vertical transfer learning supported by the library, a new approach to general constraint handling is proposed: *The Historical Revisionist Method*. Unlike similar methods, it uses a data-driven approach to recalibrate its strategy. It essentially defines a new objective function and SQL view of evaluation history by fitting model parameters to evaluation datasets, enforcing a strict preference for feasible solutions through equation-solving.

To improve the scalability of the library, DIBBOlib additionally supports running several trials in parallel. This *trial parallelism* is either enabled by a *static* user setting or by a proposed *dynamic* load balancing algorithm, a black-box optimizer continuously maximizing throughput on runtime.

DIBBOlib furthermore offers *heuristic search space partitioning*, with or without parallel optimization. Splits can be specified either statically (”manually”) or dynamically. The dynamic option obtains optimal splits based on a hybrid dynamic/-greedy programming approach, minimizing perimetric objectives based on search space geometry or game theoretic estimates of decision variable importance.

An analysis of related work concludes that DIBBOlib inhabits a unique niche among adjacent solutions, in that it is a *general-purpose solution fit for data-intensive analytics*. As found in the survey, no other solution offers this combination of features.

Experiments focused on assessing the usefulness of novel library features compared to relevant baselines. Here, both the proposed dynamic load balancing algorithm, constraint handling method, and search space partitioning approaches demonstrated significant utility for solving the example problems, compared to baseline approaches.

Contents

Chapter 1

Introduction

"The best way to predict the future is to create it" [53]. Yet decision making in a complex world is rarely as straightforward as old sayings would suggest. Actions have uncertain consequences and risks, which motivates the need for *predictive* technologies. In our day and age, predictions might be extrapolated from historical data by statistical learning, or elaborate explanatory models conceived by domain experts. Still, even perfect knowledge about the future might be insufficient for making the best choices as a human being, with the sheer number of options and their complex interrelation forming difficult obstacles. This motivates the need for *prescriptive* technologies, to support human decision making.

Within the domain of Business Analytics (BA), Prescriptive Analytics (PA) is a relatively new area, originating in the 2010s [106]. At its technical core, it involves solving *mathematical optimization problems* with respect to *predictive models*, usually obtained from some combination of *Machine Learning* (ML), *simulation*, and *domain knowledge* [58, 100].

Framing decision problems as optimization problems allows for efficient exploration of large and/or complex *search spaces* of decision options, possibly under a set of *constraints* [106]. PA, incorporating prediction into optimization, essentially makes it possible to obtain the decision option that "creates" the best possible future, perhaps making old sayings ring a bit truer to modern ears.

Back to Earth, consider as an example the decision of how much water to bring on a hike tomorrow. You want to carry the smallest load possible (objective) while fulfilling your basic hydration needs (constraint). Your hydration needs however largely depend on how sunny, humid, and windy the weather is tomorrow, which is uncertain as of now. Consider then looking up a weather forecast online to make the decision - that would be a down-to-earth example of PA.

1.1 Project motives

This project is about making a new tool for PA - why?

A considerable number of application domains have already been explored by PA researchers, from healthcare [179, 59] and education [185, 143] to sales [192, 89] and manufacturing [64, 85]. Still, being a relatively new field, PA has a limited foothold in research and industry, and dedicated tool and system support remain limited to this day [158]. Current PA applications may therefore have to rely on improvised technical solutions.

Gluing together an improvised data-driven workflow, consisting of e.g. a data management layer with separate ML and optimization tools, may introduce various *inefficiencies* and *difficulties*, depending on application type. Performance bottlenecks stemming from I/O between disparate tools, and the development challenges of having to wrestle these hundred-handed monstrosities, are examples of problems that may arise [58].

One particular type of applications in which such issues are exacerbated by a lack of *scalability* is Big Data or *data-intensive* applications. Such applications face e.g. large quantities of complex data arriving at fast rates (i.e. data *Volume*, *Variety*, and *Velocity*) as their main challenges, as opposed to CPU cycle limitations, i.e. compute challenges [90]. Currently, no complete PA solution for data-intensive applications, integrating data management with prediction and optimization, exists [58, 115]. This motivates the principal goal of this project, being the exploration of new tools for *data-intensive PA*.

While Big Data analytics might need new tools for PA, the *need* is arguably mutual, if we consider that PA as a practical field also currently faces serious *usability* challenges.

Let's first paint a rosy-coloured picture of *predictive* analytics, for comparison. As found in our day and age, large quantities of data making up comprehensive, up-to-date histories of complex problem domains, are of great utility for creating predictive analytics solutions that are primarily *data-driven*, as opposed to *expert-driven* [68]. Utilizing data mining techniques, such solutions rely on knowledge *learned* from data as opposed to knowledge *engineered* by experts [71, 100]. Instead of relying on meticulously handcrafted models of real-world dynamics, a bottom-up approach is followed, essentially based on automated pattern recognition, in which "the world is its own best model" - always exactly up to date and complete in every detail [32, p.5].

Of course, we don't mean to say that data analysis requires no expertise or skill - what we're getting at is that one can possibly get by with these skills alone, without needing a doctoral degree or similar for every new complex problem domain under consideration. When we say "expert", the "problem domain" is silent.

The data-driven approach inherent in e.g. ML, has played a significant role

in *democratizing* predictive analytics in the previous decade, enabling many new applications and business models for (especially) non-experts in education and industry [38]. A multitude of high-quality, high-level tools available on the market, along with the *AutoML* movement essentially automating the entire ML workflow, allow users to treat predictive analytics tools as *black boxes*, reducing the barrier to entry considerably [74, 135].

As for PA, the usability situation is comparatively dire, based on what is currently observed in PA research and in industry.

As observed in a recent review of the field, while data-driven prediction is the norm in existing PA research, current prescriptive methodologies are predominantly expert-driven, with optimization models and the like usually being meticulously handcrafted by domain experts, using various heuristics and modelling tricks to ensure computability and correctness [100].

Current PA methodologies being largely expert-driven seems to be echoed by poor usability among existing tools supporting some form of PA. A recent interview study in industry found that PA functionality already available in existing BA software tends to be barely used, *or not used at all* by its intended audience [60]. While recognizing the potential utility of PA, business users attribute this phenomenon to usability challenges with existing tools - they experience not having the right mix of skills and incentive to keep up with these new advanced methodologies, and therefore dismiss them to ”save bandwidth” in the end [60, p.7].

Expert-driven methodologies and poor usability of existing tools form a hindrance for democratizing PA in the same way as predictive analytics was in the previous decade - hindering its adoption in industry, education, and elsewhere. To some authors, the solution to this issue is to make future PA more data-driven in general, as they call for the development of new domain-agnostic methodologies for this purpose, enabled by (specifically) *Big Data analytics* [100].

So to summarize, this project aims to develop a tool for data-intensive PA, being motivated by challenges within scalability and usability both.

1.2 Proposed solution

The solution ultimately developed in this project is a *sample-efficient Black-Box Optimization (sBBO) library for the Apache Spark platform*. What are any of these things, and why is this solution a good idea with respect to the presented motives?

To start, Apache Spark (henceforth just ”Spark”) is the most popular platform for Big Data analytics by several metrics, such as GitHub stars and Stack Overflow activity [26, 180]. It also remarks itself by being very extensible and having great community support [88], and is therefore a natural foundation to build upon for a student project like this.

While Spark currently comes with fully-featured libraries for predictive analytics out-of-the-box, e.g. ML with MLlib, it has no built-in user facilities dedicated to mathematical optimization, or any other methods useful for PA specifically [115]. To support PA workflows better, while keeping the venture reasonably scoped, it was therefore decided to create new third-party PA facilities for Spark in this project, in the form of a new library.

We specifically opt for a Black-Box Optimization (BBO) library (no "s" for now). The reason for exploring this direction is simply that optimization is emblematic to PA [106], and that black-box analytics methods, due to their inherent facilitation of high-level abstraction, have elicited user-centered benefits within the field of ML [38].

Currently, Linear Programming (LP) and variants are the most popular prescriptive methods within PA, by a large margin [100, 115]. While very powerful in its own right, LP requires an explicitly defined (i.e. "white-box"), strictly linear, problem model [11, 91], making it challenging to incorporate e.g. black-box ML models or more "interesting" domain logic into PA workflows [23]. Furthermore, as a recurrent issue in PA research (e.g. [18, 34, 89]), many real-world problems exactly solvable with LP also turn out to be NP-hard, possibly making LP computationally infeasible to use in practical applications.

BBO algorithms generally solve problems *approximately*, thereby dodging the NP-hardness issue, and impose no strict modelling requirements on the user [91]. Their way of operation is also relatively simple to understand: They are all essentially *simulation-based* search strategies, trying out different solution options trial-and-error style [42] - all the user has to do is to give the algorithm a measure of solution quality, i.e. the objective function. With no inherent modelling restriction imposed, data-driven predictive analytics provides a lot of options for doing exactly this in a PA setting: One might for instance call a web API for weather forecasts, execute a what-if OLAP query in SQL [16], do online ML predictions... "Anything goes" inside the black box. In other words: New data-driven, more usable, methodologies for PA based on BBO might be worth looking into.

Existing third-party BBO solutions for Spark are practically all based on *population-based* BBO methods such as Genetic Algorithms or Particle Swarm Optimization (e.g. [199, 109, 36, 108]). Such algorithms iteratively improve a set of candidate solutions by assessing their individual fitness and doing cruel collective experiments based on that [91, 15]. Transferring these methods to the distributed computing architecture of Spark makes it possible to scale out population sizes to unprecedented levels [105], which is great - if we are just dealing with CPU bottlenecks.

Population-based BBO however requires a very large number of objective evaluations to work by design [42, 98]. This is not an issue when the cost of evaluating the objective can be assumed to be negligible. However, the assumption no longer holds when each objective evaluation carries with it significant computational cost

- say, for instance, an analytical query on a large distributed dataset on a Spark cluster; or maybe just time series forecasting with a large Deep Learning model [38]. In other words, population-based BBO is not conductive to the *scalable*, data-driven approach to PA we seek in our data-intensive setting.

The developed library instead focuses on an entirely different family of BBO methods, that we will put under the moniker of *sample-efficient BBO (sBBO)* [11, 42]. So far, these methods have found barely any use in PA applications [73], let alone mainstream data analytics, aside from a niche use case in hyperparameter optimization [3].

Their usual applications instead lie within *scientific simulation* and *engineering design*, within aerospace engineering, for instance [57, 98]. Here, trialing one solution candidate might in extreme cases require running a high-fidelity simulation taking hours or days, which may even fail to return a valid result [11]. Unlike population-based BBO, sBBO methods are designed to be very deliberate about which points to sample in the search space, with some even forming elaborate interpolation models to pick new trials [142].

While no sBBO tool for Spark exists to my knowledge, evaluating e.g. heavy Spark queries during optimization is well within the purview of what sBBO methods are designed for. We have therefore picked them as the focus of our new library - with the ”s” and ”BBO” in sBBO accommodating our scalability and usability motives, respectively.

We name this library *DIBBOlib (Data-Intensive Black-Box Optimization library)*. The name is a *nipioacronym*, in that the abbreviated form kind of sounds like a toddler doing their best to pronounce the unabbreviated one. If this seems absurd, note that such nomenclature is not entirely without precedent within the Hadoop-based ecosystem that Spark is a part of [8]. The motive behind the chosen name is on one hand to make it sound benign and simple, and on the other, to pay homage to DIBBOlib’s (eventually disclosed) strong kinship with the aforementioned MLlib, the go-to ML library for Spark [88].

With our motives and proposed solution described, we arrive at the following problem statement:

Problem statement

How can a scalable and usable sBBO library for Spark be designed, implemented and tested?

What has been presented so far is basically conclusions from a previous semester project [115], yet left to simmer for at bit longer. In the aforementioned venture, I surveyed PA as a research field, including its conceptual foundations, applications, solution methods, its current tool support, and its relationship to the topic of Big Data. The survey served as necessary background for deciding on building a pro-

totype for the library suggested above, which elicited a proof of concept for further work. This project can be regarded as a continuation of the previous one.

The report is structured as follows: First, necessary background for understanding the technical contribution is presented, including preliminaries about Spark, sBBO, and a bit of cooperative game theory (chapter 2). The main part of the report then goes into the design and implementation of DIBBOlib (cf. chapter 3), forming a proposed solution with respect to the problem statement, with a few new ideas to help us get there. The proposed solution is then contrasted with related works (chapter 4), followed by experiments (chapter 5) along with a conclusion and a few considerations about future work (chapter 6).

1.3 Project (de)limitations

Before we jump into it, a few remarks about what this project aims (not) to do are in order.

As encapsulated in the problem statement, the main focus is to develop a new tool with a novel core premise. As we shall see, a recurring theme within this report will be that there are a lot of purely technical challenges involved in making this core premise work: To my knowledge, there has only been very few attempts at creating *general-purpose* tools built around sBBO, let alone scalable and usable ones, and none designed for data-intensive settings specifically [122, 127].

Cutting to the chase: To make space for the sheer amount of technical challenges in this project, use cases and demonstrations of various example applications will *not* be a focus. There will be experiments intimating possible PA use cases within e.g. sales and marketing along with engineering design (cf. chapter 5) - but the point of those is just to test library features.

Holding my own work at arm's length, the result is admittedly what some people at AAU call "a solution waiting for a problem". But then again, while PA as a whole has shown promise within many different domains in applications research, it is still arguably more of a *promise* of a practical field than anything else outside of academia [99, 115]. This project is an early attempt at providing a more solid practical foundation for PA by *technology-push* as opposed to *demand-pull* [81]. The latter mechanism has hitherto been the main driver of innovation within PA applications research, which vastly dominates the field in terms of page count [58]. Both mechanisms are complementary in technological progress, and I simply picked one side of the see-saw to make this whole venture more manageable.

Bibliographical remarks

It should be noted that section 2.1 re-uses section 7.2 of my pre-specialization project [115, p. 41-44].

Chapter 2

Background

This chapter covers necessary background for understanding the solution developed in this project. It covers technical preliminaries about the two main loose ends to be joined in this project: The Spark platform and sBBO. It also includes a small detour into the area of cooperative game theory, as this was used for developing a special feature for the library.

2.1 Apache Spark

This solution developed in this project extends an existing platform, i.e. Apache Spark. Necessary preliminaries are covered in this section. It should be emphatically stated that ***all text within the next corresponding pair of quotation marks is directly copied from section 7.2 of my pre-specialization project [115, pp. 41-44]***, in concordance with the rules for re-use specified by AAU during Spring, 2022. Back in my pre-specialization project, I made the following statement, cited here:

"Spark is a unified analytics engine for large-scale data processing on computer clusters, providing high-level APIs in Scala, Java, Python, SQL, and R [169]. Every *Spark application* (cf. fig. 2.1) consists of a *driver* process running the entry-point function of a user-provided program. Through a *session object*, a *SparkContext* or *SparkSession* depending on the chosen API [88], the driver program can specify various parallel operations, or *tasks*, to be run on a cluster by a number of *executor* processes. These may operate on large partitioned data sets stored in distributed file systems like the Hadoop Distributed File System (HDFS). The driver requests computing resources for running tasks from a *cluster manager* [166].

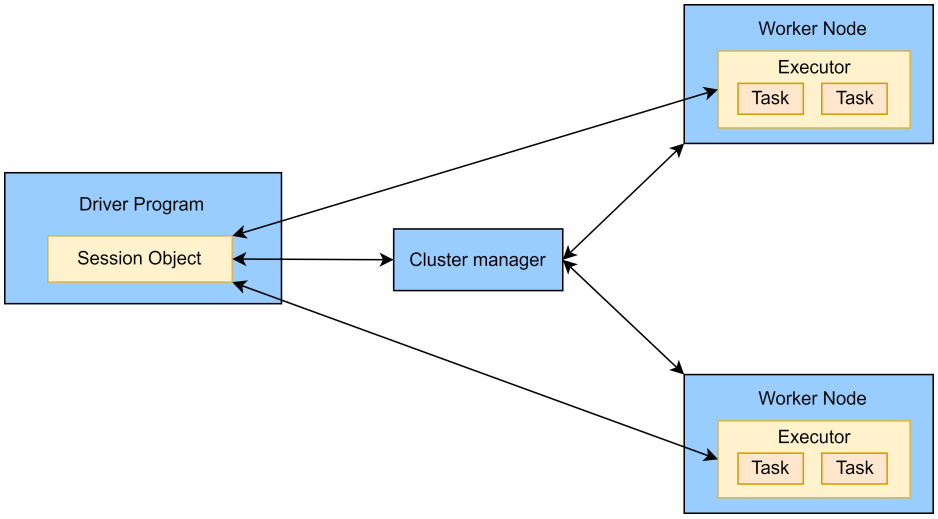

Figure 2.1: Structure of a Spark application, adapted from [166].

The key abstraction for specifying parallel computation is the *Resilient Distributed Dataset* (RDD), representing a typed collection of partitioned elements that executors can operate upon in parallel [170].

Fundamental types of RDD operations include *transformations* and *actions*. Transformations, e.g. the higher-order function *map*, can be construed as mere specifications of computations that are *lazily evaluated* and return another RDD to be operated upon by further operations. Actions, like *reduce*, are on the other hand *eagerly evaluated*, i.e. they elicit the actual operations specified on the RDD and return or display some result [88].

This snippet of a driver program written in Scala illustrates the idea:

```
1 val lines = sc.textFile("data.txt")
2 val lineLengths = lines.map(s => s.length)
3 val totalLength = lineLengths.reduce((a, b) => a + b)
```

It lazily reads an input .txt file into an RDD in the SparkContext *sc* on line 1, and lazily specifies the computations of individual line lengths with *map* on line 2. Only the *reduce* action on the final line actually elicits any task execution on the cluster, ultimately calculating the total length of the .txt file [170]

Spark SQL is an SQL API built on top of RDDs, which has effectively superseded the RDD API over time. It offers high-level declarative SQL-like operations on table-like *Dataset*s, while providing logical and physical query plan optimizations with the extensible Catalyst optimizer, whenever an action is called for [88].

While the elements of a Dataset can be strongly typed objects defined by users, the most prevalent solution is to use the more generic Dataset[Row] variant, which is also called a *DataFrame* [172].

Example usage of Spark SQL, counting individual words in a DataFrame with column named "l", containing lines of text, is found below [88, p. 9]:

```
1 val words = lines.select(explode(split(col("l"), " ")).as("w"))
2 val word_counts = words.groupBy("w").count()
```

What has been explained so far pertains to Spark Core and the Spark SQL engine (cf. fig. 2.2). However, the officially supported Spark stack also notably comes with various libraries, such as MLlib, which contains common facilities for ML workflows, from preprocessing to model management [167].

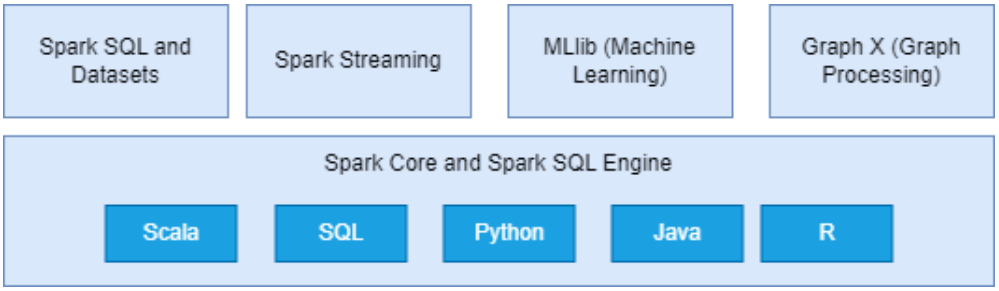

Figure 2.2: Apache Spark components and API stack, adapted from [88, p. 7].

In its current iteration, MLlib operates on Datasets [167]. An example driver program training a model and performing inference can be found below [88, p. 8]:

```
1  import org.apache.spark.ml.classification.LogisticRegression
2
3  // Instantiate SparkSession as "spark"
4
5  // Load and prepare training and test data from HDFS.
6  val training = spark.read.csv("hdfs://...")
7  val test = spark.read.csv("hdfs://...")
8
9  // ...
10
11 // Declare and fit the model
12 val lr = new LogisticRegression()
13 val lrModel = lr.fit(training)
14
15 // Predict
16 lrModel.transform(test)
```

LogisticRegression is an *Estimator*, which upon calling its *fit* method will learn model parameters from its input Dataset (line 13). The returned *lrModel* is a *Transformer*, not to be confused with the aforementioned transformation operation type. Calling the Transformer's *transform* method on the input DataFrame (line 16) will

compute the predictions given by the ML model, and return them as an additional column of the DataFrame [88]. Another key usage of Estimators and Transformers in MLlib is to perform various preprocessing tasks, such as data cleaning and normalizing values in columns [167].

Several Estimators and Transformers can be arranged into *Pipeline*s. Pipelines are themselves Estimators encapsulating a sequence of Transformers and Estimators. Typically, their *fit* method will be called on an input training data set, which then in sequence applies all Transformers and Estimators on it [168]. In the case of Estimators, they are *fit* to the input and then *transform* it with the derived model, adding e.g. an extra column with an ML prediction. A *PipelineModel* is ultimately returned from fitting a Pipeline, which is a Transformer that encapsulates the entire flow for new input data, with Estimators being replaced by their corresponding Transformer models created during the Pipeline fitting process [88].

Such a PipelineModel can be saved and loaded on demand, which allows for easier model management in production settings. An annotated snippet illustrating the usage of the pipeline system can be found below [168]:

```
// Create pipeline with two preprocessing steps and a logistic
   regressor (lr):
val pipeline = new Pipeline()
  .setStages(Seq(tokenizer, hashingTF, lr))

// Fit the pipeline to training data:
val model = pipeline.fit(training)

// Now we can optionally save the fitted pipeline to disk
model.write.overwrite()
  .save("/tmp/spark-logistic-regression-model")

// Later...

// Load it back in production:
val sameModel = new PipelineModel
  .load("/tmp/spark-logistic-regression-model")

// And make predictions on data.
sameModel.transform(productionData)
```

Note that a PipelineModel, being a Transformer, is itself a valid Pipeline stage. Thus, there is potential for assembling complex workflows from simpler components. Furthermore, the *SQLTransformer* (doing any Spark SQL query on its input) [174], user-defined Transformers, etc. provide a lot of expressiveness. In general, pipelines can express *Directed Acyclic Graph* (DAG) workflows, its sequential ordering being a topological sort of the dependencies between input and output columns of its stages [168]" [115, pp. 41-44].

This concludes the content re-used from section 7.2 of my previous project.

2.2 Optimization problems

Necessary preliminaries about optimization as such are covered here.

2.2.1 Problem types

At the onset, the library aims to support a broad class of optimization problems. To introduce some key terms without too much clutter, we first define the *basic optimization problem*, in the style of [91]:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & x \in \mathcal{X} \end{aligned} \tag{2.1}$$

Let a *design point* x be a vector of n *decision variables* $[x_1, x_2, ..., x_n]$. The *solution* of the problem is a design point x^* in the *feasible set* $\mathcal{X}$ such that $f(x^*) \leq f(x)$ for all $x \in \mathcal{X}$, i.e. the value of the *objective function* $f : \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$, or "objective" for short, is *minimized*.

Note that the minimization problem with objective $-f(\cdot)$ has an equivalent solution to the maximization problem with objective $f(\cdot)$ [91, p. 5]. Therefore, to cut the length of this chapter in half, everything henceforth presented assumes minimization problems, unless otherwise noted, without loss of generality. As an additional remark, the objective being allowed to evaluate to ∞ is sometimes a handy technique for denoting e.g. evaluation failures in practical applications [11].

Many practical challenges stem from the definition of the feasible set $\mathcal{X}$, only provided implicitly so far. $\mathcal{X}$ may simply be $\mathbb{R}^n$, and we may deem the problem *unconstrained*. Yet the feasible set can more generally be modelled as a subset of $\mathbb{R}^n$ through a set of predicates $\{c_1(\cdot), ..., c_k(\cdot)\}$ called *constraints* [91, p. 167]:

$$\mathcal{X} = \{x \in \mathbb{R}^n \mid c_1(x) \wedge ... \wedge c_k(x)\} \tag{2.2}$$

For instance, constraining some decision variables to be integers is often useful, since this enables the representation of *discrete choices* in the formal problem model, including the selection of elements from *enumerable sets* such as $\{"this", "one"\}$ or $\{true, false\}$ [120]. Furthermore, to limit the sheer size of the search space and ensure a meaningful solution (e.g. avoid negative-length geometry), it is often a good idea to bound possible values in the domain under consideration [49]. This elicits the *mixed-integer bound-constrained problem* [136, 30]. In all that follows, let the notation $[1..n]$ denote the set of integers $\{1, 2, ..., n\}$. Then the problem can be defined by:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & x_i \in \mathbb{Z} \quad \forall i \in \mathcal{I} \\ & l \leq x \leq u \end{aligned} \tag{2.3}$$

Where n is (still) the number of decision variables, $\mathcal{I} \subseteq [1..n]$ is the set of vector indices of all integer decision variables, and $l, u \in \mathbb{R}^n$, $l < u$, are vectors of *inclusive* lower and upper bounds for all decision variables. Compared to the problem in eq. (2.1), the difference in eq. (2.3) is that a particular feasible set has now been explicitly defined.

Note that variable bounds defined by strict inequalities (i.e. < and >) are usually avoided within optimization, since many incremental algorithms can get "stuck" at the boundary of such open feasible sets, making infinitesimally small improvements to suboptimal solution candidates [91, p. 7].

The constraints allowed so far only concern the domain of individual decision variables. Ultimately, we wish to up the expressiveness of constraints supported by the library significantly. As it turns out, any constraint can be rewritten to an *inequality* or *equality* with respect to zero [91, p. 169]. For instance, consider the constraint $x_1^2 \geq x_2^2$, which can be rewritten to $x_2^2 - x_1^2 \leq 0$, or that Boolean expressions evaluated to zero or one are straightforwardly comparable to zero. The *generally constrained problem*, to be supported by the library, can be formally defined as follows [15, 42]:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & x_i \in \mathbb{Z} \quad \forall i \in \mathcal{I} \\ & l \leq x \leq u \\ & g_j(x) \leq 0 \quad \forall j \in [1..p] \\ & h_k(x) = 0 \quad \forall k \in [1..q] \end{aligned} \tag{2.4}$$

Where we now have *constraint functions* $g_j, h_k : \mathbb{R}^n \to \mathbb{R}$ with $p, q \in \mathbb{N}$ being the number of inequality and equality constraints, respectively. Everything else is the same as in eq. (2.3).

Constraint functions expressed like inequalities or equalities with respect to zero can potentially quantify *how far* a given design point is from being feasible. For instance, $|h_k(x)|$ is a quantitative measure of infeasibility for equality constraint k with respect to point x. We formalize the notion of a *violation value* for a constraint function $c(\cdot)$ with respect to a design point x as follows:

$$violation(c,\ x) = \begin{cases} \max[0,\ c(x)] & \text{if c is an inequality constraint} \\ |c(x)| & \text{if c is an equality constraint} \end{cases} \tag{2.5}$$

Note that the point being feasible with respect to the constraint elicits a violation value of zero, while infeasibility elicits positive violation values. Violation values are useful for implementing several constraint handling strategies [42].

The final complication to be handled by the library is that objective and constraint functions in eq. (2.4) may be *black boxes*. This notion is more practical than formal in nature:

Definition 2.1 (Black Box) *Within optimization, a process that can only be observed in terms of its inputs and outputs, all inner workings being analytically unavailable.*

Classical examples of real-world processes modelled as black boxes may be computer simulations or laboratory experiments [11, p. 5].

Regarding the optimization problem in eq. (2.4), we now introduce a practical constraint on possible solution *methods*, as opposed to formal constraints on the solution as such. For instance, solution algorithms cannot directly exploit any derivative information of the objective function, which precludes using algorithms like gradient descent.

More precisely, the only information available about a black-box function $B(\cdot)$ to a solution method is a set of pairs $\{(x_1, B(x_1)), ..., (x_k, B(x_k))\} \subseteq B$, i.e. some corresponding domain and range values in the functional relation of $B(\cdot)$ [15].

2.3 sBBO approaches

We now turn to a brief overview of solution methods for sBBO developed within optimization research, since the workings of these methods impact our design space at the highest level of significance. For ease of exposition, we assume *unconstrained problems* in this section, i.e. $\mathcal{X} = \mathbb{R}^n$, unless otherwise specified.

2.3.1 Important caveats

As a consequence of the *No Free Lunch Theorem* [197], a library designed around a small proper subset of all optimization algorithms in existence will not be the right tool for all problems in existence.

One specific point in terms of algorithmic efficiency and performance should immediately be emphasized: *The BBO methods presented below will generally never outperform modern optimization methods leveraging additional analytical problem structure, if applicable* [11, 42].

For instance, modern LP methods can possibly handle millions of decision variables efficiently [91]. On the other hand, state-of-the art sBBO solvers can *at most* be expected to handle *hundreds* of variables efficiently for *mixed-integer bound-constrained problems* (cf. eq. (2.3)), judging from recent performance studies [136, 150]. Similarly, top-of-the-line constraint handling techniques, such as the

Lagrange multiplier method, tend to rely on analyzing the derivatives of constraint and objective functions (cf. eq. (2.4)) [102, 91]. In the general case, this is simply *impossible* to do in sBBO, which must consequently rely on less efficient plan B's.

As for population-based BBO, higher-dimensional problems with more general constraints are also generally speaking within reach than what is the case for sBBO [125, 126].

Our reasons for going with sBBO were covered in chapter 1, being the possibility of providing a good combination of usability and scalability within our data-intensive setting, as opposed to offering on-paper superior performance while disregarding limitations of human beings or Spark clusters.

2.3.2 Overview and taxonomy

At the most general level of description, sBBO algorithms are all iterative, improving *incumbent solutions* incrementally. However, this characterization is only slightly more specific than how iterative algorithms work in general, and therefore calls for refinement.

Confer algorithm 1, which is my own attempt at a synthesis: sBBO algorithms generally start out with a (dummy) initial solution (line 2). They then try to improve the incumbent solution in iterations by proposing possible replacements (line 4), assessed through the objective function (line 5-8). Finally, at the behest of some termination condition, the best verified solution is returned (line 9).

Algorithm 1 Generic Black-Box Optimization

```
1: procedure GBBO
2:     solution ← initialize()
3:     while not terminate() do
4:         candidates ← propose(solution)
5:         for candidate in candidates do
6:             evaluated ← objective(candidate)
7:             if better(evaluated, solution) then
8:                 solution ← evaluated
9:     return solution
```

For ease of understanding, we furthermore distinguish between four different sBBO approaches along two dimensions, both denoting different kinds of *search strategies*: *Direct* vs. *Model-based* and *Global* vs. *Local*. This taxonomy combines similar distinctions from e.g. [25, 42, 11].

Direct methods rely solely on elements in the relation of the objective function to guide the search for the solution [91]. Their defining feature and moniker is perhaps easier to understand when considering their Model-based counterpart, which in a sense operates *indirectly*: Model-based methods additionally leverage

properties of a *surrogate* model of the objective function to guide the search [25]. The surrogate is usually a function acting as a "well-behaved" (i.e. differentiable, noiseless, cheaply evaluated...) stand-in for the objective in *auxiliary optimization problems* solved during optimization [145].

As for the other dimension, Global methods aspire to find a solution among all elements in the feasible set, i.e. the *global optimum*, as required by all problems defined in section 2.2.1. Local methods instead only aspire to find a solution x^* such that $f(x^*) \leq f(x)$ and $\|x - x^*\| < \delta$ for some $\delta > 0$ and $x \in \mathcal{X}$, i.e. a *local optimum* [91, p. 7]. The reason for including Local methods into the library, despite calling for the global optimum in section 2.2.1 stems from the fact that their combination with other techniques can elicit viable global optimization strategies [42]. In general, these different sBBO approaches all have their own strengths and weaknesses, and can be combined into hybrid approaches to great effect [11].

A high-level overview of how each method type works, along with a few algorithmic examples, follow below.

2.3.3 Local/Direct methods

For Local/Direct methods, two prevalent approaches exist: *directional* and *simplicial* [42]. As per algorithm 1, both rely on having an *initial solution guess*, which is then improved in iterations by evaluating the objective value of a set of points in a *geometric pattern* around the incumbent solution [150]. Iterations continue until the rate of improvement drops below a threshold, the incumbent solution is satisfactory, the objective evaluation budget has been expended, or some other termination criterion holds [91].

At algorithmic iteration k, directional methods evaluate a set of points in a *pattern* $P^{(k)}$, defined as projections of the incumbent solution $x^{(k)}$ with a set of *directions* $D^{(k)}$ and *step size* $\alpha^{(k)}$ [42, p. 119]:

$$P^{(k)} = \{x^{(k)} + \alpha^{(k)} d \mid d \in D^{(k)}\} \tag{2.6}$$

The incumbent solution for the next iteration $x^{(k+1)}$ is $x^{(k)}$ if no evaluated point has a better objective value than the current incumbent, and is otherwise the point with the best objective value among all evaluated points in $P^{(k)}$ [11, p. 116]. Some algorithms accept any improvements *opportunistically* and end the iteration upon finding a better incumbent solution, while others assess all pattern points before proceeding [98].

The projection operation eliciting new evaluation points from the incumbent solution is technically called a *line search*, and is not unique to directional sBBO. It is also commonly used by gradient-based algorithms, such as gradient descent, to project the incumbent solution with a direction and step size derived from first or second order gradient information, e.g. the direction of steepest descent [91].

Without the luxury of derivatives, directional methods must determine the set of directions, as well as the right step size, by different means. The general approach employed by existing algorithms is to define $D^{(k)}$ as a *positive spanning set* of $\mathbb{R}^n$. This means that any point in the $\mathbb{R}^n$ vector space can be constructed as a *non-negative* linear combination of the directions in $D^{(k)}$ [150, 98]. Of note, evaluating points in a positive spanning set of directions around $x^{(k)}$ guarantees that at least one of them lies in a descent direction, if such a direction exists around $x^{(k)}$ [91, p. 103]. In other words: directional methods employ a kind of "zeroth-order gradient descent" strategy, using objective evaluations only.

The exact set of directions constructed varies by algorithm. A simple baseline approach is to always set $D^{(k)}$ to be the set of column vectors in $[I\ {-I}]$, where I is the $n \times n$ identity matrix [42, p. 115]. This set of directions is used in the *Hooke-Jeeves method*, sometimes called the *coordinate* or *compass search method* due to how it can be visualized in $\mathbb{R}^2$ (cf. fig. 2.3) [91].

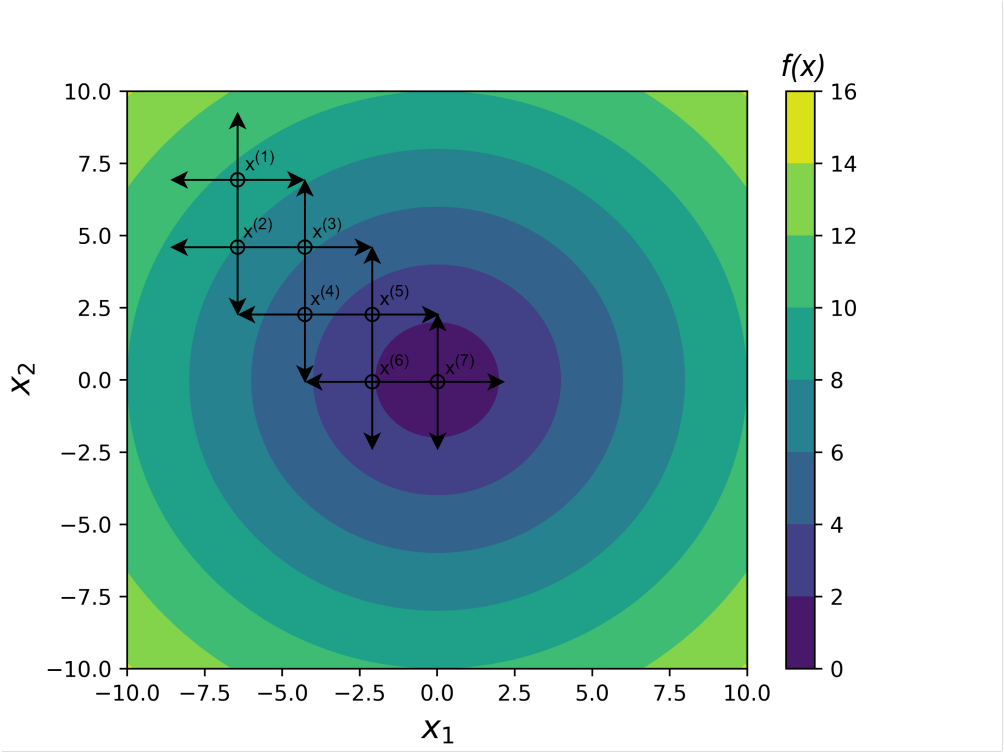

Figure 2.3: Contour plot illustrating seven iterations of compass search, minimizing the objective $f(x) = x_1^2 + x_2^2$ with $x^{(1)}$ as the initial incumbent. The color map approximates different objective values in the search space. Arrowheads denote candidates proposed by line searches, while circled crosses denote those additionally accepted as incumbent points. Starting from $x^{(1)}$, the algorithm travels South, East, South, East, South, and East, before arriving at $x^{(7)}$, the global minimum.

State-of-the-art methods for defining positive spanning sets try to make them

smaller than e.g. the compass search method to elicit fewer objective evaluations per iteration, make them more robust against pathological objective functions, or generate them stochastically to provide asymptotic convergence guarantees [10, 91].

As for the step size, the typical approach is to set $\alpha^{(1)} \in \mathbb{R}_{>0}$ and then at the end of each iteration potentially increase or decrease $\alpha^{(k)}$ by a factor $\delta \in \mathbb{R}_{>0}$, depending on whether or not the incumbent solution was improved, respectively [42]. Intuitively speaking, this provides a self-correcting mechanism against overshooting or undershooting the step size, i.e. missing promising descent directions nearby or converging unnecessarily slowly towards far-away minima, respectively. Heuristics trying out additional line searches in improvement directions on each iteration with an extra large step size scale factor, thereby potentially accelerating convergence, are also common [11].

The *Nelder-Mead Simplex method* and its subsequent improvements, is what sBBO textbooks generally regard as the set of simplicial methods [98, 42]. This aforementioned simplex method is not to be confused with the Danzig Simplex method for LP!

A simplex is a generalization of a three-dimensional tetrahedron (a triangular pyramid shape) to n-dimensional spaces. It is the most simple geometric object with flat sides possible in its corresponding space: a point in zero dimensions, a line segment in one dimension, a triangle in two dimensions, and so on [91]. Note that an n-dimensional space requires $n + 1$ points to represent the vertices of its simplex type.

The basic idea of the Nelder-Mead method is to maintain an n-dimensional simplex, consisting of $n + 1$ known solution points, which is twisted and contorted by various heuristics to fit the local landscape of objective values, suggesting slopes to follow towards the local minimum [42]. To complete the mental image, this algorithm is also known as the *amoeba method* [139, p. 402].

A single algorithmic iteration updates the simplex by evaluating a combination of four geometric transformations: *reflection*, *expansion*, *contraction*, and *shrinkage* [91]. Let "best" and "worst" here refer to the relative value of the objective function among the simplex vertices, and the *centroid* be the mean of all vertices except the worst one.

Confer fig. 2.4. Reflection and expansion obtain possible replacements to the worst vertex by reflecting it over the centroid of the simplex at different step sizes. Contraction obtains a possible replacement to the worst vertex by moving it towards the centroid [91]. Shrinkage projects all other vertices towards the best vertex. The simplex tends to shrink down in size as it approaches the local minimum - therefore, the algorithm is usually set to terminate whenever the simplex has shrunk to a sufficiently small size, according to a threshold. The returned solution

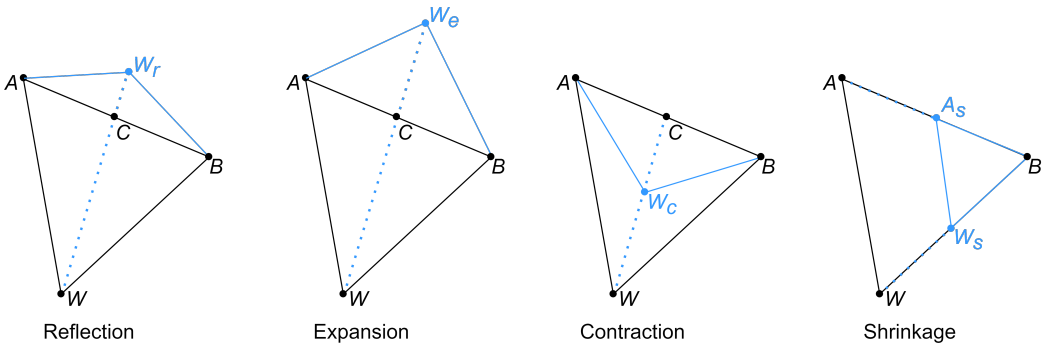

Figure 2.4: $\mathbb{R}^2$ examples of the four elementary simplex transformations used in the Nelder-Mead algorithm. A, B, and W are the three vertices of the simplex, with B and W being the best and worst points, respectively. C is the centroid of A and B. Dashed blue lines indicate projection directions, while solid lines indicate the shape of the (projected) simplex (example adapted from [91, p. 108]).

is the best vertex of the final simplex [42].

It should be noted that the original version of the Nelder-Mead method can exhibit pathological behaviour in various scenarios, e.g. the hypervolume of the simplex becoming zero far away from the local minimum [91]. It is also sometimes regarded as a heuristic method, since it doesn't guarantee local convergence in all cases [11]. Nevertheless, many improvements to the simplex operations have been suggested over time, and the algorithm remains a staple of popular libraries to this day (e.g. [56, 129, 154]).

2.3.4 Global/Direct methods

Unlike Local/Direct methods, Global/Direct ones form a motley crew with no similar taxonomic distinctions. One unifying theme that goes for all Global optimization algorithms is the need to balance *exploration* and *exploitation* of information in the entire search space to find a global minimum [93, 200]. Without surrogate models, Global/Direct methods rely on e.g. stochastic convergence guarantees or search space partitioning strategies for this [25].

Simulated Annealing, Genetic Algorithms, Particle Swarm Optimization, and similar stochastic algorithms, form the major part of what would be considered Global/Direct methods within BBO at large [126, 125], yet being designed around cheap objective evaluations, these don't fall within our scope.

As for alternative approaches, *DIvided RECTangles (DIRECT)* [87] is a widespread method within sBBO research and tools, with strong convergence guarantees under certain assumptions [93, 98, 42]. Note that bound-constrained search spaces (like the one defined in eq. (2.3)) are *hyperrectangular*, i.e. "box-shaped" in n dimensions. DIRECT scales such search spaces to the *unit hypercube*, i.e. with all variable intervals scaled to unit length, and then proceeds to evaluate midpoints of

evenly sized rectangular subregions, always dividing into thirds [86] (cf. fig. 2.5).

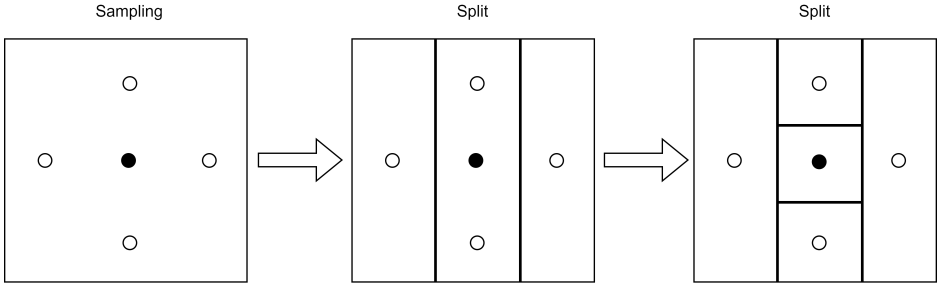

Figure 2.5: DIRECT in action in $\mathbb{R}^2$, sampling points before splitting the original search space into thirds, prioritizing the largest subregions with the best potential objective values [91, p.116].

Evaluating these midpoints makes it possible to calculate *lower bound estimates* of objective values in each subregion. This is done by assuming a *constant upper bound* on the (unknown) regional rate of change, i.e. assuming *Lipschitz continuity* [87]. The obtained information is used *heuristically* to prioritize promising regions for further subdivisions, balancing exploration and exploitation by using the aforementioned lower bound estimates [91, p. 113].

2.3.5 Local/Model-based methods

Local and Global Model-based methods are both neatly characterized by their own overarching framework: the *trust region framework* [42] and *surrogate-based optimization framework* [72], respectively, to be discussed in turn.

Methods within the first framework maintain a surrogate model approximating the objective function in a particular *trust region*. At iteration k, the trust region $\mathcal{T}^{(k)}$ is simply a set of points sufficiently close to an incumbent center point $x^{(k)}$ by some distance metric $\|\cdot\|_p$ and maximum radius $\Delta^{(k)}$ [195, 42]:

$$\mathcal{T}^{(k)} = \{x \in \mathcal{X} \mid \|x - x^{(k)}\|_p \leq \Delta^{(k)}\} \tag{2.7}$$

The L_p norms, parameterized by $p \geq 1$ include commonly used distance metrics, and are defined as follows [91, p. 439]:

$$\|x\|_p = \lim_{q \to p} (|x_1|^q + |x_2|^q + ... + |x_n|^q)^{\frac{1}{q}} \tag{2.8}$$

The limit is needed for handling the case when $p = \infty$. As examples, the Manhattan, Euclidean, and Chebyshev distance are the L_1, L_2, and L_∞ norms, respectively.

By some as of yet undisclosed means, suppose that we have obtained a surrogate model, a function $\hat{f}^{(k)}(\cdot)$ approximating the objective function in the trust

region of the current iteration. To find the next point for objective evaluation, trust region methods solve the following auxiliary optimization problem [195]:

$$\underset{x \in \mathcal{T}^{(k)}}{\text{minimize}} \quad \hat{f}^{(k)}(x) \tag{2.9}$$

That is, we are looking for the best possible solution in the trust region, according to the surrogate. This can usually be done efficiently, since the surrogate model is deliberately chosen to be cheap to evaluate, differentiable or have other attractive properties for solving the auxiliary problem [150]. Also, notice the difference from the line searches of directional sBBO (cf. section 2.3.3), in that we effectively need to pick a direction and (bounded) step size with respect to an incumbent point *simultaneously* in this problem.

Given a solution x' to the auxiliary problem, we then evaluate its actual objective value $f(x')$ and update the center and radius of the trust region for the next iteration, obtaining $x^{(k+1)}$ and $\Delta^{(k+1)}$, respectively [98]. This update is done based on the *improvement ratio*, a measure of how accurate the surrogate model was in finding a better solution [42, p. 178]:

$$\eta = \frac{\text{actual improvement}}{\text{expected improvement}} = \frac{f(x^{(k)}) - f(x')}{\hat{f}^{(k)}(x^{(k)}) - \hat{f}^{(k)}(x')} \tag{2.10}$$

Details of the update rules differ among algorithms. Yet intuitively speaking, a high improvement ratio encourages exploration, while the opposite encourages local refinement.

If the improvement ratio is above some minimum threshold, the trust region center $x^{(k+1)}$ is set to a new promising point, e.g. x' or some known point y such that $f(y) \leq f(x')$. Otherwise, $x^{(k+1)} = x^{(k)}$. Likewise, the new trust region radius $\Delta^{(k+1)}$ is increased or decreased with a sufficiently high or low improvement ratio, respectively, and is otherwise equal to $\Delta^{(k)}$ [195, 11].

Common termination criteria for trust region methods include the trust region radius getting sufficiently small, the gradient of the surrogate model getting reliably close to zero at the center point (thus indicating a "valley bottom"), or the objective evaluation budget being expended [42].

Next, we turn to how the surrogate model is managed within the trust region framework. A vector of *model coefficients* α is obtained for a surrogate model $\hat{f}_\alpha(\cdot)$ by either solving an *interpolation* or *regression* problem with respect to a set of m sample points in the current trust region with known objective values $\mathcal{S} = \{s_1, ..., s_m\} \subseteq \mathcal{T}^{(k)}$ [93, 150]. This usually entails solving a system of equations.

Interpolation requires fitting the model perfectly to all sample points (using a trivial objective function) [42]:

$$\begin{aligned} &\underset{\alpha}{\text{minimize}} \quad 0 \\ &\text{subject to} \quad \hat{f}_\alpha(s_i) = f(s_i) \quad \forall i \in [1..m] \end{aligned} \tag{2.11}$$

Regression requires solving the least squared error optimization problem [11]:

$$\underset{\alpha}{\text{minimize}} \quad \sum_{i=1}^{m} [\hat{f}_\alpha(s_i) - f(s_i)]^2 \tag{2.12}$$

While a strict advantage of perfect fits might seem evident, regression may be preferred due to its ability to incorporate more sample points without jeopardizing well-defined solutions to the approximation, thus possibly being more robust to noisy objectives than perfect fits to less information [42].

Yet in practice, since the expensive objective use case typically implies few available samples, a quadratic model interpolation of the following form is usually preferred [11, p. 172]:

$$Q(x) = c + g^\intercal x + \frac{1}{2} x^\intercal H x \tag{2.13}$$

Where $c \in \mathbb{R}$, $g \in \mathbb{R}^n$, and symmetric matrix $H \in \mathbb{R}^{n \times n}$ contain model coefficients. Unlike a purely linear model, a quadratic model can fit local curvature of the objective function, while coincidentally also being one of the only non-linear models in existence for which the *exact global solution* to the auxiliary problem in eq. (2.9) can be computed by known efficient methods [195].

However, obtaining this model requires solving equation systems with $\frac{1}{2}(n+1)(n+2)$ unknown coefficients (mind the *symmetric* matrix), and thus at least the same number of objective evaluations in the current trust region to ensure a unique solution, which may be costly [11].

Furthermore, sample points must collectively satisfy strict geometric conditions for the interpolation problem to be solvable while convergence to a local optimum is ensured [195]. As a consequence, updating the surrogate model to approximate changing trust regions properly, requires incorporating one or more *model improvement steps* in each algorithmic iteration, ensuring that the sample points in $\mathcal{S}$ can be used for interpolation within the current trust region [11]. Model improvement algorithms provably terminating in a finite number of steps have been proposed, possibly requiring extra objective evaluations for sample set replacements, or decreasing the radius of the trust region for safer approximation [42].

Another issue with trust region methods is that they are not outright compatible with discrete decision variables, since their proofs of convergence rely on properties of continuous variables [11, p. 223].

Due to such issues, emerging work within trust region optimization investigates alternative surrogate models or quadratic approximation methods [195, 25].

2.3.6 Global/Model-based methods

The *surrogate-based optimization framework* (SBO) encompasses Global/Model-based sBBO. As with Local/Model-based methods, it approximates the objective function, and solves auxiliary optimization problems to guide the search. Yet similarly to Global/Direct methods, it seeks to balance global exploration and exploitation of search space information, and not just seize local opportunities.

At a high level of description, SBO consists of two phases: *Sampling* followed by *optimization* [145, 200]. The sampling phase evaluates the objective function at a strategic set of points. This basically forms a *sketch* of the entire search space, which is used by the surrogate model in the subsequent optimization phase to propose additional evaluation points. Both phases are discussed here in turn.

Design of Experiments

The initial phase follows a particular sampling plan or *Design of Experiments (DoE)* with a given objective evaluation budget, to decide which points to evaluate [92, 200]. Recommendations differ, yet the sampling budget should at least accommodate minimal conditions for proper interpolation or regression of the surrogate, which varies by model [120, 94]. Beyond this amount, one proposed heuristic is to use about $\frac{1}{3}$ of the entire budget on the initial sampling phase for suitable coverage of the search space [57].

Given a particular sampling budget, a reasonable initial idea to have for a DoE is simply to evaluate a corresponding number of *random samples* within the bounds of the search space (cf. eq. (2.3) a)). The problem with this approach is that it cannot ensure any notion of *coverage* - e.g. all points might just as well end up in the same local search space region (cf. fig. 2.6), lobotomizing the surrogate predictor [91].

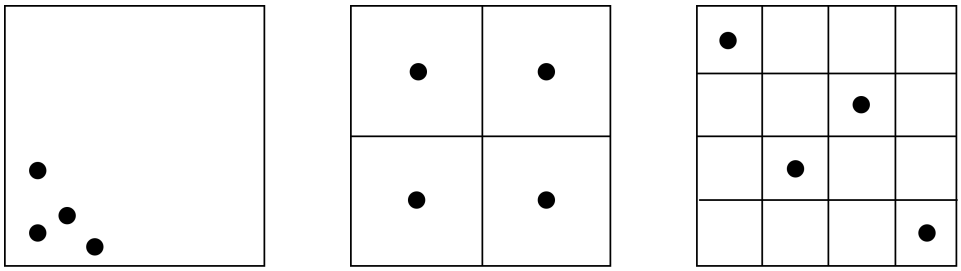

Figure 2.6: 3 possible ways to sample 4 points in $\mathbb{R}^2$. Random sampling provides no coverage guarantees. Full factorial does, but scales poorly with more than a few problem dimensions. LHS forms a middle ground.

To ensure coverage, another reasonable initial idea is to instead evaluate a *grid* of evenly spaced points in the search space, with a preset m samples per dimension

(cf. fig. 2.6 b)). The problem with this approach is that it is generally too cavalier with the evaluation budget: Following this so-called *full factorial* DoE with n decision variables requires m^n evaluations [72, 200]. Somewhat perplexingly, "factorial" here refers to how each dimension *factors* into the aforementioned exponential requirement. With an increasing number of dimensions, ensuring coverage by this method quickly becomes too costly.

The most established DoE methodology is instead based on *Latin Hypercube Sampling (LHS)*. Like full factorial designs, LHS models the search space as a grid. Yet instead of sampling all m^n cells, LHS only samples a subset of size m that is *uniformly distributed* along each dimension [142, 92]. In $\mathbb{R}^2$, this intuitively means that no two sample points share a column or row in the grid, ensuring a notion of coverage (cf. fig. 2.6 c)). A *Latin square* is an $m \times m$ grid in which each row and column contains all integers in the interval $[1..m]$ [193]. Sudokus and Latin hypercubes are more specific and general cases of Latin squares, respectively. LHS is generally implemented by randomly permuting integer intervals for each dimension, ultimately obtaining m sample indexes of the full factorial grid [91]. This randomness implies that several sampling plans are possible for each problem formulation, and edge cases with poor coverage do occur. Therefore, extensions to LHS have been proposed for counteracting them [200, 119].

Global surrogate models

Moving on to the optimization phase, an *adaptive sampling* strategy is employed with the surrogate model to suggest new evaluation points [200]. On each iteration, one or more new sample points are suggested for evaluation by solving auxiliary problems with the surrogate model. The suggested points are then evaluated with the true objective function and possibly incorporated into the surrogate approximation for future iterations. This process continues until the total evaluation budget is spent, or some other termination criterion is met [70, 25, 65].

To a large extent, the chosen surrogate model decides all other elements of an adaptive sampling strategy, and is therefore discussed first. *Radial Basis Function (RBF)* and *Gaussian Process (GP)* interpolants, along with variations thereof, are the most popular surrogate models used in computationally expensive optimization by far [94, 72]. These will therefore be the focal points of this exposition. Traditional alternatives, such as quadratic models (e.g. eq. (2.13)) and Support Vector Regressors have in a way been "superseded" by RBFs and GP's according to several authors, due to a lack of ability to capture irregular objective function landscapes, prohibitive approximation costs, or brittle approximation conditions [57, 25]. Combining *ensembles* of several surrogates, producing weighted predictions, has also been proposed [70, 157], but we will ignore this complication here.

Given a set of pairwise distinct sample points $\mathcal{S} = \{s_1, ..., s_m\}$, an RBF interpolant takes the following general form [51, p. 5]:

$$RBF(x) = \sum_{i=1}^{m} \lambda_i \phi(\|x - s_i\|_2) + p(x) \tag{2.14}$$

Where $\lambda \in \mathbb{R}^m$ contains model coefficients, $\phi : \mathbb{R}_{\geq 0} \to \mathbb{R}$ is the *radial basis* or *kernel* function, and $p : \mathbb{R}^n \to \mathbb{R}$ is a polynomial function containing additional model coefficients.

Note that $\phi(\cdot)$ is used as a function of the Euclidean distance between the input point and a particular sample point. The "radial" moniker refers to the fact that this usage makes values of $\phi(\cdot)$ constant for input points on the same hypersphere around the sample point center [195]. Intuitively speaking, the interpolated objective value for an input point ends up being based on a sum of factors weighted by (usually unit-scaled) distances to known points. This allows capturing complex, multi-modal objective landscapes, in a way that has been compared to how music synthesizers combine artificial signals to form organic timbres [57, p. 54].

Common radial bases, taking as their input the radius r, include the linear kernel $\phi(r) = r$, cubic kernel $\phi(r) = r^3$, and thin-plate spline $\phi(r) = r^2 \log(r)$ [67, 91]. While there is no formal consensus on which kernel is "the best", informal observations would suggest that the cubic kernel is a recurrent default recommendation among researchers [25, 51, 27].

The role of the polynomial tail $p(\cdot)$ is to ensure that the interpolation problem has a unique solution (avoiding matrix singularity), thereby making RBF interpolation a robust model for otherwise problematic sample point sets [51, 147]. The minimum required polynomial degree of $p(\cdot)$ is $k - 1$ where k is the kernel order of $\phi(\cdot)$ [67]. For the cubic and thin-plate spline kernels of order 2, a *linear tail* of degree 1 is thus sufficient, leading to this more concrete interpolant [27, p. 375]:

$$RBF(x) = \sum_{i=1}^{m} \lambda_i \phi(\|x - s_i\|_2) + a^\intercal x + b \tag{2.15}$$

Where $a \in \mathbb{R}^n$ and $b \in \mathbb{R}$ contain model coefficients. The model coefficients in λ, a, and b in eq. (2.15) are obtained by solving the following system of equations [25, p. 254]:

$$\begin{bmatrix} \Phi & P \\ P^\intercal & 0 \end{bmatrix} \begin{bmatrix} \lambda \\ c \end{bmatrix} = \begin{bmatrix} F \\ 0 \end{bmatrix} \tag{2.16}$$

Where $\Phi \in \mathbb{R}^{m \times m}$ is a pairwise kernel distance matrix such that $\Phi_{ij} = \phi(\|s_i - s_j\|_2)$ and:

$$P = \begin{bmatrix} s_1^\intercal & 1 \\ \vdots & \vdots \\ s_m^\intercal & 1 \end{bmatrix} \quad \lambda = \begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_m \end{bmatrix} \quad c = \begin{bmatrix} a_1 \\ \vdots \\ a_n \\ b \end{bmatrix} \quad F = \begin{bmatrix} f(s_1) \\ \vdots \\ f(s_m) \end{bmatrix} \tag{2.17}$$

The time complexity of solving this problem by LU decomposition is $\mathcal{O}(m^3)$ [51].

As for *Gaussian Processes*, these are special surrogates in that they not only provide predictions, but also allow one to measure the uncertainty of said predictions. The core model assumption is that for a finite set of sample points $\mathcal{S} = \{s_1, .., s_m\}$, their associated objective values are a random variable sampled from a *multivariate normal (Gaussian) distribution*, parameterized with a mean vector and covariance matrix of the following form [91, p. 277]:

$$\begin{bmatrix} f(s_1) \\ \vdots \\ f(s_m) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu(s_1) \\ \vdots \\ \mu(s_m) \end{bmatrix}, \begin{bmatrix} \kappa(s_1, s_1) & \dots & \kappa(s_1, s_m) \\ \vdots & \ddots & \vdots \\ \kappa(s_m, s_1) & \dots & \kappa(s_m, s_m) \end{bmatrix} \right) \tag{2.18}$$

Where $\mu(\cdot)$ and $\kappa(\cdot)$ are the chosen *mean* and *covariance* or *kernel function*, respectively. These functions are essentially what sets GP's apart from "vanilla" multivariate Gaussian distributions [1]. In practice, $\mu(\cdot)$ is usually just chosen to be a constant function returning e.g. zero [31, 2]. While the kernel function has many options of significant consequence, the go-to choice is the *squared exponential kernel* [51, 65]:

$$\kappa(s, s') = \exp\left(-\frac{\|s - s'\|_2^2}{2\gamma^2} \right) \tag{2.19}$$

Where $\gamma \in \mathbb{R}_{>0}$, the *characteristic length scale*, is a smoothening hyperparameter that should ideally fit the smoothness of the underlying data. Readers are right to notice a resemblance between the pairwise kernel distance matrix of eq. (2.16) and the covariance matrix derived from eq. (2.18) and eq. (2.19). In fact, some GP variants are regarded as special cases of RBF models [72].

Suppose that we already have a non-empty set of sample points S along with a vector of known objective values y, and now we wish to predict a vector of objective values $\hat{y}$ of a new set of points S^* - i.e. we need an objective function surrogate. We can solve this problem by modelling it as a "$\hat{y}$ given y scenario", a *conditional multivariate Gaussian distribution* [91, p. 276]:

$$\hat{y} \mid y \sim \mathcal{N}(\mu_{\hat{y}|y}, \Sigma_{\hat{y}|y}) \tag{2.20}$$

In Bayesian terms, this is alternatively called the *posterior distribution*, and SBO with GP models is therefore oftentimes called *Bayesian Optimization* [14, 137].

Coincidentally, the mean vector and covariance matrix of the posterior distribution can be calculated in closed form, and by using $\mu(\cdot)$ and $\kappa(\cdot)$ we have all the required information to do so [91, p. 281]:

$$\mu_{\hat{y}|y} = M(S^*) + K(S^*, S)K(S, S)^{-1}(y - M(S)) \tag{2.21}$$

$$\Sigma_{\hat{y}|y} = K(S^*, S^*) - K(S^*, S)K(S, S)^{-1}K(S, S^*) \tag{2.22}$$

Where by $M(\cdot)$ and $K(\cdot)$ we mean:

$$M(X_p) = \left[\mu(x_1), ..., \mu(x_p)\right] \tag{2.23}$$

$$K(X_p, X_q') = \begin{bmatrix} \kappa(x_1, x_1') & \dots & \kappa(x_1, x_q') \\ \vdots & \ddots & \vdots \\ \kappa(x_p, x_1') & \dots & \kappa(x_p, x_q') \end{bmatrix} \tag{2.24}$$

Alternatively, we can write eq. (2.23) and eq. (2.24) as functions of an individual point s [91, p. 281]:

$$\hat{\mu}(s) = \mu(s) + K(s, S)K(S, S)^{-1}(y - M(S)) \tag{2.25}$$

$$\hat{v}(s) = K(s, s) - K(s, S)K(S, S)^{-1}K(S, s) \tag{2.26}$$

These give us an expected value of s, the mean $\hat{\mu}(s)$, which is the desired objective interpolant, along with the *variance* of the predicted mean, $\hat{v}(s)$. Intuitively speaking, the variance quantifies *uncertainty* about the expected objective value. The *standard deviation* $\hat{\sigma}(s) = \sqrt{\hat{v}(s)}$ is particularly useful for explaining this. For instance, under a normal distribution, 95% of all samples are expected to fall within the *confidence interval* of $\hat{\mu}(s) \pm 1.96\hat{\sigma}(s)$ [71]. Informally, we can say that we are 95% certain that the objective value "falls" within this region according to our model. Uncertainty, i.e. the confidence region interval, tends towards zero as s gets closer to known sample points [91].

As a practical note, observe how the formulas of eq. (2.25) and eq. (2.26) may require doing matrix inversions of time complexity $\mathcal{O}(m^3)$ as with the RBF model fitting problem of eq. (2.16), yet *on prediction time*. Still, partial results may be pre-computed between several prediction runs for higher efficiency.

Adaptive sampling strategies

Now we turn to the question of how adaptive sampling strategies are formed from using surrogate models during the optimization phase [200, 70]. It may seem tempting to simply fit the surrogate to known samples, and then proceed to solve the original optimization problem as an auxiliary problem to find new samples for evaluation, replacing the original objective with the surrogate, which is likely cheaper to evaluate.

The problem with this *prediction-based exploration* approach is that it ignores the inherent value of exploring unknown regions for potentially better solutions than

what the contours of the initial DoE sketch would suggest - possibly wasting objective evaluations on reaffirming the value of samples close to the initial frontrunners [91].

Prevalent strategies instead take unknown regions into account during the optimization phase. RBF and GP models strategies pivot around *distance* and *uncertainty* measures to achieve this, respectively [70].

The *Stochastic RBF (SRBF)* strategy [147], and refinements thereof (e.g. [148, 94]), is seemingly the most popular strategy for RBF surrogates currently, judging from its prevalence in libraries and research both [25, 141, 184], being computationally cheaper than previous strategies which required solving problems similar to the one in eq. (2.16) *for each evaluation candidate point* [67]. For surrogate management, SRBF borrows ideas from the trust region framework, discussed previously (cf. section 2.3.5). On each iteration, a set of candidate points for evaluation $\mathcal{C} = \{c_1, ..., c_m\}$ is generated by perturbing the coordinates of the incumbent best solution point with samples from the univariate normal distribution $\mathcal{N}(0, \sigma^2)$ [51, 147]. Similarly to the trust region radius, σ is used as a radius around the incumbent point and is increased or decreased between iterations based on the predictive accuracy of the surrogate (cf. eq. (2.10)). Within a Global sBBO setting, this intuitively helps the algorithm consider local opportunities properly before taking its business elsewhere.

To pick the next points for evaluation, the following auxiliary problem, optimizing the *weighted distance merit*, is solved [178, 149]:

$$\underset{c \in \mathcal{C}}{\text{minimize}} \quad w \cdot s(c) + (1 - w)(1 - d(c)) \tag{2.27}$$

Where $s : \mathbb{R}^n \to [0,\ 1]$ is the predicted surrogate function value of its input, and $d : \mathbb{R}^n \to [0,\ 1]$ is the minimum distance between the input point and any previously evaluated sample point, with the values of both functions being normalized among the candidate points to lie in the unit interval. $w \in [0, 1]$ is simply one of a set of weights that are cycled through between algorithmic iterations, e.g. $\{0.3, 0.5, 0.8, 0.95\}$, to put different emphasis on exploration and exploitation [141]. The content of this set is a hyperparameter. Intuitively speaking, this auxiliary problem prefers points with a good predicted objective value that are far away from known points, by varying weightedness between iterations.

Strategies revolving around GP's seek to reduce uncertainty as well as improve the incumbent solution. A quite analogous idea to the weighted distance measure above is to weigh uncertainty by a constant factor, and solve the following auxiliary problem on each iteration, in which the objective is known as the *Lower Confidence Bound* [91, p. 293]:

$$\underset{x \in \mathcal{X}}{\text{minimize}} \quad \hat{\mu}(x) - \alpha\hat{\sigma}(x) \tag{2.28}$$

Where $\alpha \in \mathbb{R}_{\geq 0}$ is a user-specified hyperparameter. This way, even seemingly

suboptimal points according to $\hat{\mu}(\cdot)$ might be sampled if large local uncertainty suggests that their objective values could possibly be far better than the mean.
Another popular strategy for picking the next point for evaluation is to choose the one that maximizes the *Expected Improvement (EI)* of the current best objective value [51, 91, 93]. The EI of a point can be evaluated analytically under the posterior distribution of the GP, and thus optimized [72, p. 353]:

$$\underset{x \in \mathcal{X}}{\text{maximize}} \quad \begin{cases} [f(x^+) - \hat{\mu}(x)] \cdot CDF(z) + \hat{\sigma}(x) \cdot PDF(z) & \text{if } \hat{\sigma}(x) > 0 \\ 0 & \text{if } \hat{\sigma}(x) = 0 \end{cases} \tag{2.29}$$

Where x^+ is the current best solution, and

$$z = \frac{f(x^+) - \hat{\mu}(x)}{\hat{\sigma}(x)} \tag{2.30}$$

That is, z is the z-score of $f(x^+)$ with respect to the posterior distribution of objective values for design point x, quantifying deviation from the mean of the standard normal distribution. Furthermore, $CDF, PDF : \mathbb{R} \to [0,1]$ are the *cumulative distribution function* and *probability density function* of the standard normal distribution, respectively.

One can use e.g. Genetic Algorithms or any other preferred optimization algorithm to solve the auxiliary problems of eq. (2.28) and eq. (2.29) efficiently, and pick the best points for subsequent evaluation [31, 141]. Once these points are evaluated, they can be incorporated into the surrogate for further iterations, and the cycle goes on, until some termination criterion is met [200].

2.4 Constraint handling in sBBO

We now return to the murky morass of constraint handling, specifically within sBBO.

Several authors within sBBO research have noted that current constraint handling techniques are very limited in terms of efficiency and applicability to different algorithms [70, 57].

At the onset, generally constrained optimization is an NP-hard problem [161]. Within BBO specifically, we face additional *practical* issues if we want to support it: Remember, the structure of predicates to be satisfied by a solution may not even be known analytically, since constraint functions are black boxes in the general case (cf. section 2.2).

Subsequent sections will introduce a comprehensive taxonomy of how sBBO constraints are encountered in the wild, followed by a presentation of established constraint handling strategies applicable to sBBO.

2.4.1 A taxonomy of constraints

The *QRAK taxonomy* by Le Digabel and Wild [49] provides a useful overview of different constraint types encountered within practical settings. This model seems to be widely acknowledged within the sBBO research community [11, 30, 98].

As opposed to the formal distinctions between e.g. equality and inequality constraints provided in section 2.2, QRAK instead describes how information about design point (in)feasibility is practically available during an *in vivo* optimization run. The four letters represent four binary dimensions, classifying the different cases. Each dimension is described below in turn.

Q is for *Quantifiable vs. Nonquantifiable*. This distinction denotes whether constraint function values (cf. eq. (2.4)) signify degrees of (in)feasibility or are simply binary indicators of feasibility as such [49]. For example, an inequality constraint function with "1" as an output might mean that a decision variable was too large by a value of one, or, due to $1 \neq 0$, the solution was infeasible, and that's it. Note that we generally don't know what exactly (possibly black-box) quantifiable constraints quantify, but only care about *relative comparisons*: Quantifiable constraints make it possible to form a preference between several infeasible design points, if we need to, based on relative infeasibility levels [11].

R is for *Relaxable vs. Unrelaxable*. For a Relaxable constraint $c_r(\cdot)$, outputs of the objective function and all other constraint functions will still be *meaningful*, even though a design point is infeasible with respect to $c_r(\cdot)$ [49]. The definition of "meaningful" is problem-specific in nature. For instance, consider an objective function calculating the volume of a swimming pool with a negative side length, which violates an Unrelaxable constraint. According to the negative objective value, denoting the total price of installation, such a swimming pool seems to be a lucrative investment for you. After all, the total price is "1,200 DKK per cubic meter", as advertised. But the pool guy having to pay *you* for building a negative-volume swimming pool doesn't make any sense. The negative objective value of such a solution can't be meaningfully interpreted. On the other hand, if you require that construction begins on a Tuesday, a price calculated from a Wednesday launch, while violating a constraint, may still be meaningfully interpreted within the domain of consideration, and the constraint is therefore Relaxable. Note that while there might be a temptation to correlate Relaxable constraints with "mild preferences", we still always need a feasible solution in the end. The practical difference between the two cases is that Unrelaxable constraints must be satisfied for *all* intermediate solutions found in algorithm 1 to properly guide optimization, while Relaxable constraints only need to be satisfied for the *final* solution [11].

A is for *A Priori vs. Simulation-based*. The feasibility of an A Priori constraint can be assessed without running simulations (e.g. black-box functions), while a Simulation-based constraint can't [49]. In my own terms, the practical difference between the two is that the former type can be assessed *analytically*, while the

latter must be assessed *empirically*. For instance, bound constraints limiting the domain of individual variables (cf. eq. (2.3)) might be possible to assess a priori, without evaluating the objective function, as they depend only on the *input* to the objective function and some user-specified thresholds that are typically analytically available. On the other hand, any constraint regarding the *output* of e.g. a black-box objective function can only be assessed by observing the outcome of an objective evaluation. With black boxes being expensive to evaluate in the general case, the practical implication is that we might be able to cancel expensive computations, if a constraint is deemed violated a priori [11].

The final *K* is for *Known vs. Hidden*. Not all constraints are necessarily known within sBBO. A known constraint is *explicitly* given in the problem definition, i.e. as a distinct part of the definition of the feasible set, while a Hidden one isn't. Characteristically, the presence of a Hidden constraint is only known *implicitly*, upon its violation [49]. In the most well-behaved case, the objective function might simply evaluate to ∞, or some other extremely bad value, to indicate that one or more Hidden constraints are violated, and that any other solution is preferable [42]. In more pathological cases, objective evaluation simply crashes or returns an inconspicuous value well within the *meaningful* range of the problem. A real-world example of the latter was an industrial project in which the objective function returned a hard-coded value of 2.6 whenever a subroutine failed to complete evaluation, perplexing a group of researchers for a while [11, p.VI]. Unrecoverable errors during black-box evaluation, especially connected to running legacy code, are typical sources of Hidden constraints [49].

2.4.2 sBBO constraint handling methods

This section outlines general constraint handling strategies that have achieved foothold within sBBO research, judging from what is covered by textbooks and surveys within this area (e.g. [91, 42, 98, 93]).

Note that such works, due to their summarizing nature, tend to highlight strategies that work at some level of generality beyond the individual algorithm. Constraint handling strategies tailored to individual algorithms might arguably have higher performance potentials, due to their ability to exploit fewer layers of indirection. There is also an abundance of existing research within the development of such tailored methods. E.g., for GP-based SBO, it has been proposed to fit surrogate models to *each* constraint function on the fly, and then proceed to minimize the total "expected constraint violation" in compound auxiliary problems (e.g. [157, 70, 138]).

However, providing a decent baseline level of constraint handling support for many different algorithms was deemed a design priority for the intended library. From an engineering standpoint, achieving a level of *decoupling* between specific

algorithms and constraint handling strategies in DIBBOlib was also deemed desirable, to simplify design issues and ease development. Therefore, while the above-mentioned efficiency trade-off should be recognized, a more generic approach to constraint handling was ultimately preferred, which reflects the material covered here.

Note that the presented methods below only concern the handling of equality and inequality constraint functions (cf. eq. (2.4)). That is, we consider the generally constrained optimization problem of the form:

$$\begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g_j(x) \leq 0 \quad \forall j \in [1..p] \\ & h_k(x) = 0 \quad \forall k \in [1..q] \end{aligned} \tag{2.31}$$

This stems from the fact that the types of constraints inherent to mixed-integer bound-constrained problems (cf. eq. (2.3)) can easily be handled as *Unrelaxable*, *A Priori*, *Known* constraints within practical applications [11]. That is, we can just check and ensure (through e.g. rounding) user-specified variable bounds and integer mappings before each objective evaluation to ensure meaningful and feasible solutions with respect to these constraints. This design has seen success in practical settings [120, 51].

Penalty methods

Penalty methods convert constrained problems into unconstrained problems by including a penalty term in the new objective function [91]. Problems of the form in eq. (2.31) are rewritten like so [102, p. 421]:

$$\underset{x\in\mathbb{R}^n}{\text{minimize}} \quad f(x) + \rho \cdot P(x) \tag{2.32}$$

Where $P : \mathbb{R}^n \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is called the *penalty function*, and $\rho > 0$ is the *penalty magnitude*. Some variant formulations make the magnitude a vector in $\mathbb{R}_{>0}^{p+q}$ consisting of individual magnitudes applied to each constraint function [43], which is a complication we ignore here.

A simple choice for the penalty function is the *count penalty*, counting the number of violated constraints with Boolean predicates evaluated to zero or one [91]:

$$P_{count}(x) = \sum_{j=1}^{p} (g_j(x) > 0) + \sum_{k=1}^{q} (h_k(x) \neq 0) \tag{2.33}$$

Another common choice within sBBO is the *death penalty* [43]:

$$P_{death}(x) = \begin{cases} \infty & \text{if } P_{count}(x) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.34}$$

Yet another example is the *quadratic penalty* [11] unlike the others use violation values for quantifying degrees of infeasibility (cf. eq. (2.5)):

$$P_{quadratic}(x) = \sum_{j=1}^{p} \max(0,\ g_j(x))^2 + \sum_{k=1}^{q} h_k(x)^2 \tag{2.35}$$

As the story tends to go, none of these penalties is ideal in all scenarios. While the death penalty might seem like a universal sledgehammer solution, note that infinite objective values generally lead to untenable numerical issues in Model-based sBBO [57].

A key advantage of *exact penalties* like the death and count penalty is that they both guarantee that there exists a *finite* value of ρ for which a solution to eq. (2.32) is exactly a solution to the corresponding problem of the form in eq. (2.31) [93]. The problem with these methods is that they contribute to making the objective function non-smooth, and thus harder to optimize for known algorithms, which tend to assume a level of smoothness [98, 44]. *Inexact penalties*, like the quadratic one, can only guarantee a solution equivalent to one in the original constrained problem as $\rho \to \infty$, but have the advantage of being smooth functions on the border of the feasible region, which generally makes solving the unconstrained problem in eq. (2.32) easier [93].
One proposed compromise is to instead combine several penalties into the *mixed penalty* [91]:

$$P_{mixed}(x) = P_{count}(x) + P_{quadratic}(x) \tag{2.36}$$

This is an exact penalty ensuring a level of smoothness at the same time.

As for the magnitude ρ, its value can be set *statically*, i.e. before running optimization, if a sufficiently large magnitude for a feasible solution happens to be known for the problem at hand [43]. More realistically, however, a *dynamic* approach to calibrating the right magnitude during optimization is needed. The general idea is to solve a sequence of unconstrained problems of the form [102]:

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad f(x) + \rho_i \cdot P(x) \tag{2.37}$$

Where $[\rho_i]$ is an increasing sequence of magnitudes, with some initial value $\rho_1 \in \mathbb{R}_{>0}$. With the solution found at iteration i being x_i^*, this procedure terminates when $P(x_i^*) = 0$ (returning x_i^* as the solution), or an upper limit on the number of iterations has been reached (failing to find a feasible solution). For $i > 1$ and some *adjustment factor* $\gamma \in \mathbb{R}_{>1}$, setting $\rho_i = \gamma \cdot \rho_{i-1}$ or $\rho_i = \rho_{i-1} + \gamma \cdot P(x_{i-1}^*)$ are example options for increasing the magnitude dynamically [91, 44].

While the advantage of penalty methods is that they in principle always work, and unlike other methods do so *orthogonally* to the algorithm of choice, a significant disadvantage for sBBO with expensive objectives is the sheer computational cost of

solving several problems from scratch with dynamic methods. If only something could be done about that.

Barrier methods

Note that dynamic penalty methods with progressively increasing magnitudes tend to approach solutions in the feasible region of the original problem from the *outside*, and are therefore alternatively named *exterior point* methods. Barrier methods, also known as *interior point* methods, instead approach solutions of the original constrained problem from *inside* its feasible region [102].

Barrier methods operate very similarly to penalty methods, *yet only work with inequality constraints*, among other key distinctions [91]. Constrained problems of the form given in eq. (2.31), with $q = 0$, are rewritten to unconstrained subproblems of this form (note the fraction this time):

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad f(x) + \frac{1}{\rho} \cdot B(x) \tag{2.38}$$

$B : \mathbb{R}^n \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is a *barrier function*, having special properties. $B(\cdot)$ is chosen such that it is continuous and non-negative in the feasible region of the original problem. Note that the value of a barrier function is not necessarily zero for feasible points, Furthermore, $B(x) \to \infty$ as $x \in \mathcal{X}$ approaches the boundary of the feasible region [93].

Intuitively speaking, the reason that barrier methods only support inequality constraints stems from the fact that narrow feasible regions delineated by equalities don't have such "approachable" boundaries. The reasonable suggestion of simply rewriting equalities into several inequalities, since $h_i(x) = 0 \iff h_i(x) \leq 0 \wedge -h_i(x) \leq 0$, has proven unfruitful in practice, due to the prevailing search region narrowness of such rewrites [91].
Common barrier functions include the rebranded *extreme barrier* [11]:

$$B_{extreme}(x) = P_{death}(x) \tag{2.39}$$

Alternatively, the *log barrier* is a popular choice [91]:

$$B_{log}(x) = -\sum_{j=1}^{p} \log[-\max(-1,\ g_j(x))] \tag{2.40}$$

Dynamic barrier methods also work analogously to dynamic penalty methods, with an increasing sequence of magnitudes $[\rho_i]$ [102]:

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad f(x) + \frac{1}{\rho_i} \cdot B(x) \tag{2.41}$$

Assuming a barrier besides the extreme barrier, increasing the ρ_i *eases* the influence of the barrier, especially its rapidly increasing punishment as points approach the feasible boundary. Therefore, the early termination condition for solving dynamic barrier subproblems is instead that x_i^* is (approximately) equal to x_{i-1}^*, implying that loosening the thumbscrews of the barrier function no longer elicits better feasible solutions [91].

The usual numerical problems associated with infinite objective values still hold for Model-based sBBO. On the other hand, barrier methods are well-suited and popular for directional sBBO: Starting with a *known feasible point* as the initial solution (cf. algorithm 1), line searches into the infeasible region will simply face an electric fence of much worse solutions without numerical issue [42]. It should be noted that this usage of an initial solution as an anchor point for *never leaving the feasible region* during search is what tends to disambiguate extreme barrier methods from death penalty methods in optimization literature (e.g. [11]) - yet this is admittedly a subtle difference.

Advantages of interior methods compared to exterior methods include that they are typically preferred above using e.g. the death penalty, when applicable, as they provide an efficient way of handling (especially Unrelaxable) constraints without wasting many evaluations on exploring the infeasible region [102, 91].

Main disadvantages include that barrier methods only handle inequality constraints, and that the underlying optimization algorithm must be able to cope with extreme objective values, limiting its generality compared to exterior methods.

Filter methods

An inherent problem with both exterior and interior point methods is that the right magnitude of ρ needs to be determined before or during optimization, likely at the cost of extra objective evaluations.

Filter methods, and extensions thereof, use an entirely different approach, treating the original problem in eq. (2.31) like, *but not exactly like*, an unconstrained *bi-objective* optimization problem, incorporating the original objective function and some penalty function $P(\cdot)$ as separate components [55]:

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad [f(x),\ P(x)] \tag{2.42}$$

This approach opens the can of worms known as *domination*, a concept used within generalized multi-objective optimization for excluding design points *strictly worse* than others as potential solutions [91]. In our specially modified case, domination concerns pairs $(f^{(i)}, P^{(i)}) \in \mathbb{R} \cup \{\infty, -\infty\} \times \mathbb{R} \cup \{\infty, -\infty\}$. A pair $(f^{(i)}, P^{(i)})$ *dominates* another $(f^{(j)}, P^{(j)})$ if and only if $f^{(i)} \leq f^{(j)} \wedge P^{(i)} \leq P^{(j)}$, i.e. the former pair is no worse than the latter in either value. Otherwise, the latter pair is *non-dominated* by the former (either of the latter's components being strictly better) [138].

A *filter* is a set of pairwise non-dominated pairs of the aforementioned type. In the filter framework, a filter is maintained during optimization, essentially forming a current "benchmark" for new solution candidates [98]. The benchmark concerns whether a new proposed design point is *acceptable* to the filter. In the basic version, a point x is acceptable to the current filter $\mathcal{F}$, if no element in $\mathcal{F}$ dominates $(f(x), P(x))$ - that is, the new point is strictly better than current elements of the filter in either measure [138]. If this new pair is acceptable to the filter, we update the filter by removing all pairs dominated by the new pair.

Note that unlike classical multi-objective optimization, in which the goal is to find the best possible *trade-off* between multiple objectives, a solution x^* such that $P(x^*) = 0$ is required in the filter approach, which is a significant departure [55]. It means that we generally can't just implement $better(\cdot)$ in algorithm 1 on the basis of new candidate points being acceptable to the current filter, according to the basic rule described before. In practice, more elaborate rule sets must be designed for managing the state of the filter *and the underlying optimization algorithm*, ensuring that optimization won't just e.g. converge to infeasible solutions, or cycle through the same set of states without converging [12, 93].

In other news, while the filter approach has been applied to several specific sBBO algorithms [76, 157, 11], circumventing a key problem inherent to exterior and interior methods, this comes at the cost of removing a layer of indirection, making implementation of non-trivial filter extensions for individual algorithms a potential cross-cutting concern for, say, an sBBO library for Apache Spark.

2.5 Cooperative game theory and prediction models

We finally embark on a final detour to the topic of *cooperative game theory*, since we plan on incorporating some as of now undisclosed *tricks* from this domain into the library.

Unlike other areas of game theory, which tend to focus on individual player choices during the course of the game itself, cooperative game theory instead concerns itself with the ultimate outcome of some game given the initial coalition of players - that is, black-box problems in a familiar sense (cf. definition 2.1). More formally, it concerns itself with *coalitional games* [196]:

Definition 2.2 (Coalitional game) *A tuple* (N, v)*, where* $N = [1..n]$ *is the set of players, and* $v : \mathcal{P}(N) \to \mathbb{R}$ *such that* $v(\emptyset) = 0$ *is the characteristic function.*

$v(S)$ intuitively denotes the expected total *worth* or *payoff* obtained from the game by a coalition of players $S \subseteq N$ working as a team, with N also being known as the *grand coalition*. Naturally, the payoff is zero when no one plays the game.

2.5.1 Shapley values

With one foot in the real world, we might expect that players in a coalition neither contribute evenly nor linearly to the payoff of the game in the general case, due to e.g. (anti-)synergistic effects derived from teamwork. Therefore, if we wish to estimate the contribution of individual players in a coalition S to the ultimate payoff, simply dividing $v(S)$ by $|S|$ may violate a notion of *fairness*.

Shapley values, named after mathematician and Nobel Prize-winning economist Lloyd Shapley, is a way of estimating the contribution of individual players in the grand coalition N, ensuring a notion of fairness that (provably) no other attribution scheme does completely [196].

To make better sense of things for now, imagine that the grand coalition is formed one player at a time, with each player collecting their rightful share of the payoff in some order. More formally, let Π be the set of all permutations of N, i.e. all possible ways to order players in the grand coalition. For $\pi \in \Pi$ and some player $i \in N$, then let $p_i^\pi = \{j \mid j \in N \wedge \pi(j) < \pi(i)\}$ be the set of all players preceding player i in permutation π. We then say that $v(p_i^\pi \cup \{i\}) - v(p_i^\pi)$ is the *marginal contribution* of player i in the order π [117].

The Shapley value $\phi_i(v)$, attributing the individual *worth* or *influence* of a player i in a coalitional game $(N,\ v)$, is then the *average marginal contribution* of player i across all $n!$ permutations of the grand coalition [196]:

$$\phi_i(v) = \frac{1}{n!} \sum_{\pi \in \Pi} [v(p_i^\pi \cup \{i\}) - v(p_i^\pi)] \tag{2.43}$$

If we account for the fact that the order of joining a coalition doesn't matter to the outcome of the game, then p_i^π represents the same set of players S joining player i exactly $|S|! \cdot (n - |S| - 1)!$ times in eq. (2.43) [196]. Here, the first and second factor represent the number of ways to permute preceding and subsequent players in the relevant orders, respectively. We thus obtain the more commonly known version of the Shapley value formula [104]:

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \left[\frac{|S|!(n - |S| - 1)!}{n!} \cdot [v(S \cup \{i\}) - v(S)] \right] \tag{2.44}$$

More clearly now, there is weighting scheme at play, designed such that small and large cardinalities of S are assigned the greatest weights, with the least weight assigned to middle-sized coalitions.

As an example, for $n = 5$, the weights assigned to coalition sizes from 1 to 5, always including player i, are correspondingly $[24,\ 6,\ 4,\ 6,\ 24] \cdot \frac{1}{5!}$. An intuition behind this pattern is that we get more information about the contribution of individual players in the grand coalition when fewer players are present or absent in marginal estimates [117].

Shapley values *uniquely* satisfy several properties connected to a notion of fairness - this is great, but we won't dwell too much on that [196, 183]:

2.5.2 SHAP values

SHapley Additive exPlanations, or SHAP values, are simply put Shapley values of a specific kind of game, i.e. characteristic function (cf. definition 2.2).

They are used for interpreting the outputs of predictive models, e.g. ML models, as part of the mission statement of *Explainable AI (XAI)*. The game being played is a *singular* prediction of the model, outputting some real value, and each player in the grand coalition corresponds to a particular input feature value. The goal is to explain how much each feature value contributed to the prediction [117].

For example, given that we among other features have provided a Boolean input value of *true* to a model, and the output prediction reads 42, we wish to know whether this Boolean feature value contributed 0, 3.50, -1000, 9001, or otherwise to the output in *additive* terms, despite any predictive noise or non-linearity.

Models obtained from e.g. ML generally need all features present to do predictions. Therefore, a practical challenge is that we need a way to calculate marginal contributions of individual feature values when one or more other feature values are "missing" (cf. eq. (2.44)). SHAP values, basically defined as a proposed remedy to this issue [104], are Shapley values derived from *conditional expectations* of model outputs.

More concretely, we let the grand coalition $N = [1..n]$ correspond to the indices of the input feature vector $x = [x_1, x_2, ..., x_n]$, and then let $v(S) = \mathbb{E}[f(X) \mid X_S = x_S]$ in eq. (2.44) [201]. Here, the predictive model is $f(\cdot)$, $X = [X_1, X_2, ..., X_n]$ is a random vector, and $X_S = x_S$ denotes that we fix random variable components $i \in S$ to the corresponding components of x. This way, we deal with missing "players" in predictive models by fixing present feature values and integrating over the domain of the rest [103].

How exactly conditional expectations are obtained, filling out leftover details of the proposed scheme, largely depends on the type of model studied and practical constraints. With an ML model we may just approximate by taking the mean of a large number of samples, in which we replace missing features with uniformly random values from a reference dataset. For computer vision models, we might analogously grey out missing pixels and take the prediction mean [117].

In addition to general Shapley value properties, SHAP values also notably possess *local accuracy* [104]:

$$f(x) = \phi_0 + \sum_{i \in N} \phi_i \tag{2.45}$$

Where ϕ_i is the SHAP value of feature value i, and $\phi_0 = \mathbb{E}[f(X)]$. That is, each SHAP value ϕ_i forms an *offset* from an expectation across the entire domain of

predictions ϕ_0, which in sum accounts for the obtained prediction completely.

Note that SHAP values are *local* measures of feature importance, for individual feature values and model predictions. If one instead needs a *global* measure of feature importance for a particular feature and model, a natural way of extending the above approach is to simply to calculate SHAP values for m instances and take the mean of their absolute values [117]:

$$\Phi_i = \frac{1}{m} \sum_{i=1}^{m} |\phi_i| \tag{2.46}$$

This is also the approach followed in the *shap* Python library, an implementation of SHAP values by the people behind the idea [156].

2.5.3 SHAP value estimation

Computational complexity and *performance* are significant implementation factors with SHAP values. In the general case, exact calculation takes exponential time, due to the combinatorics of eq. (2.44) and the enumerative implications of how conditional expectations over domains are to be obtained.

For some specific models, like decision tree based ones, SHAP values can actually be calculated in polynomial time [103]. Linear models are even more simple, since the slope coefficients turn out to be SHAP values (cf. eq. (2.45)) [104]. As a practical joke by the cosmic powers that be, such exemplars tend to already be relatively explainable models.

More generally, however, the only practically feasible approach is to estimate SHAP values, based on random sampling. Judging from the source code of *shap*, efficient approximation strategies seem adamant about reducing information overlap between *sequentially* drawn random samples with careful enumerations over feature domains [156]. To the casual observer, the sky seems to be the limit here, when it comes to implementation complexity and optimization opportunities.

We will however try to stay horizontally inclined, to better leverage the parallel architecture of Spark. For this project, a simple model-agnostic strategy based on *Monte Carlo sampling* is used instead, with few dependencies between subtasks, at the cost of some informational redundancy among random samples.

To be faithful to the original presentation by Štrumbelj et al. [183], suppose that we have some reference dataset $Z = \{z_1, z_2, ..., z_p\}$ of points accompanying the input feature vector $x = [x_1, x_2, ..., x_n]$ for which we wish to approximate the SHAP value of feature i for a model $f(\cdot)$.

First, reminisce about the order-based weighting scheme of section 2.5.1 for a moment. Then let Π be the set of all permutation of the set $[1..n]$, i.e. all ways to order the indices of x. For some uniformly randomly chosen permutation $\pi \in \Pi$

and reference point $z \in Z$, then let $\tilde{x}^{+i}$ and $\tilde{x}^{-i}$ be n-vectors defined as follows [183]:

$$\tilde{x}_j^{+i} = \begin{cases} x_j & \text{if } \pi(j) \leq \pi(i) \\ z_j & \text{otherwise} \end{cases} \tag{2.47}$$

$$\tilde{x}_j^{-i} = \begin{cases} x_j & \text{if } \pi(j) < \pi(i) \\ z_j & \text{otherwise} \end{cases} \tag{2.48}$$

That is, we end up with two randomly modified versions of x, switching in components of z at indices after i in order π, with the only difference between the result vectors being whether x_i is retained or not. We can create i.i.d. random samples by the same process, with replacement, and take the mean of M *marginal estimates*, echoing marginal contributions, to approximate the Shapley value [117]:

$$\hat{\phi}_i = \frac{1}{M} \sum_{m=1}^{M} [f(\tilde{x}_m^{+i}) - f(\tilde{x}_m^{-i})] \tag{2.49}$$

So, by substituting in random components from the background data set while other components of x remain fixed, we can simulate missing players in the game of prediction and estimate the desired conditional expectation.

For sufficiently large values of M (e.g. conventionally $M \geq 30$), it follows from the Central Limit Theorem that the estimation error with respect to the actual SHAP value ϕ_i is approximately normally distributed, i.e. $\hat{\phi}_i - \phi_i \sim \mathcal{N}(0, \frac{\sigma^2}{M})$, where σ^2 is the *population variance* of marginal estimates for feature i [182]. While the true value of the population variance is generally shrouded in mystery, we can instead just use the *sample variance* of our M marginal estimates in eq. (2.49) as an unbiased estimate of σ^2 [183]. All in all, this means that we can use properties of the normal distribution to reason about the approximation error of a given number of samples.

Chapter 3

Technical contribution

This chapter is an attempt to explain how DIBBOlib works, and why. The exposition follows a top-down structure, starting with some overarching design considerations and ending with an outline of how core features work.

3.1 Library requirements and priorities

Functional requirements of the library are outlined and further materialized in the problem definition and background sections (cf. chapters 1 and 2). That is, at the highest level of generality, the mission statement of the library is to support solving generally constrained sBBO problems defined with respect to Spark Datasets, using the methods and definitions of chapter 2. The library achieving this in any way thus technically fulfills its principal functional requirements.

That would however be too simple to achieve, and may not lead to a "good' solution. We of course also have to deal with the non-functional requirements of *scalability* and *usability* in the problem statement. This section outlines how we plan to go about this on a strategic level.

As popularly laid out by Kleppmann, scalability denotes a system's ability to cope with increasing workloads performance-wise, by e.g. adding more computational resources to it [90, p. 10]. Now that we know all about Spark and sBBO (cf. chapter 2), we can better describe what "increasing workloads" mean in our setting, and what to do about them.

With objective evaluations involving Spark queries, as per the core premise, the scalability of our library depends to some degree on the ability of Spark to handle larger workloads in individual queries, in the shape of more complex SQL operations, larger input datasets, and the like.

In experiments for my pre-specialization project, I in fact observed that about 99% of total runtime was spent doing Spark queries while running sBBO [115], a pretty standard ratio for objective evaluations within sBBO, by the way [11]. On

the other hand, from what was covered in the previous chapter, we know that core sBBO algorithms at their heftiest only involve doing some matrix inversions (cf. section 2.3.6), with a presumably modest number of elements within the sBBO use case. Implementations of these mainly CPU-bound tasks have been fanatically micro-optimized for decades [97].

Beyond trying to integrate smoothly with Spark SQL, it may therefore seem that we can't do much about our main bottleneck in terms of workload, unless we think that we can do data-intensive queries better than Spark SQL in the general case. Given that our problem components containing these Spark queries are black boxes (cf. definition 2.1), i.e. they might do "anything", our general case within sBBO is indeed very general, with no overt opportunities for custom performance optimizations coming to mind beyond the situational.

On the positive side, if one is good at performance-tuning Spark queries, one is by extension probably also good at performance-tuning our library. But from a didactic standpoint, you are probably not happy with me delegating all responsibility for scalability to Spark SQL and calling it a day. We do luckily have a few possible leverage points for improving scalability in sBBO beyond individual query handling (which we do ultimately delegate to Spark SQL).

First, sBBO problems are typically parameterized with a *trial budget* in practical settings - i.e. a maximum number of times we can assess the solution quality of design points [141, 122]. This is indeed a workload parameter we might be able to improve scalability for. All things being equal, a system doing trials entirely in sequence will be slower and slower with increasing trial budgets. Supposing that we have plenty of computational resources available on the cluster, this is a wasteful design, performance-wise. We generally don't expect to be able to run thousands or even hundreds of Spark queries in parallel, as might be required for running population-based BBO - In my previous project, I also observed that driver memory usage quickly reaches limits of what current hardware can muster when one tries to run this many queries in parallel [115]. We will however look into possibilities for running "several" trials in parallel to improve system performance under increasing trial budgets - navigating the fact that we shouldn't completely overload the cluster in the process, of course.

Second, for some tasks within sBBO, *problem dimensionality* is a load parameter with performance implications. For SBO specifically, solving auxiliary problems will generally take up more resources as a function of this parameter, due to more complex interpolation models, and higher cardinalities required by auxiliary optimization algorithms, such as required population sizes for Genetic Algorithms [51]. We will therefore look into whether anything can be done to utilize available computational resources better when solving auxiliary problems under increasing dimensionalities. In some cases, solving them solely on the driver might be wasteful, for instance.

Moving on to the non-functional requirement of *usability*, my old HCI textbook characterized usable systems as being *effective* (solving relevant tasks for the user), *efficient* (doing so with appropriate user effort), *easy to learn*, and *safe to operate* [22, p. 81].

While we already know about the core task to be supported (”sBBO on Spark”), investigating what the other qualities mean within our scope would be likely be worth an entire semester’s work in itself. There are however many other challenges of the more technical sort in this project, and I know from experience that user studies take *a lot* of time to do properly.

To make everything more manageable, we will therefore instead more simply define and work with usability *by proxy*: That is, in our library we will try to adhere closely to the design of a similar tool that we assume to be usable. The choice of proxy fell on MLlib and, by extension, Spark SQL (cf. section 2.1): Similarly to our project idea, MLlib provides advanced analytics functionality on top of Spark, and is targeted towards users with a primary background within data analytics at different experience levels [88]. Using MLlib as a proxy, we can therefore make design decisions based on how things are handled there, and let the ”... who are already familiar with MLlib” be silent when we talk about ”users”.

To spell it out for posterity: We define the library design as being *usable* to the point at which it is analogously similar to MLlib, and when we henceforth say ”user” we are thinking of someone already comfortable with MLlib.

It is for instance relatively easy to verify by proxy, that we shouldn’t force users to debug ”ill-conditioned interpolation sets” when running SBO or configure 50+ hyperparameters for each optimization query with our library, since all components of MLlib run in a plug-and-play fashion, with sensible defaults [167]. By analogy, that design is immediately transferable to our setting.

We will similarly try to provide a library API adhering closely to MLlib’s OOP-based design, which at the very least is *familiar* to users of MLlib. Furthermore, as we shall see, we will find a need to come up with new ideas on how to provide usable sBBO functionality in our library, for e.g. constraint handling, since many existing ideas within sBBO are (in my estimation) designed to work well for doing benchmarks in scientific papers, as opposed to working well in a more ”domesticated” tool for users at varying experience levels, like MLlib.

As for limitations, design by proxy can of course never replace doing real-life user studies etc. fully. Yet it forms an *initial proposal* for a usable design, and leaves some needed breathing room to investigate an idea whose sheer technical ramifications have not been considered before in any other tool, to my knowledge.

3.2 Architectural outline

This section draws a sketch of the library as a whole, including key architectural decisions along with its most prominent features. More implementation specifics will be provided in subsequent sections.

3.2.1 Key decisions

First, while Python is actually the most popular language for interfacing with Spark [88], it was ultimately decided to implement the library in Scala. The possibility of interfacing with Spark internals in their native language was valued, e.g. as leverage for higher efficiency, although how much one could actually benefit from these facilities was ultimately unknown at the onset of the project. At the end of the day, Scala was chosen over Python, since one could in principle always create a Python API to the library, as was similarly done for Spark with PySpark.

Another key architectural decision was to integrate the library with existing advanced analytics functionality of the platform, specifically MLlib (cf. section 2.1), to make PA application development and user onboarding easier. While this does introduce external dependencies, note that MLlib already comes prepackaged as an officially maintained component of the Spark platform, in concrete terms being a folder in the very same repository [88]. The possibilities for code reuse and integration with existing functionality was ultimately deemed worth this dependency.

At its core, the library is basically just an implementation of a new subclass of *Transformer* from MLlib, named *BlackBoxOptimizer*. Confer fig. 3.1. Behind the established MLlib API, BlackBoxOptimizer implements optimization query handling common to all algorithms: Partitioning into subproblems in $transform(\cdot)$ and individual subproblem management in $solve(\cdot)$. The responsibility of its subclasses is solely to provide implementations of individual sBBO algorithms in $minimize(\cdot)$, at the very least solving *unconstrained minimization problems in* $\mathbb{R}^n$, possibly leaving general constraint handling entirely to BlackBoxOptimizer. Apart from familiarity, BlackBoxOptimizer being a Transformer also has the benefit of the library being fully compatible with the Pipeline system of MLlib (cf. section 2.1), essentially advancing its possible applications from predictive to prescriptive analytics (cf. chapter 1). $transformSchema(\cdot)$, as defined in BlackBoxOptimizer, specifies the output schema of calling $transform(\cdot)$, being (informally speaking) the projection of solution and objective columns unto the input Dataset.

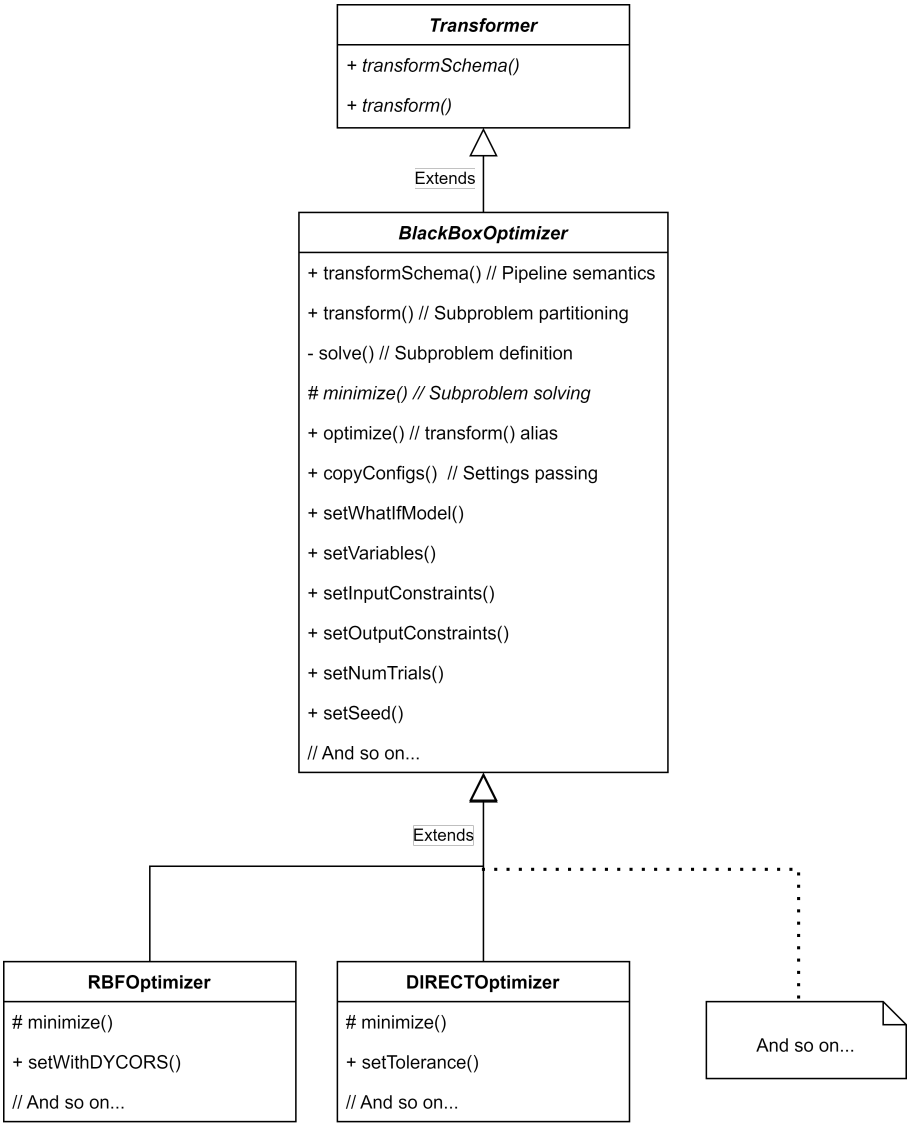

Figure 3.1: The principal class hierarchy of the library, in UML. Method parameter types left out for readability.

Yet another key architectural decision is the one that objective functions are all overtly MLlib Transformers, or converted into such programmatic objects under the hood. That is, as opposed to a higher order function representing objectives with e.g. lambda expressions, BlackBoxOptimizer rather acts like a *higher-order Transformer*. This was done with two possible benefits in mind. The first one is connected to *problem model management*. Objectives being compatible with the MLlib Pipeline system means that they can be (de)composed and (de)serialized through existing MLlib functionality - from a user's viewpoint, managing these

crucial components is thus no different from managing other models in MLlib (cf. section 2.1).

The other benefit is connected to *performance*. The predicament of Python being the most popular client language for Spark means that one should always think twice before implementing features based on *User-Defined Functions* (UDFs), e.g. objective functions as lambda expressions [35]. UDFs come up in several features of the Spark platform, being used for defining e.g. custom scalar transformations in Spark SQL [171]. As a recurrent headache with the PySpark API, when UDFs are implemented with Python functions without further ado, executors have to spawn a new Python interpreter process upon each evaluation. This introduces a significant overhead, when the same Python UDF is called many times in sequence [88]. While recent contributions have sought to remedy this platform-wide problem to a degree [9], "evaluating the same function many times in sequence" unfortunately fits the job description of sBBO algorithms all too well (cf. algorithm 1). Note however, that Transformers like SQLTransformer have a direct translation between a Python and a Scala object [35]. It was therefore decided to represent objectives as Transformers in the library, with shorthands for simple use cases, such that objective evaluations can run entirely in the JVM.

For similar reasons, constraint functions are ultimately obtained from Spark Column expressions [40], which can also be translated between different language APIs.

A final key architectural decision was connected to the problem of where all algorithmic functionality is supposed to come from, i.e., if something existing could be reused or not. High-quality, open-source, JVM implementations of sBBO methods mentioned in section 2.3 are, to the best of my knowledge, *extremely scarce*, and the ones that do exist tend to introduce a large number of dependencies compared to the functionality they would provide by being included into this library (e.g. [162]).

Looking beyond the JVM, some high-quality solvers exist in external languages, written in e.g. C++ or MATLAB [124, 113]. Still, it was ultimately decided not to patch the library together with solvers written in an assortment of different languages, due to the potential *deployment complexity* of such a design.

Instead, it was ultimately decided to implement an initial suite of optimization algorithms from scratch, only using the Scala core language and standard library along with the Breeze linear algebra library [152], with the latter already being an MLlib dependency [173].

Moving the focus of this project towards implementing idiosyncratic sBBO algorithms as opposed to core library functionality was however a grave concern of mine. To have a chance of giving life to the basic algorithmic facilities of the library without jeopardizing the project focus entirely, I ultimately owe a lot of thanks to

Kochenderfer and Wheeler's [91] lucid code snippets, written in the Julia programming language. The aforementioned code is graciously freely available for re-use as long as the original source is acknowledged - consider this an acknowledgement that *all* of my Scala implementations of sBBO algorithms draw directly from their example implementations.

3.2.2 Main features

The following is an overview of library core features, with some of them being motivated by our non-functional requirements of scalability and usability, while others were more or less inspired by alternative sBBO solutions or design issues found during development.

Multi-level Parallelism

As previously mentioned, we are interesting in trying to introduce parallelism when we can, to improve the scalability of our tool, including doing several objective evaluations or *trials* in parallel (cf. section 3.1). The library provides several features for parallelism as per the needs of the application - both the aforementioned *trial parallelism,* as well as *solve parallelism*, i.e. executing several optimization runs in parallel, using e.g. different random seeds for each subproblem. In this sense, parallelism features of the library are marketed as being "multi-level".

Yet trying to introduce the right level of parallelism for sBBO workloads, on top of the parallelism that a Spark cluster already inherently provides, may easily compound complexities and performance issues already connected to running Spark queries: be it driver resource bottlenecks, multi-tenant contention for cluster resources, fluctuating availability of cloud computing resources, task dependency on specific data partitions making CPU cores non-interchangeable performance units, bandwidth limits when data needs to be shuffled around on the cluster, or otherwise [35]. In other words, getting parallelism right becomes a compounding usability issue when the fundamental workload unit is a Spark query. To ameliorate uncertainty about the "right" settings, the library furthermore provides some experimental facilities for *dynamically load balanced* trial parallelism, monitoring throughput of the application on runtime.

Vertical Transfer Learning

An understandable user concern is the one of having to "start over", when e.g. doing several runs on the same optimization problem with different algorithms, which is of course even more aggravating when lost progress concerns expensive computations. Within sBBO research and existing tools, efforts have therefore been made to support different kinds of *transfer learning* [14, 101].

The purpose of transfer learning, as known within sBBO research, is to accelerate solving a problem at hand, by reusing previously obtained knowledge about the same problem (*vertical* transfer), or a similar problem (*horizontal* transfer) [63].

Current horizontal transfer learning methods and benefits are highly dependent on individual problems and solution algorithms, with the usual approach being to incorporate adjacent problem knowledge into an enhanced conditional distribution for Bayesian optimization methods (cf. section 2.3.6) [14].

A more generally applicable approach is preferred in our library setting, which supports a generic kind of vertical transfer as one of its most prominent features. The overall idea is much inspired by the file-based checkpointing feature of Deep Learning frameworks like Keras, which have had to find a solution for similar user frustrations with "starting over", when doing long-running computations [38].

We simply treat trials as *data*, logging everything there is to know about them upon each objective evaluation, and persisting this information to storage in *normalized* form: design points, objective values, constraint function values, time taken to evaluate, etc. This *TrialHistory* in programmatic terms can be reloaded by e.g. an SBO algorithm to continue where it, *or some other algorithm*, left off in a previous run, *without any loss of information about the search space*. Local algorithms can similarly benefit by looking for a good initial incumbent point in past evaluation data. Even relatively rigidly structured algorithms like DIRECT or Nelder-Mead (cf. sections 2.3.3 and 2.3.4) may benefit from basically treating TrialHistory as a lookup cache, omitting evaluations of already known points to quickly recapitulate progress.

Note that this generic way of passing the baton between *several* solution methods and subproblem definitions, as opposed to simply restoring the state of individual algorithms, is what makes vertical transfer more than a pretentious alias for "checkpointing" - it operates more generically than what checkpointing in Deep Learning libraries does, for example [38].

This kind of vertical transfer allows seamlessly chaining different algorithms together to solve the same problem, using e.g. SBO to find the search space region with the global minimum, followed by an efficient local optimizer to refine the solution - such flows are similar to what is called *memetic* algorithms within population-based BBO research [125].

Generic General Constraint Handling

In alternative sBBO solutions, I found no ready-made answer on how to provide efficient, extensible, yet consistent, general constraint handling (cf. eq. (2.4)) in a library like ours, accommodating all methods and constraint types encountered within sBBO (cf. sections 2.3 and 2.4.1). From what I've seen, the usual design in alternative tools is either to lock constraint handling into e.g. one specific penalty

method (e.g. [21]), or to only support general constraint handling on an algorithm-by-algorithm basis (e.g. [101]). The reason behind this might simply be different design priorities. This library however ultimately takes it onset in a broader analytics context (cf. chapter 1), where we cannot ignore the issue of general constraints.

Significant efforts were therefore made to develop a generic approach to general constraint handling for the library, supporting everything from "cookie-cutter" approaches like penalty methods to algorithm-specific ones like filter methods (cf. section 2.4), while not bothering users about how these methods interoperate with Spark behind the scenes.

To provide an alternative to dynamic penalty methods less dependent on users being good at guessing the right penalty magnitudes etc., the library additionally offers a new constraint handling method, leveraging the aforementioned vertical transfer mechanism and some SQL for a more data-driven approach: the *Historical Revisionist Method*.

Search Space Partitioning

Regarding the issue of providing features for parallel processing, it was mentioned that the library also supports doing several optimization runs in parallel. Given that we are trying to solve the same original problem, and we can assume very little about its internal structure, the question is then what constitutes a subproblem for parallelization?

For stochastic algorithms, several optimization runs with different seeds is of course an option, which is a feature that the library provides, due to how simple it is to support it. This kind of partitioning might however only be "worth it" for relatively cheap optimization queries with small input datasets, where we don't worry about possibly arriving at similar solutions with multiple times the computational effort.

An alternative avenue for problem decomposition used with BBO at large is instead based on partitioning the *search space*: either by splitting up individual variable intervals, or by optimizing mutually disjunct subsets of variables more or less independently from one another [125, 107, 194, 191].

Existing techniques for search space decomposition mainly stem from population-based BBO research, and arguably all have disadvantages with respect to our setting: Current methods either require that the piloting user knows the number of interacting problem subcomponents beforehand, or require an unbounded number of extra objective evaluations through custom sampling plans, analyzing problem structure by perturbing variables, or by building special throwaway surrogate models [107, 125]. Neither of these types of solutions are ideal with the respect to our design priorities.

A new dynamic search space partitioning feature is therefore proposed with the library, requiring no special add-on sampling plans or hyperparameter set-

tings, based on a combined dynamic/greedy programming approach and e.g. cooperative game theory (cf section 2.5). Unlike previous attempts at providing such features (e.g. [191]), ours is very simply based on splitting the search space, conceived as a large rectangle, into a number of smaller rectangles, in the hope of making the usefulness of partitioning a bit easier to reason about heuristically. We additionally support doing user-specified search space partitioning, in the event that users have heuristic reasons for splitting certain dimensions.

On top of this, the level of parallelism used for solving subproblems with these partitioning features can be limited as per user needs, making search space partitioning orthogonal to the issue of supporting various forms of parallelism with the library (cf. section 3.1).

3.3 At a first glance

We now move on to an initial illustration of what using the library is like. Consider the following polynomial toy problem:

$$\begin{aligned} \underset{x \in \mathbb{R}}{\text{minimize}} \quad & x^2 + 2x \\ \text{subject to} \quad & -1000 \leq x \leq 1000 \end{aligned} \tag{3.1}$$

The algorithmic wizard of the *WizardOptimizer* subclass of BlackBoxOptimizer, doing automatic algorithm selection, provides the most high-level way of interacting with the optimization facilities of the library and is therefore a natural place to start.

It provides an extension method for performing *how-to queries* on Spark Datasets in an ad-hoc fashion [112], analogous to a vanilla Spark SQL query, which is used for solving the problem in eq. (3.1) below:

```
1 val df = (2 to 2).toDF("a")
2 df.howTo(
3   minimize (expr("x * x + a * x")),
4   subjectTo (-1000.0 <= hcol("x") <= 1000.0),
5   forTrials (100),
6   withOptions ("deadline" -> (2 minutes))
7 ).show()
```

As shown on line 1, each optimization problem assumes an input Dataset, providing context for the optimization problem at hand. In our toy example, we have just placed a coefficient to be used in the objective function there in one column, ”a”.

Ultimately, variable assignments of any feasible solution found by the optimizer, along with their associated objective value, will be projected unto the input Dataset, thus returning a new DataFrame as the output of the optimization run:

```
+---+--------------------+---------+
|  a|                   x|objective|
+---+--------------------+---------+
|  2|-0.9999999982970849|     -1.0|
+---+--------------------+---------+
```

Had the algorithm not been able to find any solution, then columns would simply have been assigned *null* (in the "unknown" sense).

Otherwise, configurations on line 3-6 specify, in order: the objective function, specified as a Column expression here; constraints, where we just declare one bound-constrained variable with an *hcol*, i.e. a *hypothetical column*; the *trial budget* for this run; and finally, miscellaneous options, only including an early termination criterion in this case, based on total processing duration.

We immediately note from the expression on line 3, and the output DataFrame shown before, that variable instantiation in the library is entirely *column-based*: I.e., the objective expression performs computation on the input DataFrame, expecting the value of variable "x" to be a columnar constant "x".

Individual variables are hence easy to access by column identifiers. Note that this design choice in large part stems from the fact that the number of variables to be handled in sBBO is expected to be no more than maybe a few dozens (cf. section 2.3.1) - on the other hand, input datasets might contain millions of rows in our data-intensive setting.

Returning to WizardOptimizer, this is a special subclass of BlackBoxOptimizer, in that its optimization routine encapsulates calls to one or more other optimizers in the library suite, selected based on various heuristics regarding the problem type, cost of evaluating the objective, etc. In our special case, where we notably just have one continuous variable, WizardOptimizer actually encapsulates a call to an optimizer implementing the DIRECT algorithm (cf. section 2.3.4), equivalent to what is shown below:

```
1 val df = (2 to 2).toDF("a")
2 new DIRECTOptimizer()
3   .setWhatIfModel(expr("x * x + a * x"))
4   .setVariables(-1000.0 <= hcol("x") <= 1000.0)
5   .setNumTrials(100)
6   .setDeadline(2 minutes)
7   .optimize(df) // Alias of transform(*)
8   .show()
```

It is now clearer how the library is really just a set of new MLlib Transformer classes at its core. Equivalent configurations to the previous how-to query syntax are given in line 3-6 here, instead instantiating a Transformer subclass explicitly and using setter methods conventional to how other Transformers are configured in MLlib. In BlackBoxOptimizer, $optimize(\cdot)$ is simply declared as an alias of $transform(\cdot)$, the quintessential abstract method of the Transformer class.

For reasons given previously (cf. section 3.2.1), all BlackBoxOptimizers are more or less overtly higher-order Transformers. A more verbose way of achieving the same thing as in the previous example would have been the following:

```
1  val df = (2 to 2).toDF("a")
2  val whatif = new SQLTransformer()
3    .setStatement("""SELECT x * x + a * x AS objective
4                     FROM __THIS__""")
5  new DIRECTOptimizer()
6    .setWhatIfModel(whatif)
7    .setVariables(-1000.0 <= hcol("x") <= 1000.0)
8    .setNumTrials(100)
9    .setDeadline(2 minutes)
10   .optimize(df) // Alias of transform(*)
11   .show()
```

The *what-if model*, e.g. as configured on line 6 in the previous example, is important to note as a key library abstraction. At the very minimum, this Transformer just calculates an objective value such that the library can find it in the first row of the column named ”objective” in the output DataFrame (the expected column name being an overridable default). More generally however, the output DataFrame might contain all sorts of information in other columns of the first row, specifying e.g. constraint function values for Simulation-based constraints (cf. section 2.4), depending on what is most convenient for the user. The chosen moniker thus reflects the general role of this Transformer of calculating outcomes based on hypothetical column assignments, as opposed to just being e.g. ”an objective function wrapper”.

3.4 The basics

Suppose that we have a mixed-integer bound-constrained sBBO problem (cf. eq. (2.3)) that we want to solve with the library, without any kind of parallelism beyond what the Spark platform provides out-of-the-box for individual what-if model evaluations. This section covers how the library handles this relatively basic case, only adding more convoluted issues on top later.

3.4.1 Configuring BlackBoxOptimizers

MLlib Transformers are configured through usage of the *Param* class, and this convention extends to BlackBoxOptimizers [167]. The field governing the expected location of the objective value is for instance declared like this:

```
1  final val objectiveCol = new Param[String](this, "objective", "
       the expected objective column name of the what-if model")
```

The second and third constructor arguments are a lookup key, and a user-directed explanation, respectively. Wrapping what would otherwise have been ”normal” fields into a parameterized Param provides some neat standardized handling across MLlib. Params associated with a Transformer can be operated with through a HashMap-like API [175]. They can be used for specifying default values, like so:

```
setDefault(objectiveCol -> "objective")
```

A setter overriding the *objectiveCol* field can be specified through lookup, like so:

```
def setObjectiveCol(value: String): this.type = set(
    objectiveCol, value)
```

The lookup-based approach of using Params come in especially handy when other optimizers are configured indirectly through WizardOptimizer, as demonstrated in section 3.3, since Params from different Transformers can be merged with properly overridden defaults in one fell swoop, with the $copyValues(\cdot)$ method. All the wizard had to do to configure and run the DIRECT optimizer properly, including the deadline etc., was the following:

```
copyValues(new DIRECTOptimizer()).optimize(df)
```

In its current form, the BlackBoxOptimizer class contains about 30 Params, covering common library functionality, with subclasses adding custom algorithmic configurations, as needed. An effort was made to design this possibly overwhelming number of options with sensible defaults etc., such that they can generally be ignored by users who are only interested in the very basics. Only a what-if model, variable declarations, and the trial budget is mandatory information for running. We will introduce several more Params as they become relevant to our discussion.

The declaration of variables and their handling is especially important. Out of the box, the library supports three types of variables: *Real*, *Integral* and *Categorical*, of which the first two are numeric and the third denotes a sets of String labels. This array of options was chosen to reflect what other sBBO tools generally provide (e.g. [21, 127]). A combination of Scala method overloading and extension methods on numeric types provides the following syntax for declaring variables:

```
df.howTo(
  minimize (foo),
  subjectTo (-1000.0 <= hcol("x1") <= 1000.0, // Real
             -1000 <= hcol("x2") <= 1000, // Integral
             hcol("x3") in Seq("a", "b", "c")), // Categorical
  forTrials (bar)
).show()
```

I tried to make this syntax fit in with the rest of Spark SQL, while making it visually distinct and avoiding needless verbosity. Normally one refers to columns in Spark SQL with $col(\cdot)$, and as such, $hcol(\cdot)$'s are just "special" *hypothetical* columns, instantiated right before what-if evaluation. Reusing standard SQL predicate syntax, e.g. *x1 BETWEEN -1000.0 AND 1000.0*, was ultimately decided against, for better writability.

Note that it is *mandatory* to declare variables together with complete bound information. Not only is this a friendly nudge for users, to reconsider whether their domain of interest is truly e.g. the entire range of the Double datatype.

It also enables uniform handling of mixed-integer bound-constrained problems across the entire library (cf. eq. (2.3)). Ultimately, bound information etc. is aggregated on subclasses of the interface named *Variable*. By always having bound information available here, it is possible for algorithms to simply do optimization in $\mathbb{R}^n$ within the corresponding numeric bounds of the different variable types, while the library handles Integer and String conversions behind the scenes, whenever a trial is called for. For a Categorical variable containing three categories, optimization algorithms can simply send Double values within the $[0, 2]$ interval for evaluation, for instance. Methods for such conversions are provided on individual Variables, which are furthermore aggregated on a *SearchSpace* class instance.

Furthermore, it allows a universal policy of handling mixed-integer bound constraints as *Unrelaxable* constraints (cf. section 2.4.1): If a point proposed by the algorithm cannot be converted to a value within the domain of interest by library code, e.g. it deals in negative swimming pool geometry, evaluation simply returns an infinite objective value without evaluating the what-if model. Note that dealing with these Unrelaxable constraints is generally easy, by e.g. scaling the search space down to the unit box.

3.4.2 The optimization flow

Whenever $optimize(\cdot)$, or equivalently $transform(\cdot)$, is called by the user or the optimization wizard, the overall optimization flow runs through three principal methods: $transform(\cdot)$, $solve(\cdot)$, and $minimize(\cdot)$ to be discussed here in turn. The first two, implemented in BlackBoxOptimizer provide universal query handling, while the final one is implemented by algorithmic subclasses, being implementations of e.g. DIRECT or some other specific algorithm (cf. fig. 3.1).

transform

The responsibility of $transform(\cdot)$ is to provide universal routing required for the input optimization problem, with subproblem decomposition and multi-threaded solving taking up the larger part of functionality. We will however ignore these features for now. Without any parallelism involved in our limited case, $transform(\cdot)$

simply parses a few Params, obtaining e.g. the SearchSpace object from the provided variables, and delegates all other problem solving to $solve(\cdot)$. By default, $transform(\cdot)$ also notably *caches* and *uncaches* the input Dataset at the default storage level, before and after calling $solve(\cdot)$, respectively. The reason for this is that the input Dataset will possibly be accessed many times during optimization, making caching a likely performance boost.

Finally, $transform(\cdot)$ obtains a number of solution proposals from each subproblem handled by calls to $solve(\cdot)$, only one in our case. The optimization run is thus concluded by $transform(\cdot)$ projecting the best feasible solution and objective columns among all subproblems unto the input Dataset, if applicable - otherwise it assigns null values to solution and objective columns, as mentioned previously (cf. section 3.3).

This way, regardless of whether a feasible solution was found or not, the output schema of the returned DataFrame is always compliant with the one specified by BlackBoxOptimizer in its *transformSchema*, the other mandatory method to be implemented by Transformer subclasses, needed for compatibility with the MLlib pipeline system (cf. section 2.1) [168].

solve

$solve(\cdot)$ is given a series of inputs specifying a singular optimization problem to be solved, including the search space, input DataFrame, information about the objective, total trial budget for this call, etc.

Its main task is to handle what is arguably the single most important abstraction used by library internals: a *functional closure* encapsulating all necessary handling associated with evaluating the what-if model, known only as the $objective(\cdot)$ function by optimization algorithms, and therefore known to us as the *objective closure* for short.

From a library designer's viewpoint, the objective closure basically inverts control back to us from library clients, allowing "arbitrary code execution" upon each trial, regardless of what any particular minimizer is doing. The objective closure leverages the fact that putting a black box inside a black box is a *closed operation*, i.e. it yields yet another black box: That is, we can augment the objective function exactly as we like without interfering with the solution strategy of any algorithm, providing consistent handling across algorithms in the process.

As an additional leverage point for sBBO in particular, we may reasonably assume that the objective closure will be treated as an *expensive function* by library algorithms, e.g. not to be called asynchronously in an infinite loop, given the core premise of the library. Doing relatively slow I/O operations and the like within the objective closure is therefore sometimes a worthwhile possibility.

In our case, $solve(\cdot)$ simply needs to declare the objective closure from problem information obtained on runtime, call $minimize(\cdot)$ with it, and relay the best

solution found by $minimize(\cdot)$ to $transform(\cdot)$.

At its core, the objective closure does two things: It hides all details of interfacing with Spark from $minimize(\cdot)$, and has the final say about access to what-if model evaluations according to user-provided termination conditions, e.g. the maximum trial budget. It is also the final arbiter as to which solution is returned from the optimization run, logging the best feasible solution found so far - this way, algorithms don't need to worry about feasibility at all when looking for new incumbents.

The objective closure handed over to $minimize(\cdot)$ simply takes an Array of Doubles as input and returns a instance of the *Trial* class, including information about the objective value, among other things. When it is time to terminate, the objective closure will always just return *None* when called, i.e. a null-like value.

Underneath this veneer, the objective closure utilizes the SearchSpace class to convert the input Array to values within the domain of interest, e.g. Strings for Categorical variables. If the caller hasn't done this already, Categorical and Integral variables are obtained from Double values by *rounding* - sometimes a controversial approach, but nonetheless one that has proven useful in many practical applications [51, 120]. As mentioned before, bound constraints are *Unrelaxable* library-wide, and we therefore return infinite objective values in the event of any violation, without accessing the what-if model. In the usual case however, we obtain a *Map* from column names to literal values, ready for projection unto a Spark Dataset. The heart of what-if model evaluations looks like this:

```
val row = whatIf.transform(df.withColumns(cols)).head
```

That is, we call the $transform(\cdot)$ method of the what-if Transformer, with the variable assignments of the *cols* Map projected unto the input Dataset *df*, and retrieve the first *row*. In our simple case, we just need to get the objective value from the specified *objCol* column, like so:

```
val objValue = sign * row.getAs[java.lang.Number](objCol)
  .doubleValue()
```

The *java.lang.Number* roundabout is a way of ensuring that users can return all sorts of numeric types as objective values without pedantic conversion errors, as long as they ultimately make sense as Doubles on the JVM platform.

As for the meaning of *sign*, recall that inverting the sign of an objective function converts maximization problems into solution-equivalent minimization problems (cf. section 2.2). While the default of the library is to do minimization, and this is the functionality offered by implementations of $minimize(\cdot)$, maximization is supported with a Param toggle and handled by just using -1 as the value of *sign* in the objective closure.

Termination conditions, including the trial budget being spent, is also accounted for through the objective closure. The accountant of the objective closure is an in-

stance of the *BlackBoxBudgeteer* class, which at its core is just an interface to an *AtomicInteger*, that is, a thread-safe counter [39]. This is used for keeping track of the trial budget. Access to objective evaluations is controlled by the initial part of the inline objective closure declaration:

```
def objective(sol: Array[Double]): Option[Trial] =
  bbb.budgetEnsuredCount().map{ trialNum =>
  // what-if model evaluation here.
  }
```

That is, before we enter the body of the objective closure, we first ask the BlackBoxBudgeteer *bbb* for a valid Trial ID with $budgetEnsuredCount(\cdot)$. If it for whatever reason is time to terminate, the aforementioned method evaluates to None, which in Scala entails that the entire $map(\cdot)$ call will also evaluate to None.

Apart from the trial budget, we also support a few other termination conditions in the objective closure. We basically just check these upon each trial, inside the objective closure. For instance, the deadline option seen previously (cf. section 3.3) is supported by simply starting a timer at the onset of the query, and then doing the following check inside the objective closure:

```
if (deadline.isOverdue()) {
  bbb.terminateEarly()
}
```

That is, if the user-specified deadline is overdue, we instruct BlackBoxBudgeteer to return None from now on, whenever access to trials is requested, which is effectively a signal for $minimize(\cdot)$ to terminate. Note that supporting user-specified deadlines *exactly* is difficult, for a number of reasons - we want to terminate processing in a well-defined state and not just kill threads in the middle of e.g. doing I/O to backup progress. We therefore make clear in the user-directed info String of the deadline Param that deadlines are just checked "regularly", and allow remaining trials to terminate before stopping the optimization run.

Custom termination conditions, including the shown deadline option along with options specifying a desired (minimum/maximum) objective value for a run are all supported through Param options on BlackBoxOptimizer, and are thus common to all algorithmic subclasses.

Having built the objective closure from all relevant runtime information, $solve(\cdot)$ passes the baton to $minimize(\cdot)$.

minimize

BlackBoxOptimizer subclasses implement specific sBBO algorithms in $minimize(\cdot)$ (cf. fig. 3.1), solving minimization problems with possibly *any* type of method mentioned in section 2.3.

Allowing for this level of generality and flexibility imposed some constraints on how to design the API for algorithmic implementations. Some existing solutions for sBBO impose very opinionated structures in this regard, requiring implementation of specific methods representing e.g. $propose(\cdot)$ (cf. algorithm 1) or other generic building blocks identified for sBBO algorithms [21, 63]. Such designs arguably have the benefit at offering more fine-grained framework-side control of the optimization process than what this library provides.

For the purpose of this library, which has a relatively broad scope compared to others, a problem with such design patterns is however that they tend to map more or less well to different sBBO approaches (cf. section 2.3). For instance, imposing an API where each algorithm performs one $iteration(\cdot)$, called repeatedly to obtain a new proposal for evaluation, would work fine for SBO, but is at best a clunky design for directional and simplicial algorithms (cf. section 2.3.3), which need to evaluate a *set* of points within one iteration to work properly - algorithms that are quite stateful and rigidly structured, like DIRECT for instance (cf. section 2.3.4), are even more hopeless to fit into any such mold.

At the end of the day, it was therefore decided to design the API for algorithmic implementation to impose as few restrictions as possible, only imposing structure *indirectly* when the objective closure is called from $minimize(\cdot)$.

Note that this design does require a level of *cooperation* from algorithmic implementers: Once the objective closure begins to return None, signalling that it's time to terminate, implementations might read the room and exit the main loop for termination - or not. For all we know, algorithms might not even do optimization at all, or even worse, population-based BBO. However, it was ultimately decided to work under the following assumption:

Design Assumption 3.1 *Library clients and developers are consenting adults.*

This is a pervasive way of thinking in e.g. the Python ecosystem (and an officially endorsed analogy not invented by me) [69]. That is, we don't engage in *defensive design*, imposing arbitrary restrictions on well-meaning developers, for fear of getting "hacked" by e.g. infinite while loops in implementations of $minimize(\cdot)$. In fact, algorithmic implementers should feel free to "hack" library facilities as much as they like, with the invisible hand guiding the most useful solutions to prevail in the end.

Back to the implementation, $minimize(\cdot)$ is declared as an abstract method in BlackBoxOptimizer with the following signature:

```
protected def minimize(eval: Evaluator,
                       ss: SearchSpace,
                       initialSolution: Option[Array[Double]],
                       history: Option[TrialHistory],
                       seed: Long): Option[Trial]
```

The *Evaluator* encapsulates calls to the objective function, and ultimately parallelizations thereof, to be discussed later. The SearchSpace encompasses all information about variables of the problem, including e.g. their number, bounds, types, along with some commonly useful utilities for e.g. converting back and forth between unit-scale and at-scale design points. To not waste any trials on out-of-bounds proposals, a minimalistic, but useful approach for algorithms to follow is to do bound-constrained minimization in $\mathbb{R}^n$ with this information, leaving rounding for discrete variable types and other constraint handling to the objective closure.

The *initialSolution* is generally speaking a user-specified "point of interest": a possible initial incumbent for Local algorithms, a custom addition to the interpolation set for Model-based algorithms, etc. Its impact varies by algorithm. To conserve resources, algorithms obtaining little to no value from evaluating this point can therefore potentially choose to ignore it (but they generally don't in the standard suite). Initial solutions are enabled by BlackBoxOptimizer Params specifying either a *Map* of variable assignments or a Boolean making the library look for it in the input Dataset inside $transform(\cdot)$.

The role of *TrialHistory* pertains to the vertical transfer mechanism of the library, to be discussed momentarily. The *seed* is the assigned seed for random number generation for this call to $minimize(\cdot)$, enabling a feature for running the same algorithm with different seeds in parallel, while enabling reproducible results for individual runs. sBBO algorithms are commonly stochastic, with DIRECT being the only exception among algorithms of the implemented suite.

At the generic level, implementations of $minimize(\cdot)$ may use this information as they please to perform sBBO, using None values returned from trials as a signal to terminate processing and return the best Trial found, if any, as the proposed solution, which propagates back to $transform(\cdot)$ for possible projection unto the input Dataset.

3.4.3 TrialHistory and vertical transfer

The library mechanism for vertical transfer works as an additional layer on top of the aforementioned flow, and its significance warrants separate discussion.

As described previously (cf. section 3.2.2), the basic idea is to retain information about all objective evaluations executed so far, so that future optimization runs can build on top of this instead of starting over. Similar to how checkpointing is done in e.g. Deep Learning frameworks like Keras, vertical transfer is enabled by configuring the wizard with a checkpoint directory Param:

```
1 df.howTo(
2   minimize (foo),
3   subjectTo (bar),
4   forTrials (baz),
```

```
5   withOptions ("checkpoint path" -> "my/favourite/path")
6 ).show()
```

Or equivalently by using *setCheckpointPath*$(\cdot)$ on any BlackBoxOptimizer. In doing so, library code will now intermittently persist Trial records to this directory, and algorithms may use any existing records to accelerate progress.

Upon each objective evaluation, the role of the objective closure declared in *solve*$(\cdot)$ is to save new Trial information in an in-memory buffer, which is written to the checkpoint path at configurable intervals, or when the *minimize*$(\cdot)$ run is over. The idea looks something like this in the objective closure (omitting messy synchronization handling):

```
1 historyPath.foreach{ path =>
2   trialBuffer += trial // Buffer new Trial.
3   if (trialBuffer.length >= $(checkpointInterval)) {
4     // Persist to storage, and flush:
5     trialBuffer.toSeq.toDS()
6       .write.mode(SaveMode.Append).save(path)
7     trialBuffer.clear()
8   }
9 }
```

As indicated by line 5-6, we reuse Spark's API for handling I/O for convenience, converting records to a Dataset before appending them to the checkpoint file, which is a Parquet file by default.

From the point of view of *minimize*$(\cdot)$ implementations, the provided TrialHistory instance is simply an interface, or "view" in SQL terms, to a series of Trial records, a case class declared with the following fields:

```
1 case class Trial(solution: Array[Double],
2                  objective: Double,
3                  /* Constraint handling stuff */
4                  duration: Option[Long],
5                  evaluation: Int) {
6   /* Some instance methods */
7 }
```

The first two fields contain information about design points and their known objective values, likely to be used by all algorithms interfacing with TrialHistory. Note for *solution*, that we can always convert between Double Arrays and Variable subtypes in both directions, using the SearchSpace interface. Furthermore, objective values always reflect the minimization direction, and are thus also in a sense "normalized" with respect to the original problem representation. The final two fields are mainly of interest to library internals, saving the total *duration* taken to evaluate this Trial point in milliseconds, along with a unique *evaluation* ID among all Trial records, as accounted for by the BlackBoxBudgeteer.

TrialHistory is implemented as another case class, taking a DataFrame loaded from the checkpoint path as its input, exposed as a Spark Dataset of Trial records:

```
case class TrialHistory(hisDF: DataFrame,
                        scope: Option[SearchSpace] = None) {

  val trials: Dataset[Trial] = hisDF.select(trialFields:_*)
    .as[Trial]
  /* ... */
}
```

While the *trials* field supports more advanced use cases in $minimize(\cdot)$, basically allowing ad-hoc Spark SQL queries on the Trial Dataset, the intent is rather that algorithms should not have to worry about Spark at all, as is also the intent with objective evaluations. Therefore, TrialHistory exposes a few methods meant to handle what I would consider to be "low-hanging fruit" to most sBBO algorithms: Accessors to all (distinct) Trials, an accessor to the Trial with the best objective value, and so on.

Duplicate Trials are a possible casuality of e.g. running the same optimizer in parallel with different seeds, and are best avoided in e.g. interpolation sets for Model-based sBBO (cf. section 2.3). To preclude such worries, TrialHistory therefore exposes this method, removing all duplicate points from the returned collection:

```
def distinctTrials: Array[Trial] = trials
    .dropDuplicates("solution").collect()
```

Model-based algorithms can thus possibly obtain an initial interpolation set. With an increasing set of records across different calls to $minimize(\cdot)$, such shorthands can be used in algorithm-specific ways to build further upon what was learned in previous calls, regardless of which algorithm acted as the learner previously. We will show a few examples of how this works in practice shortly.

3.4.4 The algorithmic suite

More information about the algorithmic library suite, providing implementations of $minimize(\cdot)$ are in order, to illustrate how the design presented so far works in practice, still in our limited case with only bound-constrained mixed-integer problems (cf. eq. (2.3)) and no parallelism whatsoever.

The implemented algorithmic suite covers a total of 7 core algorithms, or 10 if one counts algorithmic variants separately. As per our emphasis on global optimization (cf. eq. (2.4)), 5 of 7 core algorithms are Global methods, while the remaining two represent directional and simplicial Local methods, respectively.

Class name	Algorithm	Scope	Strategy	Variants
BayesianOptimizer	GP-based SBO	Global	Model-based	EI or LCB
CMAESOptimizer	CMA-ES	Global	Direct	N/A
DIRECTOptimizer	DIRECT	Global	Direct	N/A
LatinHypercubeOptimizer	LHS	Global	Direct	Standard or symmetric
MADSOptimizer	MADS	Local	Direct	N/A
NelderMeadOptimizer	Nelder-Mead	Local	Direct	N/A
RBFOptimizer	RBF-based SBO	Global	Model-based	SRBF or DYCORS

Table 3.1: Algorithms implemented in the standard library suite.

Confer table 3.1. Most algorithms are familiar to us from chapter 2. *Covariance matrix adaptation evolution strategy (CMA-ES)* is a popular Global/Direct algorithm that is much more sample-efficient than its population-based cousins [91]. *Mesh-Adaptive Direct Search (MADS)* may be regarded as a state-of-the art method within directional sBBO [42, 10]. BayesianOptimizer supports GP-based SBO with either Expected Improvement (cf. eq. (2.29) or Lower Confidence Bound (cf. eq. (2.28)) auxiliary objectives. RBFOptimizer supports RBF-based SBO based on the weighted distance merit function (cf. eq. (2.27), with the Dynamic COordinate Search (DYCORS) variant being an extension fit for higher-dimensional problems [148]. We support various kinds of LHS as well, with LHS being regarded as a "dummy" Global/Direct optimization algorithm in its own right, as by other authors [11].

On top of this, we have the WizardOptimizer, basically routing to different combinations of these algorithms depending on various heuristics.

Due to my apprehensions about jeopardizing the focus of the project by spending a lot of time on implementing sBBO algorithms from scratch, I ultimately decided not to implement any trust region methods, representing the Local/Model-based approach (cf. section 2.3.5): I found by vicarious observation that these methods would be the most complex to implement by a large margin, and that they were simply not that popular in other solutions, with MADS and Nelder-Mead instead dominating the Local niche. Still, a trust region approach is "kind of" represented by the implemented stochastic RBF variants (cf. section 2.3.6).

We will now go over a few representative examples to show how these algorithms were implemented with the *minimize*$(\cdot)$ API.

MADS and Local search

MADS is seemingly very influential within sBBO research, even within Global sBBO benchmarking [13, 148]. Its name stems from the fact that its pattern points for each iteration are generated stochastically from a grid-like structure, a *mesh*, made more or less fine-grained across iterations with the step size parameter, as per the generic directional framework (cf. section 2.3.3) [11]. In the library imple-

mentation, MADS handles most transactions with the library API in the first few lines of code:

```
1 val rand = new Random(seed)
2 val n = ss.nVars
3 val objective = eval.objective(_: Array[Double])
4
5 // Get initial solution:
6 val init = initialSolution.flatMap { sol =>
7   val fromHis = history.flatMap(_.point(sol))
8   if (fromHis.isDefined) fromHis else objective(sol)
9 }
10 val his = history.flatMap(_.bestTrial)
11 var best: Trial = (init, his) match {
12   // Best from init, his, or a random one generated with ss.
13 }
```

That is, we obtain the random seed and basic problem information about dimensionality and the objective in lines 1-3, from the inputs given to *minimize*$(\cdot)$. On line 6-13 we then look for the best possible initial incumbent point, as required by Local algorithms. We first consider any user-specified point by looking up a "cached" version in TrialHistory with *point*$(\cdot)$ (line 7), only evaluating the objective on cache misses (line 8). Then we look for the best candidate in the input TrialHistory, if it exists, on line 10. Finally, we pick the best found candidate, or default to a random bounds-feasible point if nothing better is available, generated with the input SearchSpace (line 11-13).

Together with a custom termination condition regarding a minimum step size, MADS keeps an eye on termination conditions by asking the Evaluator if it's time to stop before entering the main loop body again:

```
1 while (!eval.terminate && stepSize > tol) { /* ... */ }
```

Note that !*eval.terminate*, conferring with a BlackBoxBudgeteer under the hood, is designed such that a return value of *true* guarantees that there is at least one more evaluation left. By thus always "looking before jumping", objective evaluations can take place with very little notational overhead, simply unpacking the returned Trial object from the Option monad on line 2 below:

```
1 val newSol = project(best.solution, stepSize, d) // Line search
2 val Some(newTrial) = objective(newSol) // Objective evaluation
```

The *newSol* Double Array will be converted to whatever variable domains these values correspond to by the objective closure - MADS only works with vectors in $\mathbf{R}^n$, leaving explicit constraint handling to other parts of the library.

By playing along the gentleman's agreement of the API (cf. design Assumption 3.1) and leaving explicit constraint handling to the objective closure, MADS

can thus do its work mostly undisturbed and return the best found Trial whenever optimization is at an end.

SBO and Global search

In our presented taxonomy, SBO forms the polar opposite to Local/Direct methods like MADS, and is therefore a natural place to complete our tour of examples (cf. section 2.3). Using these strategies requires an initial set of sample evaluation, forming a DoE. For this, the library provides the LatinHypercubeOptimizer, a subclass of BlackBoxOptimizer doing DoEs with different LHS variants.

Note that LHS (with "S" now denoting "Search") can be regarded as an optimization algorithm, in that it is natural to return the best design point in the sample as a proposed optimum [11] - it is therefore technically a Global/Direct algorithm in our taxonomy (cf. section 2.3). Of course, due to just being a simple sketch of the search space, LHS only really shines when combined with other algorithms: Providing initial incumbents for Local methods (like MADS), or interpolation sets with good coverage for SBO. For these reasons, it was found best to *decouple* LHS from other methods, making it a standalone optimizer in programmatic terms.

The prescribed way of passing the baton from LHS to some other optimizer is by the library's standard mechanism for vertical transfer, i.e. using a checkpoint path:

```
1  // Configure LHS:
2  val lhs = new LatinHypercubeOptimizer()
3    .setWhatIfModel(whatIf)
4    .setVariables(vars)
5    .setNumTrials(doeTrials)
6    .setCheckpointPath(path)
7
8  // Run LHS:
9  lhs.optimize(df).show()
10
11 // Pass on problem information to optimizer:
12 val opt = lhs.copyConfigs(new BayesianOptimizer())
13   .setNumTrials(optTrials)
14
15 // Run SBO:
16 opt.optimize(df).show()
```

On lines 1-9, we run an initial DoE with a set of problem configurations, including a certain Trial budget. On lines 12-13, we use the $copyConfigs(\cdot)$ shorthand (cf. fig. 3.1) to pass on all common BlackBoxOptimizer Params (the what-if model etc.) to a GP-based SBO algorithm, overriding with a new trial budget for optimization on line 13, before running SBO on line 16.

Due to BlackBoxOptimizers being MLlib Transformers, we can of course encapsulate the above flow in a PipelineModel (cf. section 2.1), if such a shorthand is desired:

```
val sbo = new Pipeline().setStages(Array(lhs, opt)).fit(df)
sbo.optimize(df).show()
```

In our case, the call to $fit(\cdot)$ just checks if input and output schemas are compatible among steps in the pipeline, which they are here, assuming I didn't bungle the definition of $transformSchema(\cdot)$ in BlackBoxOptimizer. PipelineModels being Transformers, we can add on even more components later, e.g. a problem-configured MADSOptimizer named *mads*, to refine the solution obtained from SBO, forming a custom hybrid algorithm:

```
val hybrid = new Pipeline().setStages(Array(sbo, mads)).fit(df)
hybrid.optimize(df).show()
```

There is nothing much new to say about how $minimize(\cdot)$ is implemented in LatinHypercubeOptimizer: Based on its allotted trial budget, it simply generates its sampling plan within the unit hyperbox, and then uses input SearchSpace utilities to rescale the obtained unit Double Arrays, before passing them to the objective closure, which handles the rest.

To make the obtained sample suitable for interpolation and avoid bad LHS edge cases, we borrow a trick I found elsewhere, and generate a number of different possible sampling plans randomly, until the proposed matrix of sample rows has *full column rank*, thus ensuring non-degenerateness with respect to the equational systems we expect to solve (cf. section 2.3.6) [141, 120].

By a Param option, LatinHypercubeOptimizer additionally supports doing either a basic "classic" LHS, or (by default) a more advanced "symmetric" LHS, which ensures better coverage by enforcing symmetry among sample points across all search space axes [200].

Adaptive sampling strategies utilizing LHS results are implemented in their own BlackBoxOptimizers. There are two classes for these, RBFOptimizer and BayesianOptimizer, corresponding to RBF and GP surrogate approaches, respectively - we use the more popular term for GP-based SBO to name the class (cf. section 2.3.6). Params are used to specify which adaptive sampling strategy, auxiliary problem solver, etc., to use with respect to the chosen surrogate.

Due to their conceptual overlap, the aforementioned subclasses implement $minimize(\cdot)$ in a similar way. They obtain their initial interpolation set from the input TrialHistory (assumed present in SBO), which is used for constructing the initial surrogate model:

```
// Get interpolation set from input history:
val Some(prevTrials) = history.map(_.distinctTrials)
```

```
3
4  /* ... */
5
6  // Construct initial surrogate:
7  val X = DenseMatrix(prevTrials.map(_.solution): _*)
8  val fX = DenseMatrix(prevTrials.map(_.objective): _*)
9  val surrogate = new RadialBasis(ss) // Or GaussianProcess
10 surrogate.addPoints(X, fX)
```

Note that the above snippet is slightly simplified, for current explanatory purposes. We construct Breeze matrices for sample points and their corresponding objective values (lines 7-8), and use these to construct the initial surrogate on lines 9-10.

After this, the implemented SBO algorithms simply progress within their main loop, until the BlackBoxBudgeteer relays a gentlemanly termination signal through the Evaluator. A simplified version of the main loop of BayesianOptimizer looks like this:

```
1  while (!eval.terminate && convergenceCheck) {
2
3    // Get k candidate points from adaptive sampling strategy:
4    val proposals = bo.proposePoints(k)
5
6    // Evaluate all candidates:
7    val evals = eval.objective(proposals).flatten
8
9    // Possibly update best incumbent solution:
10   val bestTrial = evals.minBy(_.objective)
11   if (bestTrial.objective < best.objective) {
12     best = bestTrial
13   }
14
15   // Update GP surrogate with new Trials:
16   val (pts, objs) = /* Map evals to Breeze matrices */
17   gp.addPoints(pts, objs)
18 }
```

As in other algorithms, we ask Evaluator if there are any evaluations left before iterating further (line 1). On line 7, we then evaluate a number of candidate points obtained from adaptive sampling and auxiliary solving (cf. section 2.3.6), using an Evaluator shorthand for evaluating several points in batches (just in sequence for now). Otherwise, we just update our solution and surrogate information on lines 9-13 and lines 15-17, respectively.

As in the LHS class, adaptive sampling strategies ensure through unit scaling that new proposals are all bounds-feasible, and we can therefore just feed rescaled Double Arrays to the objective closure without doing any explicit constraint han-

dling on our own, of note, without having to worry about numerically unstable objective values for violating bound constraints.

I am tempted to present a lot of developer notes about how the different classes encapsulating adaptive sampling strategies and auxiliary solving are implemented (*bo* in the previous snippet), as this turned out to be a quite involved programming task for me. We will however try to stay focused on the big picture here, with respect to what the library can do within its design space.

One general issue of import with respect to scalability (cf. section 3.1) was how to handle *auxiliary problem solving*, an important part of Model-based methods with many degrees of freedom (cf. section 2.3.6). Namely: how do we best leverage computational power for this in a Spark application, noting our aspiration to support parallelism when it makes sense? I found that the answer to a degree depends on the problem at hand, along with individual algorithmic niches.

RBF auxiliary problems are always solved entirely *locally*, i.e. on the driver machine with no Spark queries involved. We just generate a local set of sample candidates, perturbed around the incumbent solution, and pass them for ranking by the distance-weighted merit function (cf. section 2.3.6). The reason for this choice is in part that the "'officially recommended" number of candidates for SRBF is only a few hundred, depending on the number of decision variables, making a full-fledged Spark query at this scale seem like "overkill" [146, 141].

A relevant alternative to consider might have been to e.g. execute a Spark query using e.g. UDFs to perturb a large number of candidates on the cluster, and then retrieve the best ones at the end by aggregation, possibly getting better auxiliary solutions. The question is however whether solving auxiliary problems this way on each algorithmic iteration is actually "worth it" in the general case, if such a case even exists? That is, we might risk that solving auxiliary problems on the cluster, doing a lot of I/O etc., takes similar or more computational resources vs. evaluating the objective, or hogs resources from objective evaluations in other threads - all for an approximate solution to an approximate problem, looking at the bottom line.

Handing off non-trivial auxiliary subproblems to remote worker nodes to deload the driver node might also end up backfiring - the driver still needs to manage task completion etc. over the entire course of the likely correspondingly non-trivial Spark query [35]. On top of this, real-world applications of sBBO have been doing just fine without solving auxiliaries with cluster computing [11].

At the end of the day, and acknowledging that *there really is no universal answer*, I estimated that for many problems, doing Spark queries for auxiliary problem solving is simply *not worth it*.

I still decided to tinker a bit with the concept in BayesianOptimizer, however. GP-based SBO, being a very deliberate method (cf. section 2.3.6), shines in applications where the objective is massively expensive to evaluate, taking e.g. *several*

days to run high-fidelity simulations [57, 142]. In such cases, high-effort auxiliary problem solving might be worth considering. To solidify this niche of BayesianOptimizer, I decided to try out an approach with distributed auxiliary solving with it. The general idea is to capture the chosen auxiliary objective in a (you guessed it) functional closure, then execute a large number of differently parameterized optimization runs on the cluster in parallel, and finally return the best solution(s) found for evaluation with the real objective function. The number of auxiliary optimizers and the number of solutions desired for evaluation are Param-configurable hyperparameters in this approach, as per the needs of the application, yet defaults to $2 \cdot n$, based on a similar heuristic found elsewhere [141].

For an auxiliary optimizer, I decided to implement Adaptive Simulated Annealing from Kochenderfer and Wheeler's book [91] - it is a variant of the Global/Direct algorithm mentioned previously, able to adjust important hyperparameters without user intervention (cf. section 2.3.4). To enforce search space coverage, we use a symmetric LHS to provide an initial incumbent for each auxiliary subproblem, and different random seeds for each run. In code, the auxiliary optimization query ultimately looks like this:

```
1  val optimizer = udf(adaptiveSA(_: Array[Double], _: Double)
2                      (auxObjective, searchSpace))
3  val sol = spark.sparkContext
4    .parallelize(initSols).toDF("initial_solution")
5    .withColumn("seed", rand(rng.nextLong()))
6    .withColumn("res", optimizer($"initial_solution", $"seed"))
7    .select($"res._1".as("solution"), $"res._2".as("objective"))
8    .dropDuplicates("solution")
9    .orderBy(asc("objective"))
10   .limit(k)
11   .getSeq(0)
```

So, on line 1 we define the auxiliary optimizer with the necessary problem information in an initial curried parameter list, the algorithm now only expecting an initial solution array and a seed to run. On line 3-4 we create a DataFrame with the LHS initial solution Arrays in *initSols*. Lines 5-6, we add a random Double to each row as the seed, and parameterize each subproblem accordingly. Lines 7-9, we dissect the result tuple of the UDF, remove duplicate solutions, and rank them by objective value, having of course inverted the sign when using Expected Improvement in the closure. Lines 9-10 we return the up-to-k best solutions found, retrieving the solution arrays from the first row field (zero).

Is it worth it or necessary to solve auxiliary problems this way in a given application? It depends on a number of factors, including available cluster resources over time, how expensive it is to evaluate the actual objective closure and the intricacy of the underlying optimization problem.

3.4.5 The optimization wizard

We conclude our discussion of the basics with a few more details on how the optimization wizard, including the how-to syntax, works. Unlike other subclasses of BlackBoxOptimizer, WizardOptimizer overrides BlackBoxOptimizer's $transform(\cdot)$ method (cf. fig. 3.1), thus routing to $optimize(\cdots)$ calls of different BlackBoxOptimizers instead of $solve(\cdot)$ calls of different subproblems.

The sorcerous moniker is meant to convey that this algorithm, like an *installation wizard*, essentially guides the user through a *predetermined* array of options. It helps the user select among algorithms of the standard library suite with a fixed control flow, and is *not* implemented with extensibility in mind.

In an alternate approach, one might have provided hooks for adding user-defined solution strategies or selection criteria on the WizardOptimizer class. These could then be used for assigning priorities or costs with respect to input problems for $transform(\cdot)$, and going with the best strategy in the end. However, I ultimately decided to prioritize other development challenges than making an extensible *meta-optimizer*, based on the *YAGNI principle* - that is, at a stage of development where we verily only have the algorithms in this library suite available, "You Ain't Gonna Need It" [84].

The resultant niche of WizardOptimizer is to provide a quick-and-dirty solution for people not interested in the small stuff. The list of factors considered in $transform(\cdot)$ includes:

1. What is the number of variables?
2. Are all variables real-valued?
3. Is the problem generally constrained?
4. What is the total evaluation budget?
5. Can we use a checkpoint path for vertical transfer?
6. Is an initial solution available?
7. Can we expect infinite objective values?
8. Is the objective very expensive to evaluate?
9. Can we chain several optimizers while complying with a deadline?

These are assessed directly from problem configurations, by examining any non-empty TrialHistory (e.g. for item 8), or by making decisions based on on runtime performance (item 10). The flowchart of fig. 3.2 provides an outline of how algorithmic choices are made.

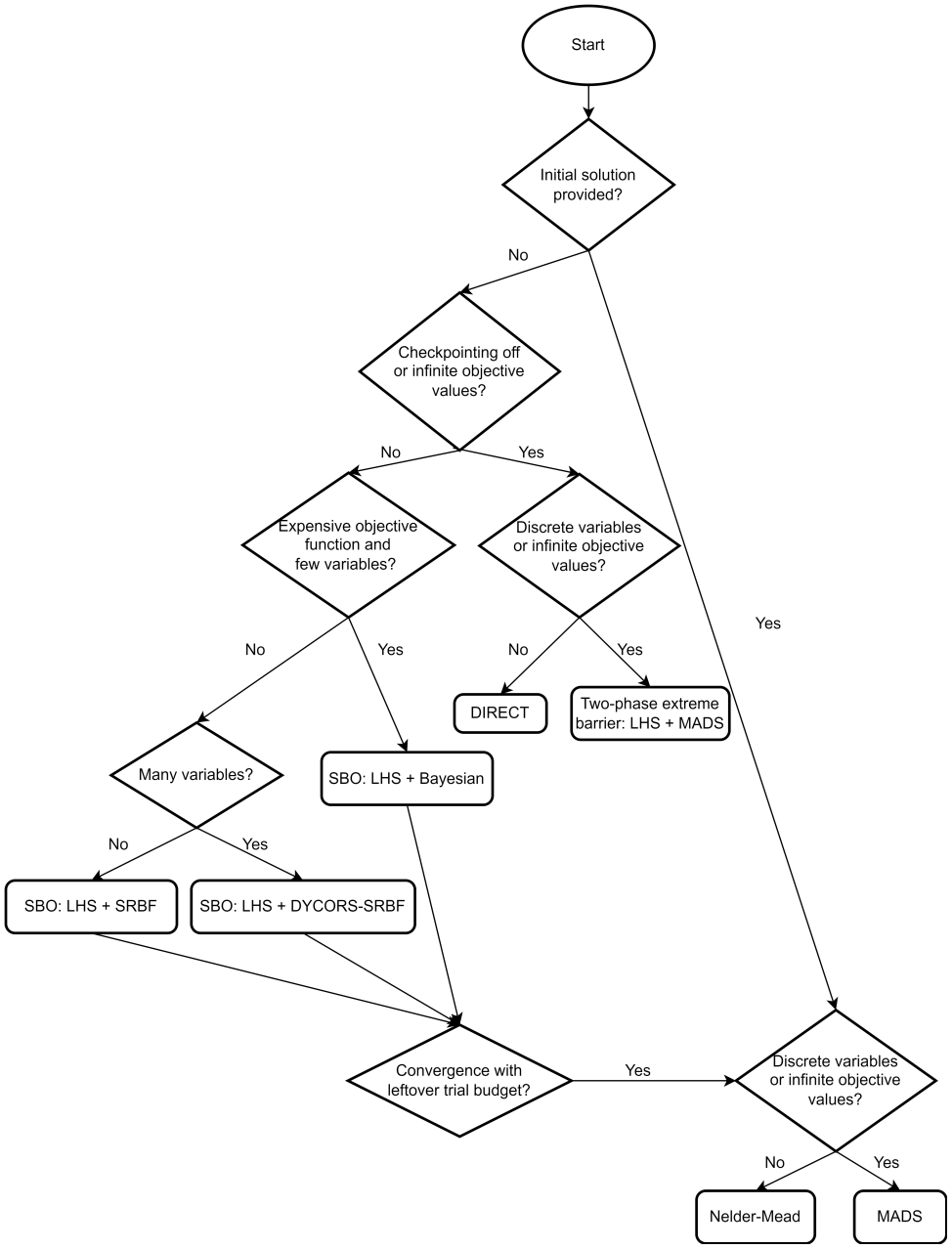

Figure 3.2: An outline of how the wizard picks one or more suite algorithms to handle an input problem. Not depicted here, the wizard will stop chaining algorithms if an input deadline has already passed.

For the secondary purpose of demoing the capabilities of the *entire* suite through the wizard, using e.g. MADS and RBF-based SBO for everything would be a boring magic trick. When the user specifies an initial solution, we will therefore (somewhat questionably) assume that they are interested in a local optimum and route

to either Nelder-Mead or MADS depending on item 2 and 7, with MADS being preferred for handling discrete and infinite values.

If the answer to item 5 is no or the answer to item 7 is yes, then we "cannot possibly" rely on SBO. If the problem is solely a lack of checkpointing, and the answer to item 2 is yes, then we use DIRECT (spuriously correlating "real" with "continuous" for demo purposes). We otherwise employ a hybrid strategy, where we find an initial solution with an LHS, and let MADS improve it - this Global/Direct flow is called the *two-phase extreme barrier strategy* in literature (cf. section 2.4) [11].

Otherwise, we do SBO with an initial LHS, possibly with final solution refinement by MADS, if SBO converges and a possible input deadline allows it (item 10). We only prefer Bayesian Optimization if the objective is "very expensive" (more than 2 seconds to evaluate on average) and there are less than 20 variables involved (item 1) - otherwise we prefer the more light-weight RBFOptimizer, using the DYCORS variant [148] to handle the high-dimensional case. As for the DoE budget, we merge a few known heuristics and allocate $\max(2 \cdot (n+1), \frac{b}{3})$ for it, where n is the number of problem dimensions and b is the total evaluation budget. That is, for safe interpolation, we allocate at least approximately twice the number of dimensions [120], and possibly up to a third of the total budget, if available, for better overall search space coverage [57].

The how-to syntax, designed to mimic what users might expect from Spark SQL [35], is supported by an extension method on the DataFrame class, provided through a Scala implicit class. The idea of supporting how-to queries this way was to make their impact on one's existing Spark configuration as small as possible, without requiring e.g. special syntactic and/or semantic extensions. The method takes case classes as input to disambiguate the different cases:

```
1  implicit class WizardImpl(df: Dataset[_]) {
2  def howTo(maximin: Maximin[_],
3            st: subjectTo,
4            ft: forTrials,
5            wo: Option[withOptions] = None): DataFrame = {
6
7    val wiz = new WizardOptimizer()
8    confWhatIf(wiz, maximin)
9    confsubjectTo(wiz, st)
10   wo.foreach(confOptions(wiz, _))
11   wiz.setNumTrials(ft.num)
12
13   wiz.optimize(df)
14 }
```

That is, we use pattern matching on subtypes of the different input classes (line

2-5) to configure the wizard on lines 7-11 before running the previously described algorithm selector in $optimize(\cdot)$ on line 13. *forTrials* as a case class exists solely to not break the pattern in the resultant query syntax.

3.5 Generic general constraint handling

We now relax the assumption that we are only dealing with bound-constrained mixed-integer problems (cf. eq. (2.3)). As demonstrated so far, such constraints can be handled simply and efficiently by the library, by using some combination of unit scaling, bounds checking and rounding, requiring little to no situational awareness in implementations of $minimize(\cdot)$. While we strive for a similar level of indirection, the implemented handling of general constraints, modelled with equalities and inequalities with respect to zero (cf. eq. (2.4)), is more involved and therefore discussed here separately.

3.5.1 Constraint declaration and Spark Predicates

In the library, general constraints are declared very similarly to how predicates are declared in Spark SQL. In vanilla Spark SQL, one might for instance write a WHERE clause like this in a spatial query [88, p. 88]:

```
1 df.select(col("distance"), col("origin"), col("destination"))
2   .where(col("distance") > 1000)
3   .orderBy(desc("distance"))
```

In programmatic terms, the WHERE clause on line 2 is specified by passing an object of type *Column* to the $where(\cdot)$ method [40]. In fact, *all* method arguments in the previous example are *Column*s: It is the fundamental type for all user-specified building blocks in Spark SQL queries, from literals to column aggregations [88].

During compilation of Spark SQL queries, Column objects are parsed to *Expression*s, which is the type that the Catalyst optimizer works with internally [35]. The Column argument on line 2 in the previous example is parsed to a *GreaterThan* Expression.

For people familiar with compilers and abstract syntax trees, Expressions are predictably *tree-structured*: In our example GreaterThan tree, an Expression subtype corresponding to $col(\text{"}distance\text{"})$ forms a left sub-tree, while an Expression subtype corresponding to literal *1000* forms a right sub-tree. All Column expressions denoting Boolean conditions in Spark SQL are parsed to subtypes of *Predicate*, a subtype of Expression.

The library reuses Spark's Column and Expression class hierarchies, to make general constraint declaration easier to learn and extend for users familiar with standard Spark SQL, while leveraging Spark's abilities as a compiler to provide efficient handling without reinventing the wheel.

From a user's point of view, *Column* is the expected type for specifying general constraints. These constraints refer either to the input or the output of the what-if model $transform(\cdot)$ method - the input being the input DataFrame with concrete variable assignments projected unto it, and the output being whatever DataFrame the what-if Transformer returns. Query specification explicitly disambiguates whether e.g. a column named "x1" refers to a column in the what-if input or output:

```
1  df.howTo(
2    minimize (foo),
3    subjectTo (inputs(expr("x1 + x2 >= 0"),
4                      $"x1" + sum($"a") <= 42),
5               outputs($"x1" + sum($"a") <= 42,
6                       someUDFPredicate($"x1")),
7               -1000.0 <= hcol("x1") <= 1000.0,
8               -1000.0 <= hcol("x2") <= 1000.0),
9    forTrials (bar)
10 ).show()
```

For the MLlib Transformer API, one equivalently uses the $setInputConstraints(\cdot)$ and $setOutputConstraints(\cdot)$ setters to declare these constraints. We specify a total of four constraints here, two on the input, and two on the output of the what-if transformation. Satisfying the constraints on line 4 and 5 may or may not coincide for a given design point, since e.g. input column "a" may have turned into something else after what-if evaluation.

Referring to variable columns as shown (i.e. without aggregation) implicitly uses the $first(\cdot)$ SQL aggregator on the DataFrames. Recalling that variable assignments elicit columnar constants (cf. section 3.3), this allows specifying simple arithmetic constraints like the one on line 3 without tediously having to write e.g. *"first(x1) + first(x2) >= 0"*. The constraints on line 4 and 5 illustrate how it is similarly possible to refer to a mix of aggregations and variable references.

Note that since we reuse Column from Spark SQL, there is a lot of flexibility for how to specify constraints: One can use any function built into Spark SQL (e.g. trigonometric or datetime functions), UDFs (as shown on line 6 above), custom subtypes of Column, or whatever is the most expressive tool.

Also, note that any lack of desired expressivity with the shown Column syntax is likely covered by including a PipelineModel stage for calculating the troublesome constraint value in the what-if model, and include it in the final output to be assessed with an output constraint.

The distinction between input and output constraints is included for the sake of convenience, since it is deemed likely to vary how easy it is to specify a particular constraint with either type. Simple constraints on arithmetic relationships between variables are likely more naturally specified as input constraints, while constraints requiring complex application logic or partial results from objective evaluations to

assess are likely more naturally specified as output constraints.

3.5.2 High-level overview

Before digging into specifics, a few high-level notes on what needs to be accomplished with the library to support common constraint handling methods (cf. section 2.4) are in order.

First, we need to convert the user-provided Column instances, which are to be regarded as Boolean predicates, into a form where we can ultimately assess the violation value (cf. eq. (2.5)) for each constraint, to be used within e.g. penalty functions (cf. section 2.4).

Second, we need to incorporate feasibility checks into the optimization process. This is done through the objective closure, computing each constraint function value based on design points, while making sure that all relevant feasibility information is reflected in Trials used for return values to $minimize(\cdot)$ and for vertical transfer.

For these two issues, we leverage the internal Expression format of Catalyst, but also Catalyst's status as an expression interpreter [52], to elide Spark queries for simple arithmetic input constraints entirely.

As the third issue, we need to support constraint handling strategies in a way that is extensible to both existing general approaches (cf. section 2.4, essentially structured around doing several calls to $minimize(\cdot)$ during a call to $solve(\cdot)$, while allowing for individual algorithms to override library defaults and provide any desired custom handling in e.g. $minimize(\cdot)$. We accomplish this with (you guessed it) a functional *run closure* encapsulating the objective closure, providing a subproblem handle for e.g. penalty and barrier methods.
These three issues are discussed in order next.

3.5.3 From Columns to constraints with Catalyst Expressions

When users more or less overtly set the BlackBoxOptimizer Params corresponding to input and output constraints, the first thing that happens is a *parsing step*. For output constraints, the setter looks like this:

```
def setOutputConstraints(value: Column*): this.type = {
  val (eqs, ineqs) = getPenalties(value.toArray)
  set(outEqConstraints, eqs)
  set(outIneqConstraints, ineqs)
}
```

Method $getPenalties(\cdot)$ parses the input Columns, evaluating to Booleans, to a set of alternate Columns instead evaluating to Doubles, denoting degrees of (in)feasibility. As a part of this step, we make sure to separate equality (*eqs*) from inequality (*ineqs*) constraints, since they require different handling by existing

methods (cf. section 2.4). While there are exceptions to the rule, input constraints are parsed similarly.

Zooming in on how each individual constraint Column is handled, pattern matching on the underlying Spark Expression, accessed by the *expr* field on Column, is used to identify and handle the different cases:

```
1  pred.expr match {
2    /* Rewrite to canonical form first: */
3    case Not(LessThanOrEqual(l, r)) => /* Recursion */
4    /* Etc... */
5
6    // Inequalities:
7    case LessThanOrEqual(l, r) => /* Handling */
8    case GreaterThanOrEqual(l, r) => /* Handling */
9    case LessThan(l, r) => /* Handling */
10   case GreaterThan(l, r) => /* Handling */
11
12   // Equalities:
13   case EqualTo(l, r) => /* Handling */
14   case EqualNullSafe(l, r) => /* Handling */
15
16   // Default fallback:
17   case _ => /* Handling */
18 }
```

So, as indicated on lines 2-4 there is an initial rewriting step in which we unwrap wholesale negation by rewriting the original pattern and recurring on the new one. In the shown case on line 3, we rewrite $\neg(l \leq r)$ to $l > r$, for instance.

As for the remaining cases, recall that all constraint functions can be expressed canonically with either equalities or inequalities with respect to zero (cf. eq. (2.4)). We ultimately want to be able to calculate violation values based on this framework (cf. eq. (2.5)).

A naive approach to accomplish this would be to implement the rule on line 7 as follows:

```
1 case LessThanOrEqual(l, r) => new Column(l) - new Column(r)
```

That is, if we wish to quantify how far $l \leq r$ is from being satisfied with a zero-based inequality, we can construct a new compound Column expression subtracting r from l, since $l \leq r \iff l - r \leq 0$. We can implement similar quantitative rewrites of the original Boolean expressions for the other cases.

The reason why the above handling is deemed ”naive” is that not all input Columns can be safely rewritten to such quantifiable forms, i.e. they ultimately denote *Unquantifiable constraints* (cf. section 2.4.1).

Not all (in)equalities are straightforwardly Quantifiable: Consider for instance *operator overloads* for String comparisons, also available in Spark SQL - unless we

engage in "creative" interpretations of e.g. the Levenshtein distance on behalf of unsuspecting users, we cannot simply quantify a String inequality $leftString \leq rightString$ with $leftString - rightString$. At best such rewrites might raise a few eyebrows, and at worst they elicit runtime errors. Any user-defined operator overloads might lead to similar issues. On top of this, there is of course the case where the user doesn't even use (in)equalities to express constraints. They might just e.g. provide some UDF predicate, evaluated to *true* or *false*. To handle such cases, we have a default fallback option:

```
val default = (!pred).cast(DoubleType)
```

That is, we negate the Boolean value of the original Column expression *pred*, and regard is as a Double. Upon evaluation, we thus obtain 0.0 when the constraint is satisfied and 1.0 when it is violated. Note that this signifies rewriting to a canonical equality constraint (cf. eq. (2.4)), since $pred \iff \neg pred = 0$, for Boolean values evaluated to 1 or 0. This approach of course comes at the cost of quantifiability for guiding optimization, and we will therefore prefer something quantifiable, if we can.

As implemented, we fall back on the above *default* when the user doesn't use (in)equalities. When the user does use these operators, we utilize a combination of SQL $coalesce(\cdot)$, Spark UDFs, and Scala pattern matching to disambiguate quantifiability. For instance, the LessThanOrEqual case is handled as follows:

```
case LessThanOrEqual(l, r) =>
  coalesce(leqV(new Column(l), new Column(r)), default)

val leqV = udf((l: Any, r: Any) => (l, r) match {
  case (l: java.lang.Number, r: java.lang.Number) =>
    Some(l.doubleValue() - r.doubleValue())
  case _ => None
})
```

We thus only assume quantifiability when the operands are numeric JVM types, and otherwise fall back on the *default* option. *leqV* ("V"' is for "Violation") will either evaluate to a feasibility quantifier on numeric types or null-coalesce to the *default* option of rewriting to an Unquantifiable equality constraint in canonical form. In doing this rewrite with $coalesce(\cdot)$, re-using the original comparison operands at different steps, we rely on Spark SQL not eagerly re-evaluating e.g. the same expensive aggregation expression twice [35].

The reason for using such an arguably convoluted solution stems from the fact the we are engaging with Spark internals on a purely *syntactical level* here, and that Spark is (sometimes unfortunately) very extensible. We don't know the runtime type of any operands of the comparison operators, and type information of identifiers, UDFs and built-in functions (e.g. $pow(\cdot)$) alike, are unresolved at this point. What we are making is essentially a "macro" substitution, with all missing

type information to be resolved in the runtime SparkSession context.

Nonetheless, on the bottom line, we can convert any set of Boolean-valued Column expressions to a corresponding set of new Column expressions, representing general constraint functions in canonical form.

3.5.4 Constraint function evaluation

Constraint function evaluation is handled by the objective closure, as defined by relevant runtime information in $solve(\cdot)$ (cf. section 3.4.2). Upon each trial, the objective closure will ensure that any relevant Column expressions signifying constraint functions are evaluated with respect to the what-if input and/or output DataFrame, and that all constraint function values are provided as vectors on the generated Trials.

A few overarching design choices for constraint function evaluation with respect to the QRAK taxonomy (cf. section 2.4.1) should be noted before going into implementation details, the issue of quantifiability already having been discussed in the previous section.

It is important to note that the library assumes that all constraints besides bound constraints are *Relaxable* (cf. section 2.4.1). That is, besides ensuring variable domains, we generally don't try to arbitrate *meaningfulness* on behalf of users: We just evaluate the objective and constraint functions as provided and leave their resultant values to the individual constraint handling strategy.

We take the same stance on *Hidden* constraints, only known through crashes or anomalous behavior, since it is similarly impractical to make guesses on intended behavior or arbitrate the meaningfulness of "2.6" system-side.

As for a priori vs. simulation constraints, we don't try to elide what-if model evaluations on the basis of e.g. input or A Priori constraints being violated either (cf. section 2.4.1). In our library setting, we are generally interested in retending *all* Trial information across several runs, to support vertical transfer.

Back to how constraint functions are handled in the objective closure - in the general case, we put our normalized constraint function Columns into Array Column expressions, and evaluate them in Spark queries, obtaining one Double Array of constraint function values for inequality and equality constraints, respectively. For the output constraint Columns, we first construct a Map like this:

```
1 val outPenMap = Map(eqCol -> $(outEqConstraints),
2                     ineqCol -> $(outIneqConstraints))
```

Where *eqCol* and *ineqCol* are just some unique column names, and the attached values are Column expressions encapsulating all parsed constraint function Column expressions in a Spark Array. We can ultimately evaluate all output constraint functions together with the objective value of the what-if model like this:

```
val row = whatIf.transform(df.withColumns(cols))
  .withColumns(outPenMap)
  .head
```

From this row, we can obtain e.g. the Array of inequality constraint values like so:

```
val ineq = row.getAs[Seq[Double]](ineqCol).toArray
```

Such Arrays of constraint function values are ultimately aggregated on the Trial class, and are thus available for various useful applications library-wide.

We can similarly handle input constraints by doing a separate query on the what-if input DataFrame, and so on:

```
val inPens = df.withColumns(cols).withColumns(inPenMap).head
```

However, this approach for handling input constraints feels bad for several reasons. First, note that DataFrames are *immutable*. This entails that we might wastefully do non-trivial calculations with identical results as many times as we do Trials in a run. For a non-variable column "a", we might therefore for instance recalculate $sum(\$"a")$ in an input constraint over and over for a huge input Dataset.

Another reason why this feels bad is that, unlike the output constraint vector retrieved in the same query as the objective, we are executing a separate Spark query with its own overhead to evaluate input constraints - firing up powerful computational artillery to evaluate expressions like e.g. $x1 + x2 >= 0$ seems especially wasteful.

To handle this problem, I was inspired by the Python optimization library known as Mystic [121], which offers relatively elaborate constraint handling. One such facility is the one of *symbolic constraints*, which are user-provided algebraic String expressions like $"x1 + x2 >= 0"$, evaluated by an *interpreter* to assess feasibility.

Luckily I didn't have to write an interpreter from scratch, since Spark SQL already has a quite capable one: Catalyst can evaluate simple Expressions without involving any Spark sessions or queries, using the $eval(\cdot)$ API [52]. We might for instance write:

```
expr("2 + -2 >= 0").expr.eval()
```

Which returns a value of *true*. Note however that we are quite limited in what we can do here: Without any Spark session or end-to-end query processing involved, there is namely no symbol table of user-specified identifiers available, precluding the usage of column names, UDFs, and the like. We can basically only use literals along with the suite of Spark SQL built-in functions and operators (like e.g. $<=$ and $pow(\cdot)$).

We can however in many cases *precompute* partial results, e.g. aggregations and UDF calls, once and for all, in a single Spark query, and *graft* them unto the original

Expression tree as literals. This elicits new Column expressions from the parsing phase, which are primed for constraint evaluation with Catalyst. Further substituting variable references with literals in the objective closure, we can evaluate input constraints with $eval(\cdot)$, i.e. without executing any Spark queries. This general idea outlines the approach followed in the implementation.

Before venturing any further, it should be acknowledged that our usage of Catalyst as a symbolic constraint interpreter deviates from its originally intended purpose, debugging during development [52], by a significant margin, and could thus be regarded as a tad "abusive". Nonetheless, within the context of this being a mere student project for learning purposes, I decided to go in this direction, to have some fun with the concept without e.g. having to write a feature-complete Spark SQL interpreter from scratch.

The first step of providing Catalyst interpretation of input constraints is to figure out which input constraints can actually be handled this way. Note that we can potentially precompute aggregation and UDF values during parsing in a single "normal" Spark query, but only if we know the column arguments beforehand - this is not the case for variable columns, which are only instantiated upon objective evaluations. As the first step of input constraint parsing, we therefore separate the wheat from the chaff, based on which strategy is appropriate:

```
1 val (resInCts, unresInCts) = inCts.partition(c =>
2   isSingleQueriable(c.expr, vars, builtins))
```

That is, we partition into resolvable and unresolvable input constraints based on the *isSingleQueriable* predicate. Resolvable constraints can be handled by Catalyst, while we fall back on the previously shown default handling strategy for the other kind.

isSingleQueriable is called with the underlying Expression of the user-provided Column constraint, the array of decision variables, along with a symbol table of Catalyst-interpretable Spark built-in functions, obtained from the *SparkSession.catalog* field [132]. In words, the aforementioned predicate simply checks whether there are any variable column references in arguments of non-built-in functions or aggregations. This is done by recurring down the Expression tree and disallowing variable column references in e.g. UDF sub-trees:

```
1 case e: UnresolvedFunction if !(builtins contains e.prettyName)
      =>
2   e.children.forall(isSingleQueriable(_, vars, builtins,
      varsAllowed = false))
```

In this case, we just check whether the function name, as used in e.g. *expr* Strings, refers to a non-built-in, and if yes, then no sub-tree may contain variable references, since we ultimately cannot evaluate such expressions outside of a Spark session. A limitation in this approach is of course that we assume that users don't

have a habit of defining new methods named e.g. ”pow” taking three strings, or similar - Spark is indeed very extensible, which can also be a design challenge.

We do the same thing for AggregateExpressions, i.e. aggregations. Doing aggregations over constant columns might seem a bit ”weird” in the first place, but we nonetheless aim for consistency.

Constraint Columns compliant with the aforementioned rules are subsequently converted into new Column expressions, in which we basically precompute subtrees not containing references to variables or built-ins and replace the original subtrees with Literal Expressions. This pre-computation happens once during runtime, within the context of the SparkSession attached to the input Dataset.

We first find the Expression subtrees in need of precomputation, including aggregations, UDF calls, and references to non-variable columns:

```
val toResolve = constraints.map(c => unresolvedSubtrees(c.expr,
    vars, builtins))
```

These Expressions are then converted into valid Columns to be retrieved in one fell-swoop Spark query on the input DataFrame:

```
val graftExprs = toResolve.map(graftArr)
val row = df.select(graftExprs:_*).head
```

A bit later, we have obtained mappings from the original Expression subtrees to their pre-computed Literal equivalents, which are then used for simplifying all relevant subtrees, and obtaining a new Column expression for all resolvable input constraints:

```
val grafts = (constraints zip graftMaps)
  .map{case (c, gMap) => new Column(graftedExprTree(c.expr,
   gMap, vars))
```

To summarize, when we are dealing with input constraints like *$”x1” + sum($”a”) <= someUDF(42)*, we can simplify them to Column expressions like e.g. *$”x1” + 9001 <= 84* during parsing as shown above, doing the usual penalty function conversions with coalesce etc. afterwards (cf. section 3.5.3).

Then we can evaluate these constraints separately upon each trial in the objective closure, by interpretation with Catalyst, after having substituted in concrete variable assignments as literals. With the concrete variable assignments given in a Map named *graft*, constraint evaluation with the Catalyst interpreter looks something like this:

```
instantiatedTree(constraint.expr, graft).eval()
```

This procedure notably doesn’t involve wastefully recalculating results or the overhead of executing an end-to-end Spark query.

3.5.5 Supporting constraint handling strategies

So far, it has been described how we obtain separate vectors containing constraint function values for canonical equality and inequality constraints, aggregated on each completed Trial. They are not worth the hassle without a constraint handling strategy to leverage them, however.

As mentioned in the relevant background section (cf. section 2.4), some constraint handling techniques are idiosyncratic to individual algorithms, while others strive for a higher level of abstraction. My aim was to accommodate both approaches in the library, thus making it possible for implementations of *minimize*$(\cdot)$ to do whatever is desired with Trial feasibility information retrieved from the objective closure, while also supporting implementations that basically only inspect the *objective* field on the Trial class and delegate general constraint handling to a generic strategy.

Note that both barrier and penalty methods require running the same optimization algorithm several times in a sequence of subproblems with different settings regarding some penalty function (cf. section 2.4). Implementation-wise, a consistent, extensible way of incorporating such subroutines into *solve*$(\cdot)$ was needed, while not imposing forced choices on implementers preferring to keep constraint handling inside *minimize*$(\cdot)$.

To support methods structured around doing several optimization runs, we therefore declare a *run closure* in *solve*$(\cdot)$, with the following signature:

```
1 def runClosure(initSol: Option[Array[Double]],
2                penalty: ObjectivePenalizer,
3                runBudget: Int): Option[Trial]
```

This closure takes an initial solution, a penalty function, as well as a prescribed trial budget for this run. The penalty function calculates a *penalized objective value* from the raw objective value along with two feasibility vectors. The run closure is actually the method that calls *minimize*$(\cdot)$, and will ultimately return the best Trial found by one such call. The body of the run closure does a few important things related to constraint handling.

It relays the current *penalty* measure (including its magnitude) to the objective closure of the run, such that the objective closure will penalize all Trials returned to *minimize*$(\cdot)$ accordingly, allowing the latter algorithms to interface with the problem as an unconstrained one. The objective closure calculates the *objective* field of the returned Trial from the raw objective and feasibility vectors like so:

```
1 val penalizedObj = penalty(objValue, eqPen, ineqPen)
```

For the TrialHistory provided as input to *minimize*$(\cdot)$, the run closure also *reprojects* all Trials seen so far, such that the objective field, inspected by most algorithms, reflects the current penalty measure. We accomplish this with a UDF:

```
1 val penUDF = udf(penalty)
2 val projObj = penUDF(col("rawObjective"),
3                      col("eqPenalties"),
4                      col("ineqPenalties"))
5
6 val runHistory = getHistory(spark, historyPath).map{ his =>
7   TrialHistory(his.trials.withColumn("objective", projObj))
8 }
```

As shown on line 2 and 7, Trials actually have fields for both a ”raw” (i.e. unpenalized) objective value and a penalized value, ”objective”, making it possible for algorithms to use both separately, if needed. When algorithms thus inspect the *objective* field of a returned Trial from the objective closure, it reflects any current measure of penalty.

Note also how we on line 6 actually *reload* the TrialHistory as a new DataFrame from the checkpoint path - this is done such that e.g. Model-based algorithms can interpolate all Trials seen across *all runs* with different penalty measures, and thus make incremental progress without wastefully reevaluating the same points again. If we didn't do this, algorithms would only see Trials in the DataFrame loaded initially in $transform(\cdot)$, due to the immutability of Datasets.

In $solve\cdot)$ the run closure is finally passed on to a *SolutionStrategy*, which decides what to do with it. Instances of this class implement penalty methods, barrier methods and the like, utilizing the run closure and other relevant information. Preferred constraint handling strategies are specified through a Param, looking up in an extensible catalog of SolutionStrategy's aggregated on BlackBoxOptimizer, like so:

```
df.howTo(
  minimize (foo),
  subjectTo (bar),
  forTrials (baz),
  withOptions ("constraint handling" -> "penalty method")
).show()
```

Through overriding, each concrete subclass can add or remove strategies from the catalog Map, based on their merits, e.g. the extreme barrier method might not be the best choice for Model-based optimizers, but is a good choice for MADS [11].

$solve(\cdot)$ kicks of optimization by calling the $run(\cdot)$ method of the preferred SolutionStrategy. The simplest solution strategy does nothing to enforce feasibility - we call this strategy *LaissezFaire*, and its $run(\cdot)$ method is declared like so:

```
1 override def run(initSol: Option[Array[Double]],
2                  minimize: RunClosure,
3                  history: => Option[TrialHistory],
4                  solveBudget: Int): Option[Trial] = {
```

```
5   minimize(initSol, noPenalty, solveBudget)
6 }
```

So, the method is relayed any initial user-specified solution, the run closure (calling *minimize*$(\cdot)$, with some housekeeping on top), a TrialHistory handle (if needed), along with the *total* evaluation budget connected to the calling instance of *solve*$(\cdot)$. While budgetary matters are mostly relevant when running with parallelism, note that different calls to *solve*$(\cdot)$ get their own piece of the total evaluation budget specified by the user. It is the grown-up responsibility (cf. design Assumption 3.1) of SolutionStrategy's to decide what to do with the assigned solveBudget. As seen on line 5, LaissezFaire just spends the entire thing on a single call to the run closure, returning the result.

noPenalty$(\cdot)$ is sort of a dummy implementation of a penalty function, in that it just relays the original objective value without modifying it:

```
1 def noPenalty(objective: Double,
2               eq: Array[Double],
3               ineq: Array[Double]): Double = objective
```

The purpose of LaissezFaire is simply to provide a baseline for algorithms implementing their own custom constraint handling in *minimize*$(\cdot)$, and to demonstrate the basics in this document, of course.

More interestingly, a basic penalty method approach (cf. section 2.4) is implemented in the *PenaltyMethod* class. It divides the total budget into a Param-configurable number of bins, and proceeds to call *minimize*$(\cdot)$ in a loop, using a configurable penalty function of varying magnitudes:

```
1 for (budget <- budgets; if !feasible || hasHistory) {
2   val currentPen = penalizer(magnitude)
3   res = minimize(x, currentPen, budget)
4   x = res.map(_.solution)
5   magnitude *= magAdjustment
6   feasible = res.exists(_.isFeasible)
7 }
```

As shown on line 1, it terminates this process once it has found a feasible solution (*!feasible*), as per tradition for penalty methods (cf. section 2.4) - unless we are running with vertical transfer on (*hasHistory*), in which case we have "nothing to lose" from trying to improve the solution further.

The initial *magnitude* and adjustment factor *magAdjustment* are configurable hyperparameters, with default values 1.0 and 2.0, respectively. PenaltyMethod has its own catalog of penalty functions, all parameterized with a magnitude. The quadratic penalty is for instance implemented like this:

```
1 def quadraticPenalty(objective: Double,
2                       eq: Array[Double],
3                       ineq: Array[Double])
```

```
4                       (magnitude: Double): Double = {
5   val eqPen = eq.map(vi => pow(abs(vi), 2)).sum
6   val ineqPen = ineq.map(vi => pow(max(0, vi), 2)).sum
7   objective + magnitude * (eqPen + ineqPen)
8 }
```

As seen on line 3 here, and line 2 in the PenaltyMethod snippet from before, we use a separate parameter list to instantiate a penalty function with the desired magnitude.

Another important SolutionStrategy is the *ExtremeBarrier*, which is exactly equivalent to LaissezFaire, except that it uses this penalty function instead of $noPenalty(\cdot)$:

```
1 def deathPenalty(objective: Double,
2                  eq: Array[Double],
3                  ineq: Array[Double]): Double = {
4   objective + (if (eq.exists(_ != 0) || ineq.exists(_ > 0))
5                 Double.PositiveInfinity else 0)
6 }
```

The above examples outline how one can define a variety of different constraint handling strategies utilizing penalty or barrier functions on top of core sBBO algorithms, notably with a layer of *indirection* in-between: Algorithms can be completely oblivious to such methods being used on top of them, and vice versa.

Furthermore, BlackBoxOptimizer subclasses can override and expand on the constraint handling catalog as they please - e.g. use LaisseFaire and implement a filter method or some custom constraint handling method inside $minimize(\cdot)$, using Trial feasibility information.

To the best of my knowledge, the library thus accommodates all kinds of constraint handling strategies commonly used within sBBO today (cf. section 2.4).

There is of however one joker here, ruining the illusion of harmony: that of *infinitely valued* outputs of objective, penalty or constraint functions. The ability to handle such values, as used within e.g. the extreme barrier approach, depends on the individual algorithm, with the only real option within sBBO being directional methods (cf. section 2.3.3).

The best solution I've seen for fixing possible numeric instability in Model-based algorithms is to "dampen" infinite objective values by replacing them with the *square* of their interpolated value in the surrogate model [57]. Problems here of course include the identity operation of squaring 0 or 1, and that squaring may neither be sufficient to ensure numeric stability nor ensure a preference for feasible solutions during optimization.

Ultimately, users therefore regrettably have to decide whether they really need to use infinite values in their problem model, and in such cases, pick an algorithm capable of handling them. In the library suite, obtaining a good initial solution with a sizeable LHS, followed by running MADS with e.g. the extreme barrier

strategy, would be the way to go, and a simple user recommendation to provide - this is also the approach used by the library wizard upon detecting possible numeric instability (cf. section 3.4.4).

3.5.6 The Historical Revisionist Method

Out of the box, the library offers a suite of sBBO algorithms along with a suite of SolutionStrategy's, implementing various penalty and barrier approaches, thus supporting generally constrained optimization (cf. eq. (2.4)). I ultimately decided to not implement filter approaches for individual algorithms, since this time investment would only have localized benefits to the library. Instead the filter approach inspired a new constraint handling method that is more generic.

As default library facilities, penalty and barrier methods each have their own issues (cf. section 2.4). While the overhead of solving the same problem over and over again has been somewhat ameliorated by reloading TrialHistory on each run, thus snowballing progress in e.g. surrogate model building, the issue of finding the right magnitude and adjustment factor for guiding optimization remains.

Reprojecting TrialHistory to reflect the current measure of penalty was originally just a bug fix. However, the level of control over what information is available across different calls to $minimize(\cdot)$ afforded by this relational operation, essentially providing an SQL view, got me wondering, whether an alternate approach for constraint handling could be built around it: Enter, the *Historical Revisionist Method* (HRM).

To explain the name, *historical revisionism* denotes the human activity of reinterpreting history to reflect current knowledge or motives [33]. For instance, the "good guys" tend to win in the end, since history is always written by the victor. In the HRM approach to constraint handling, feasible solutions are the "good guys", and SQL is the victor.

Like penalty methods, HRM is an exterior method, progressing towards feasibility and being *orthogonal* to the underlying sBBO algorithm by incorporating a penalty term in the objective across several runs. Like barrier methods, HRM tries to maintain an *invariant* preference for feasible solutions during optimization. Like filter methods, HRM leverages previous evaluations as *data* for guiding optimization and enforces something akin to a dominance hierarchy among points. None of the aforementioned methods possess all of these traits by themselves. Used car salesmanship aside, it should also be emphasized that HRM still has downsides, as we shall see.

The overall framework is very similar to the one for penalty methods shown in eq. (2.32), in that we solve a series of subproblems with an augmented objective function:

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad \Pi_i(x) \tag{3.2}$$

Where $\Pi_i : \mathbb{R}^n \to \mathbb{R}_{\geq 0}$ is the *historical penalty function* used in subproblem i, defined differently depended on the composition of data in TrialHistory. Note the immediate absence of any adjustable magnitude. Instead of making the user guess the right value a priori or increase it by a constant factor blindly, we formulate a different historical penalty function on each iteration, incorporating a magnitude fitted on the history of evaluations so far.

The library implementation of the main loop goes like this:

```
for (budget <- budgets) {
  val penalty = historicalPenalty(history, corePenalty)
  res = minimize(x, penalty, budget)
  x = res.map(_.solution)
}
```

That is, we just split the overall budget over a number of runs, parameterized with a different measure of penalty, based on the TrialHistory so far, along with a *core penalty function*, e.g. the mixed or quadratic one (cf. section 2.4). Note that unlike a traditional penalty method, we never stop upon finding a feasible solution. This stems from the fact that running with a new historical penalty function solely signifies *recalibrating* optimization as opposed to *restarting* it.

The recalibration frequency as well as the core penalty function are hyperparameters of the method. The library defaults to recalibrating the penalty measure about every 10% of the total run, along with using the mixed penalty (cf. eq. (2.36)), combining a smooth penalty function with an exact one. Note that due to the vertical transfer mechanism, users making educated guesses on these parameters up-front is not crucial, since no Trial information is lost across different calls to $optimize(\cdot)$, and HRM statelessly progresses on what is already known about the search space, without having to iterate through a different number of predetermined magnitude settings first, unlike classic penalty methods.

The interesting part is of course how $historicalPenalty(\cdot)$ is implemented. algorithm 2 is an attempt at outlining what happens in there. In what follows, note that "penalty functions", as a library construct and as depicted in algorithm 2, actually calculate *penalized objective values*, and not just penalty terms to be added to raw objective values.

In brief, we use the TrialHistory so far to build an augmented penalty function around the core penalty, incorporating scaling functions and a calibrated magnitude - that is, algorithm 2 is a higher-order function, returning a (you guessed it) functional closure. The augmented penalty function is created such that we maintain an objective value ranking, based on the ordering scheme known within the constrained optimization world as the *Superiority of Feasible Solutions* (SFS) [95]:

Algorithm 2 Historical Penalty

1: **procedure** HP($history$, $corePenalty$, $baseMag = 1$, $\delta = Double.delta$)
2: **if** $isFeasible(x), \forall x \in history$ **then**
3: **return** $corePenalty(baseMag)$
4: **else**
5: $ineqMax_j \leftarrow \max\{\max(0,\ g_j(x))\ :\ x \in history\}, \forall j \in [1..p]$
6: $eqMax_k \leftarrow \max\{|h_k(x)|\ :\ x \in history\}, \forall k \in [1..q]$
7: $objMin \leftarrow \min\{f(x)\ :\ x \in history\}$
8: $objMax \leftarrow \max\{f(x)\ :\ x \in history\}$
9: ▷ *scaler* scales $f(\cdot)$, $g(\cdot)$ and $h(\cdot)$ values and passes them to *corePenality*:
10: $scaled(mag) = scaler(corePenalty, ineqMax, eqMax, objMin, objMax, mag)$
11: ▷ Reprojects *history* with base magnitude:
12: $scaledHis \leftarrow \{scaled(baseMag)(x)\ :\ x \in history\}$
13: ▷ Find minimum magnitude for proper SFS ranking:
14: $equationMag = \max\left\{\frac{f_s(x_i)+\delta-f_s(x_j)}{P_s(x_j)-P_s(x_i)} : x_i,\ x_j \in scaledHis \wedge P_s(x_i) < P_s(x_j)\right\}$
15: **return** $scaled(\max(equationMag,\ baseMag))$

- Prefer feasible solutions among feasible and infeasible solutions.
- Prefer lower objective values among feasible solutions.
- Prefer lower constraint violation values among infeasible solutions.

It basically means that constraint violation and objective values are respectively used as the primary and secondary sort key of a *lexicographic order*. SFS is a commonly used ordering for comparing solution quality among different algorithms in a benchmark setting [95].

In HRM, we essentially *fit* an augmented penalty function to the TrialHistory dataset, our "filter", such that the SFS ranking scheme holds for objective values among known points in the search space - our own notion of "domination" (cf. section 2.4). The reprojected TrialHistory dataset given to the algorithm on the next run, along with the fitted objective function, will thus reflect this prioritization scheme, and thereby our priorities within constrained optimization as such, while accomplishing what penalty functions are supposed to.

While we can of course only do estimates based on existing Trial data, the idea is that subsequent runs will tend to operate under more and more accurate pretenses with an expanding TrialHistory database.

As shown in algorithm 2, there are two different cases for fitting the augmented penalty function. When there are no infeasible Trials to reason about, or no Trials at all, we default to using the core penalty function with the base magnitude of 1 on the next run (cf. line 2-3, algorithm 2):

$$\Pi_i(x) = f(x) + P(x) \tag{3.3}$$

When infeasible solutions are present, the augmented penalty function instead looks like this:

$$\Pi_i(x) = f_s(x) + \rho \cdot P_s(x) \tag{3.4}$$

Where $f_s(\cdot)$ and $P_s(\cdot)$ are *unit-scaled* objective and penalty functions, respectively, and $\rho \geq 1$ is a magnitude fitted to TrialHistory data. Note that HRM additionally modifies the original objective function, on top of adding a penalty term to it, for numerical reasons.

Scaling constraint violation values is just a general recommended practice: It is done to overcome the problem that e.g. Unquantifiable constraints, eliciting violation values of 1 (cf. section 3.5.3), are insignificant blips on the radar versus Quantifiable constraints operating on a scale of e.g. thousands, even though both are in a sense equally important feasibility-wise.

Scaling constraint functions is however easier said than done, due to the fact that the range of possible violation values for a given constraint function is unknown in the general sBBO case, and we therefore don't know the right scaling parameters for each constraint a priori. We will however make *a posteriori estimates* here, based on TrialHistory data so far. We use *min-max scaling* throughout:

$$scaled(x) = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{3.5}$$

Each constraint function gets its own scaling parameters (cf. line 5-6 in algorithm 2). For each inequality constraint function $g_j(\cdot)$, we set its $x_{max}^{g_j}$ parameter to be the maximum positive value of $g_j(x)$ found in TrialHistory, or zero if no such value exists. We similarly set $x_{max}^{h_k}$ for each equality constraint function $h_k(\cdot)$ to be the maximum value of $|h_k(x)|$ found in TrialHistory. When we ultimately scale violation values with the $scaler(\cdot)$ on line 9, we exploit the fact that no infeasible violation value equals zero and set the corresponding values of x_{min} to zero for both equality and inequality constraints. This way, parameters are fitted such that all infeasible values of $g_j(x)$ and $|h_k(x)|$ in TrialHistory can be scaled into the range of $(0,\ 1]$.

We similarly estimate parameters for min-max scaling objective values to lie in the range of $[0,\ 1]$, based on the maximum and minimum values of $f(x)$ found in TrialHistory, for numerical reasons to be explained (lines 7-8, algorithm 2).

With the $scaler(\cdot)$ on line 9, all these parameters can be used for scaling the objective and infeasible violation values in the augmented penalty function, before passing them to the core penalty function:

```
1 corePenalty(scaledObj, scaledEqs, scaledIneqs)
```

Where with *gj* as $g_j(x)$ and *gjMax* as $x^{g_j}_{max}$, each value of scaledIneqs is:

```
if (gj <= 0.0) gj else scaled(x, gjMax)
```

And with *hk* as $h_k(x)$ and *hkMax* as $x^{h_k}_{max}$, each value of scaledEqs is (retending the original sign with $signum(\cdot)$):

```
if (hk == 0.0) hk else signum(hk) * scaled(abs(hk), hkMax)
```

And scaling is done by:

```
def scale(v: Double, vMax: Double) =
  if (vMax == 0.0) v else v / vMax
```

We thus leave feasible values unchanged (their violation values being zero in all penalty functions, by definition), and don't try to scale infeasible violation values for which we don't know anything about the infeasible range (yet).

The remaining handling in algorithm 2 is about finding a magnitude to be used with the core penalty function, fitted to TrialHistory such that its objective values reflect the SFS scheme. The reason for also scaling the objective value is ultimately to ensure that SFS can be enforced through a magnitude, without making penalized objective values obscenely large, and hence numerically unstable, when infeasible solutions have really good objective values.

When infeasible values are present in the data, we find a sufficiently large magnitude, such that smaller penalty values imply smaller objective values, as per SFS. We first use the functional closure on line 9 to rescale all points in history with the found parameters and the base magnitude of 1 (cf. line 12, algorithm 2). Then, let $f_s(\cdot)$ and $P_s(\cdot)$ still respectively be the re-scaled objective and core penalty functions. For each pair of points x_i, x_j in the rescaled TrialHistory such that $P_s(x_i) < P_s(x_j)$, we solve equations on line 14 for the magnitude ρ, trying to ensure the following property:

$$f_s(x_i) + \rho \cdot P_s(x_i) < f_s(x_j) + \rho \cdot P_s(x_j) \tag{3.6}$$

That is, we look for sufficiently large values of ρ, such that better feasibility values always take precedence over better objective values among all known points, ensuring SFS ranking among penalized objectives in TrialHistory. At the same time, we don't just want to set ρ to be e.g. *Double.MAX_VALUE* and call it a day, since this might jeopardize numeric stability, exacerbate non-smoothness, and/or lead to the penalty term vastly outweighing the influence of the objective function during optimization. We instead leverage the fact that optimization algorithms tend to only work with objective values through relative comparison (i.e. "better is better" regardless of scale), and aim for a *minimum* magnitude to obtain the proper SFS ranking, so the optimization algorithm gets the story straight.

Using a small delta for the Double type, $\delta > 0$, to minimally ensure strict inequality, we can solve eq. (3.6) problems as equations in closed form, in $\mathcal{O}(h^2)$ total time for h points in TrialHistory, as shown on line 14 in 2:

$$f_s(x_i) + \rho \cdot P_s(x_i) + \delta = f_s(x_j) + \rho \cdot P_s(x_j) \iff \rho = \frac{f_s(x_i) + \delta - f_s(x_j)}{P_s(x_j) - P_s(x_i)} \tag{3.7}$$

The magnitude that will be used by the core penalty function for the next run is the largest ρ among all solutions found in these equations, or the base magnitude of 1, whichever is larger (cf. line 15, algorithm 2). Note that feasible solutions are already trivially in proper SFS order among themselves, all having zero-valued penalty terms. As a performance boost in the implementation, we can therefore get by with only including the worst feasible point (SFS-wise) in the pool of equations to be solved on line 14, ensuring the proper ranking by transitivity.

To summarize, HRM works by regularly fitting a new augmented penalty function to all evaluation data seen so far, ensuring that objective and constraint functions operate on similar, numerically tame scales, and that an SFS ranking holds among penalized objective values: TrialHistory, as understood by the underlying algorithm on the next run, paints feasible solutions as strictly better than infeasible ones, better objective values as strictly better for feasible solutions (trivially so), and smaller constraint violation values as strictly better for infeasible solutions. The new objective thus reflect our priorities in constrained optimization.

At its core, HRM is best described as a reformation of dynamic penalty methods, as a minor modification of the usual penalty method framework (cf. section 2.4), and not some revolutionary new approach to constraint handling.

The main novelty here is that the core penalty function is adjusted in an entirely *data-driven* way, without requiring any user guesses on adjustment factors or magnitudes - on top of core library mechanisms (e.g. the objective closure) ensuring that no knowledge about design points is lost across runs when using vertical transfer, removing a serious overhead using the vanilla penalty framework as-is.

Borrowing some useful ideas from barrier and filter methods to automate the parameter fitting process, while remaining a mostly orthogonal method to the algorithm-problem pair, HRM thus has some attractive properties from a systems design and user-oriented perspective, as a ”plug-and-play option” for general use.

However, being essentially a weird penalty method, HRM carries with it some of the same quirks: As an method-agnostic approach, its high level of indirection implies that it generally cannot be expected to outperform a constraint handling method exploiting properties of individual optimization algorithms or problems.

Also, HRM is built around the idea that analyzing h TrialHistory elements in $\mathcal{O}(h^2)$ time now and then, and reinitialize the underlying algorithm, won't become

a serious performance issue when piloting the method. Given the core premise of sBBO, we are certainly not too concerned about the TrialHistory dataset having a troublesome size, and simply analyze it in-memory on the driver. Still, this should be mentioned with respect to how the method generalizes to different settings.

As other method-specific issues, how HRM performs will depend highly on the composition of the TrialHistory datasets analyzed to fit the augmented penalty function, which depends on the underlying problem-algorithm pair. Still, the performance of classic penalty methods similarly depends on the mental disposition of the piloting user, for choosing the right hyperparameters.

As an additional issue, we of course assume that users don't mind running optimization with checkpointing enabled when using HRM - however, given that we are dealing with sBBO, checkpointing progress may not be that hard of a sell.

Nonetheless, HRM can be regarded as a possible alternative to running with a basic penalty method, with its merits to be assessed empirically, of course.

3.6 Multi-level parallelism

DIBBOlib offers two tiers of parallelism: Running several trials in parallel in $minimize(\cdot)$, and running several $solve(\cdot)$ calls in parallel in BlackBoxOptimizer's $transform(\cdot)$ method (cf. table 3.1), to be described here in turn.

3.6.1 Trial parallelism

As shown with $minimize(\cdot)$ in section 3.4.2, each run is afforded an instance of the Evaluator class. From the point of view of the sBBO algorithm, Evaluator just exposes a few utilities concerning objective evaluations: fields informing about how many trials are left and whether it is time to terminate, along with methods for trialling one or more design points.

Under the hood, Evaluator also plays the role of being the *single entry point* for trial parallelism when enabled. For example, BayesianOptimizer will ask its Evaluator, *eval*, to trial a number of points, based on a suggestion from the *parallelism* field on Evaluator:

```
1 // Propose points by adaptive sampling:
2 val proposals = bo.proposePoints(eval.parallelism)
3
4 // Do trials with Evaluator:
5 val arrs = // Map proposals to Arrays
6 val evals = eval.objective(arrs).flatten
```

By library default, *eval.parallelism* on line 2 will just evaluate to 1. Dedicated BlackBoxOptimizer Params can however be used for specifying higher levels of

parallelism, which will thus lead to more proposals for evaluation in the previous example. The *objective* method on line 6 is declared like this in the abstract Evaluator class:

```
1 def objective(pts: Array[Array[Double]]):
2   Array[Option[Trial]] = {
3   val futures = pts.map(p =>
4     Future[Option[Trial]](obj(p))(executionContext))
5   futures.map(threads.awaitResult(_, Duration.Inf))
6 }
7
8 protected val executionContext: ExecutionContext
```

Under the hood, the Scala Futures API is used for evaluating the objective closure (named $obj(\cdot)$ on line 4) with various levels of concurrency, before relaying the results back to the caller like nothing happened.

Depending on Param settings and Evaluator subtype, these tasks will either be executed in the calling thread (no concurrency) or in an internally maintained thread pool of *static* or *dynamic* size (to be explained). These different ”execution contexts” is what the ExecutionContext on line 4 ultimately specifies, depending on user settings.

Me having limited experience with concurrency on the JVM platform, I decided not to ”roll my own” at the onset. It should be acknowledged that this implementation of thread pools, combining Futures with various ExecutionContexts, is not created by me, but reuses open source code from Spark Core.

Note also that this implementation assumes throughout, as per a gentleman's agreement (cf. design Assumption 3.1), that $minimize(\cdot)$ implementations only engage in trial parallelism through the assigned methods of their Evaluator instance - hence the meaning of introducing it as the *single entry point* for trial parallelism. To simplify implementation, there is for instance no ”ConcurrencyClerk” (mirroring BlackBoxBudgeteer) in place to ensure that renegade algorithms won't just disregard user settings and fire up their own thread pool with e.g. 10 million evaluations in parallel.

Assuming that the user wants to run with a more modest maximum of e.g. 3 trials in parallel, all they have to do is to set a Param:

```
df.howTo(
  minimize (foo),
  subjectTo (bar),
  forTrials (baz),
  withOptions ("trial parallelism" -> 3)
).show()
```

Or equivalently use the $setTrialParallelism(\cdot)$ setter with the Transformer API. Based on this information, $solve(\cdot)$ will then instantiate a *StaticEvaluator*, parameterized with the desired level of parallelism, *par*:

```
class StaticEvaluator(/* etc.*/) extends Evaluator(/* etc.*/) {
  override def parallelism: Int = par
  protected val executionContext =
    threads.getExecutionContext(par)
}
```

When some algorithm asks for the designated level of *parallelism*, the Evaluator will then say 3, and all points sent to the Evaluator will then be evaluated in a thread pool of size 3. For instance, on each iteration in the main loop of BayesianOptimizer, 3 trials will thus be executed in parallel, as per user specification. Had the user instead specified a level of 1, then we would have used a special "dummy" execution context to run evaluations in the same thread as the caller, without introducing significant overhead.

Note however, that the ability to exploit parallelism depends highly on the design of the individual sBBO algorithm. SBO algorithms are arbitrarily flexible in this regard, since we can in principle just ask them to propose several points in sequence, using interpolated objective values for points we have already decided to evaluate in subsequent adaptive sampling phases. More evaluations is however not necessarily better - directional algorithms working opportunistically (cf. section 2.3.3) are designed to avoid doing needless evaluations when a descent direction has already been found in an iteration: evaluating all pattern points, which is expensive in the general case, is therefore not necessarily a desirable algorithm design, even when a level of parallelism is possible.

It is therefore ultimately up to the individual implementation of $minimize(\cdot)$ to make the best use of the assigned "parallelism budget", even if this means always running sequentially, regardless of user input - perhaps with a friendly warning message, when parallelism cannot be not fully leveraged for algorithmic reasons.

Dynamic trial parallelism

Moving on, the library also (experimentally) supports a *dynamic* kind of trial parallelism, to be used with flexibly concurrent algorithms like SBO - that is, instead of the thread pool size during a run being set in stone by an initial *static* user setting, it can change *dynamically* on runtime, according to load characteristics tracked over time. When the user does like this:

```
df.howTo(
  minimize (foo),
  subjectTo (bar),
  forTrials (baz),
  withOptions ("trial parallelism" -> 3,
               "dynamic trial parallelism" -> true)
).show()
```

3 will then be regarded as an "initial guess" on the right thread pool size, with the best setting to be discovered on runtime. We use an underlying new load balancing algorithm for this.

To preface what is to come, I know nothing about established methods for dynamic load balancing. They were never a course topic during my stay at AAU. But I knew something about sBBO, and when all you have is sBBO, everything looks like an experiment. I mean no insult by disregarding existing solutions within any proud and storied research field here, and the project was simply reaching a point where more desktop research would be a problematic time investment.

Upon detecting user settings like in the previous example, $solve(\cdot)$ will instantiate a DynamicEvaluator instead of a StaticEvaluator. The former class offers the same interface for $minimize(\cdot)$ as any other Evaluator, yet the size of the thread pool, and thus the value of *eval.parallelism*, varies during optimization. The informal intuition (and nothing more) behind the proposed load balancing algorithm is to solve the following optimization problem:

$$\underset{p \in \mathbb{N} \setminus \{0\}}{\text{maximize}} \quad \text{throughput}(p) \tag{3.8}$$

Where p (for parallelism) is the thread pool size, and $throughput : \mathbb{N} \to \mathbb{N}$ is the number of evaluations per second when using a thread pool size of p during the run.

There is however more to this problem than it seems: We are solving it with sBBO, and in doing so additionally have a *Hidden*, *Unrelaxable* constraint when trialing solution candidates (cf. section 2.4.1): To avoid cluster machines crashing from e.g. running out of memory. That is, trying out $p = 10,000,000$ with an expensive query may violate this constraint on a small cluster.

Another issue not captured in the problem model above is that the objective function is *time-dependent*: The throughput achieved by running with some level of parallelism is likely to change over time with a number of external factors. The optimal value of p might therefore change over time. We can say that we actually need to solve *several* optimization problems, each representing a point in time, where we need to decide on the best value of p until the next point in time. Formalizing dynamic load balancing as a bog-standard optimization problem is maybe low value for effort.

The proposed approach ultimately instead takes its *onset* in directional sBBO, specifically Compass Search that we are already familiar with (cf. section 2.3.3). Among such algorithms, Compass Search tends to fall short whenever line searches in cardinal directions fit the contours of the objective function poorly [42]. This is however a non-issue when our search space is one-dimensional, since cardinal search is the only option.

Leaving some practicalities for later, the core load balancing approach can be summarized by algorithm 3.

Algorithm 3 Directional Dynamic Load Balancing

1: **procedure** DDLB($speedup_{threshold} = 1.1$, $parallelism = 1$, $decay = 0.99$)
2: $throughput_{current}$, $direction$, $stepsize \leftarrow 0,\ 1,\ 0$
3: **while** true **do**
4: $linesearch \leftarrow \max(1,\ parallelism + direction \cdot stepsize)$
5: $throughput_{new} \leftarrow throughput(linesearch)$ ▷ Estimate throughput
6: **if** $speedup(throughput_{new},\ throughput_{current}) \geq speedup_{threshold}$ **then**
7: $parallelism,\ throughput_{current} \leftarrow linesearch,\ throughput_{new}$
8: $stepsize \leftarrow stepsize + 1$ ▷ Increase momentum
9: **else**
10: $direction \leftarrow -1 \cdot direction$ ▷ Switch poll direction
11: $stepsize \leftarrow 1$ ▷ Reset momentum
12: $throughput_{current} \leftarrow decay \cdot throughput_{current}$ ▷ Devalue estimate

So, line 1 introduces three hyperparameters with example defaults: a threshold $speedup_{threshold}$ for accepting new incumbents based on speedup (to be explained), an initial solution *parallelism* (thread pool size), along with a *decay* factor used for ensuring that the algorithm won't get stuck on outdated throughput estimates indefinitely. The algorithm conceptually works in an infinite loop (line 3), estimating the obtained speedup from increasing or decreasing the current level of *parallelism* through line search. Line 4 betrays the algorithm's directional heritage, as a new candidate level of parallelism is decided by a line search from the current incumbent. On line 5, we devise a way to estimate the throughput of running with the new level of parallelism (to be explained). Then we update the state of the search strategy based on a measure of speedup. As mentioned, our notion of throughput within our sBBO setting is:

$$throughput = \frac{|trials\ completed|}{time} \tag{3.9}$$

That is, the number of objective evaluations completed per time unit, running with some thread pool size. $speedup(\cdot)$ in algorithm 3 is then calculated by:

$$speedup(throughput_{new},\ throughput_{current}) = \begin{cases} \infty & \text{if } throughput_{current} = 0 \\ \frac{throughput_{new}}{throughput_{current}} & \text{otherwise} \end{cases} \tag{3.10}$$

In practice, the first case is only used for getting a throughput estimate for the initial value of *parallelism*. Otherwise the notion of speedup simply pertains to whether we can expect to get more trials out the door per time unit with one thread pool size or the other.

One line 6 of algorithm 3 we use a speedup threshold to decide whether any neighboring *parallelism* level is deemed good enough to update incumbent information. This is used to make the algorithm more consistent in the face of insignificant performance fluctuations, and the suggested default threshold of 1.1x speedup is chosen to be quite conservative. In the event that we achieve a sufficient level of speedup, incumbent information is updated, and the step size is increased (lines 7-8) - upon the next iteration, line search then continues opportunistically in the ascent direction, with increased momentum. In the event that the increase in speedup is insufficient, e.g. performance has plateaued, we then reset momentum and switch poll direction to assess whether we went too far in the wrong direction (lines 10-11). On line 12, we account for the fact that current throughput estimates may no longer reflect reality, and therefore use a simple *decay* factor to naturally devalue old throughput estimates over time - after e.g. ten unsuccessful polls, the current estimate will be $0.99^10 = 0.9044$ times less valuable when assessing speedup.

A fair criticism of algorithm 3 is that thread pool size is only decreased upon *speedup improvements*, as might occur when the sheer number of concurrent tasks becomes burdensome to the cluster. With increasing momentum, the algorithm might hence converge to a pool size much larger than actually beneficial by overshooting the step size. One might consider dropping the momentum mechanism from the algorithm altogether. Nonetheless, the algorithm was kept as is, since the optimum *parallelism* value for running several Spark queries in parallel was expected to be pretty tame in practice, thus limiting the expected momentum buildup. Also, I had already started running experiments upon realizing this, so there's that.

Implementation-wise, algorithm 3 is incorporated into DynamicEvaluator, by *unrolling* its entire loop update and incorporating it into each objective evaluation:

```
override def objective(pts: Array[Array[Double]]):
Array[Option[Trial]] = {
  startTimer()
  val res = super.objective(pts)
  stopTimer()
  optimize(res.count(_.isDefined)) // Do unrolled update here.
  res
}
```

The implementation thus evaluates and conveys Trials for $minimize(\cdot)$ as normal, but then also times them (with *System.nanoTime*) and does an internal algorithmic update, based on the number of completed evaluations, ultimately used for estimating throughput by the given definition (cf. eq. (3.9). The skeleton of $optimize(\cdot)$ looks like this:

```
private def optimize(numCompleted: Int): Unit = {
```

```
2   numEvals += numCompleted
3
4   // Poll complete:
5   if (numEvals >= lineSearch) {
6     val newThroughput = numEvals / getAndResetDuration()
7
8     // Update based on speedup etc.
9   }
10 }
```

To explain, we *estimate* the throughput of running with a particular thread pool size by using a stopwatch ($getAndResetDuration(\cdot)$), and waiting until at least one evaluation per thread in the current thread pool size (*lineSearch*) has completed - intuitively speaking, giving the polled thread pool size a chance of demonstrating perfect linear speedup with respect to a sequential run. When its time for an update, we reset the stopwatch (line 6), and update the direction, step size, and/or incumbent information, including the thread pool size - rinse and repeat.

This approach of course has significant limitations: It is only meant to work with algorithms in the library using *eval.parallelism* to decide on what level of parallelism to run with at regular intervals, i.e. SBO algorithms as it stands. As also mentioned with static trial parallelism, individual algorithmic design largely decides how parallelism can be leveraged within sBBO.

Another issue is the peculiar fact that we don't actually ever set the thread pool size to be the optimal one, with convergence being characterized by a sequence of polls of immediate neighbors, e.g. "4, 6, 4, 6,..." for an optimum number of 5 threads. This stems from the fact that the algorithm always needs to assess whether current settings can be improved through line searches, and that this only possible in practice though direct experiment with different thread pool settings.

An entirely different matter is how this implementation handles the Unrelaxable, Hidden constraint that we should not make cluster machines crash when trying to maximize throughput. Current handling largely relies on algorithmic design and domain assumptions to enforce this *implicitly*. That is, we immediately switch poll direction upon plateauing to adjust for overshooting the step size, and assume that the optimum number of concurrent threads in our expensive query setting is relatively low, thus limiting how much momentum leads to overshooting, and how pressured a driver machine with current hardware will likely be for e.g. available memory.

More *explicit* handling might have involved keeping track of driver JVM memory peaks during execution to set limits, or asking the user to specify a maximum number of threads allowed, as a safeguard. Such approaches however don't take executor machine loads into account, which is an equally relevant but much more difficult factor to account for - the very motivation behind the proposed approach to dynamic load balancing is that it is difficult to reason about the right number of

threads to run with in practice beyond doing black-box experiments.

3.6.2 Solve parallelism

Not much was initially said about BlackBoxOptimizer's $transform(\cdot)$ implementation in section 3.4.2, since its main job is to route to one or more $solve(\cdot)$ calls, making up different subproblems in the optimization query, to be solved more or less in parallel, as preferred. Apart from the default option of running with a single $solve(\cdot)$ call, DIBBOlib supports running the same algorithm with different seeds, as well as search space partitioned runs, depending on user preference.

Seed parallelism is discussed here to show how the basic idea is implemented, while the latter option is discussed in its own section.

Seed parallelism

The user can configure seed parallelism by a Param specifying the number of different seeds to use when running:

```
df.howTo(
  minimize (foo),
  subjectTo (bar),
  forTrials (baz),
  withOptions ("seed parallelism" -> 5)
).show()
```

Due to the common need for stochastic behavior as well as repeatability when running sBBO, BlackBoxOptimizer aggregates a global random seed Param. Upon spotting that this option is chosen, BlackBoxOptimizer's $transform(\cdot)$ will use this progenitor seed to generate 5 new random seeds, split the total trial budget evenly among them, and relay the seed-budget pair to different $solve(\cdot)$ calls in a thread pool:

```
1  val seeds = // Generate from progenitor seed.
2
3  // Split total budget evenly:
4  val budgets = binnedSplit(totalBudget, seeds.length)
5
6  // Solve in thread pool:
7  val threadPool = threads.getExecutionContext(seeds.length)
8  /* etc. */
9
10 // Solve, and collect the best feasible solution:
11 futures.map(threads.awaitResult(_, Duration.Inf))
12   .flatten.minByOption(_.rawObjective)
```

For k seeds and a total trial budget of T, the $binnedSplit(\cdot)$ on line 4 will allocate at least $\lfloor \frac{T}{k} \rfloor$ for all subproblems, with one extra trial for T mod k subproblems,

thereby forming evenly sized bins. As shown on line 11-12, the library will, as always, pick the best solution found among all $solve(\cdot)$ calls as the candidate for projection unto the output DataFrame of the optimization query. Note here that the objective closure makes sure that only the best *feasible* solution found is ever returned from $solve(\cdot)$, if any.

$solve(\cdot)$ is designed to work indifferently to its sibling calls, with $transform(\cdot)$ relaying the necessary synchronization primitives, such as the global Trial ID counter. One interesting subtlety pertains to the reload of TrialHistory from the checkpoint path on each run closure call (cf. section 3.5.5): Algorithms in separate $solve(\cdot)$ calls are free to leverage any information about the search space found in the optimization process at large, consistent to their current measure of penalty, of course. This is especially useful for building better surrogates with Model-based methods. It should however be noted that one's mileage might vary, especially if Local algorithms pile unto the same best solution found across all subproblems, and thus might produce very similar solutions in the end, depending on the impact of running with different seeds for individual algorithms.

Another negative side is that repeatability of results is not guaranteed when running checkpointing with this kind of parallelism, since available TrialHistory information on each call to $minimize(\cdot)$ depends on what happened to be reloaded before the next run. Still, I preferred not artificially hiding any search space information from optimization algorithms in the end.

As already mentioned (cf. section 3.2.2), I am a bit skeptical about the usefulness of seed parallelism under a limited evaluation budget, but it is nonetheless offered as an option, due to the ease of supporting it, and the groundwork it provided for supporting more interesting kinds of subproblems.

3.7 Search space partitioning

Existing sBBO research has looked into the merits of partitioning the search space of problems, with good results found with various problems and algorithms [191, 194]. The reasons given for such procedures that I've stumbled upon might be appealing, but are also purely *heuristic* in nature. For instance, that doing local search in several search space partitions leads to better exploration of the global search space overall [191].

There is arguably a good reason behind the general-case motivation being purely heuristic: Such partitioning strategies universally improving the performance of any algorithm would contradict the No Free Lunch theorem with respect to optimization: That if nothing can be assumed about problem structure, the average performance of each algorithm is the same across all possible problems [197]. Any increased performance on some problems with respect to the mean is paid for in kind by a corresponding decrease on others [198].

Imagine for instance that we are using separate instances of a brute force strategy blindly enumerating design points, in different search space partitions with a limited evaluation budget for each. Performance would depend entirely on which solutions each instance of the algorithm stumbles upon before running out of trials, and is therefore ultimately a function of the problem-algorithm pair.

The practical consequence of this is that search space partitioning is only offered with DIBBOlib as a *heuristic strategy*, to be used at the discretion of the user, when it "makes sense" for the problem-algorithm pair. Note that this doesn't necessarily require expert knowledge from users, but may instead be facilitated by simple trial-and-error, as already exercised within e.g. ML, which operates under its own No Free Lunch Theorem [78].

For instance, a user might find that MADS, our resident directional sBBO algorithm, converges really fast to a feasible solution on a particular problem, which is great. The search space is however quite large, so this seems a bit fishy. Given that we have a decent number of trials to spare, starting several instances of MADS in their own local subregion might therefore "make sense" in this scenario, to improve global exploration.

The library offers static (manual) as well as dynamic (automatic) search space partitioning, to be explained in turn.

3.7.1 Static partitioning and local SearchSpaces

Consider the case when one dimension is much larger than the other. Splitting it in half might form a reasonable search heuristic for a problem in question. The user can specify such splits manually, with a Map denoting how much many pieces each dimension is to be split into:

```
1 df.howTo(
2   minimize (foo),
3   subjectTo (-1000000.0 <= hcol("x") <= 1000000.0,
4              -100.0 <= hcol("y") <= 100.0),
5   forTrials (baz),
6   withOptions ("partitioning keys" -> Map("x" -> 2))
7 ).show()
```

In this example, we specify the desired split of variable "x" line 6. We thereby end up with two search space partitions, one where $-1000000.0 <= hcol("x") <= 0.0$, and one where $0.0 <= hcol("x") <= 1000000.0$, with variable "y" having its original bounds in each. We have simply split the rectangle making up the bound-constrained search space into two equally large pieces, only overlapping on $x = 0$ to ensure a continuous global range. Had the user instead specified $"y" -> 2$, then we would have ended up with as many pieces as the *product* of splits, i.e. $2 \cdot 2 = 4$ equally large pieces of the original search space.

Discrete variable ranges, i.e. Integral and Categorical variables, are instead split by *binning* values in their range evenly, similarly to the procedure used for splitting the trial budget from earlier. Splitting $hcol("z")inSeq("a","b","c")$ in half, we obtain $hcol("z")inSeq("a","b")$ and $hcol("z")inSeq("c")$ in separate search space partitions.

Completely analogously to the seed parallelism case, BlackBoxOptimizer's $transform(\cdot)$ will split the original SearchSpace and specified trial budget in half, and relay the different local SearchSpaces to separate $solve(\cdot)$ calls running more or less in parallel. While the default is to run with one thread per partition, the user may optionally specify a maximum size of the thread pool with an additional Param, including 1, if partitioning is all they care about:

```
df.howTo(
  minimize (foo),
  subjectTo (-1000000.0 <= hcol("x") <= 1000000.0,
             -100.0 <= hcol("y") <= 100.0),
  forTrials (baz),
  withOptions ("partitioning keys" -> Map("x" -> 2),
               "partitioning parallelism" -> 1) // Sequential
).show()
```

If the user specifies an initial solution, it will only be relayed to local SearchSpaces containing it. Generating the local SearchSpace when having splits specified amounts to calculating the Cartesian product among all dimensional splits, which is an $\mathcal{O}(2^n)$ operation. We don't worry too much about having large values of n in our setting, however,

Subproblem management in $solve(\cdot)$ gets a bit tricky with search space partitioning in the mix. There are two main issues: One or more discrete variables may have been split into bins containing only one value (happened to "z" in the previous example), and thus effectively be *constants* in some subproblems, not to be subjected to optimization. Furthermore, consider the usage of vertical transfer: How can algorithms interface with only the Trials relevant to their *local* SearchSpace, keeping in mind Model-based approaches like SBO where a reduced interpolation set might make surrogate model building impossible?

To make a long story short, the solution was to introduce a *layer of indirection* for SearchSpaces and TrialHistory both throughout the entire library, providing a *local* view by default to algorithms, with the possibility of accessing a *global* one on demand. Upon trial evaluation, our handy objective closure can make sure to project constant dimensions in and out with respect to the calling algorithm as needed, as if nothing happened. SBO algorithms can opt for interpolating the entire SearchSpace, leveraging all available information, while only doing optimization within their local region, as exemplified here:

```
// Surrogate interpolates the entire search space:
```

```
val gp = new GaussianProcess(ss.global)
gp.addPoints(X, fX)

// Adaptive sampling only concerns the (default) local one:
val bo = new BayesianOptimization(ss, gp, seed, /* etc. */)
```

3.7.2 Dynamic partitioning and perimetric honeycombs

The library also offers facilities for dynamic partitioning of the search space. Here, the user simply specifies the number of desired partitions, and the library then decides where to split on runtime, i.e. dynamically. So, they might say:

```
df.howTo(
  minimize (foo),
  subjectTo (bar),
  forTrials (baz),
  withOptions ("partitioning factor" -> 5,
               "partitioning metric" -> "range")
).show()
```

Or something to that effect with the alternative Transformer API. This query specifies that the user would like to split the search space into 5 subregions, and that splits should be decided based on the *range* of variable bounds, i.e. with a preference for splitting the longest dimensions - there are of course other options, to be described in time. Beyond the standard handling for partitioning already described for the static case, the library now has to decide on which split is the ”best” given this specification, which is what we will cover in this section.

As the reader may have noticed, the library approach to partitioning the search space is based on splitting a hyperrectangle into a number of evenly sized smaller hyperrectangles filling out the entire space with minimal overlap - it is not the only possible way to do this kind of partitioning, but unlike alternative approaches found in research (e.g. [194, 191]), I find that it is relatively simple to reason about for heuristic purposes. It also fits our design of always having bounds information available for variables.

The technical name for our desired geometric result is variously called a *tiling*, *tessellation* or *honeycomb*, with the latter term being the preferred one for higher-dimensional spaces [79]. Yet the user requirement of a particular number of partitions in the results, and the need to accommodate an arbitrary number of dimensions, make our problem a bit more involved than the vanilla honeycombing problem. The fact that discrete variables in our setting can only be split to a limited degree is another complication that we need to handle. Such issues motivated the homebrew to follow.

We ignore the complication of discrete variables and finite splittability for now. I ultimately found that the most useful way to think about our special geometric

problem was in terms of *integer factorization*. Let's say that the user's desired number of splits is an integer $k > 0$, that we have $n > 0$ variables in the problem, and that each of these dimensions has some positive measure of magnitude, e.g. their bounded range, in an n-vector $d = [d_1,\ d_2, ...,\ d_n] \in \mathbb{R}^n_{>0}$.

Our initially stated goal is to formulate a product of n positive integers equal to k. Consider for instance when $n = 3$ and $k = 4$. Possible ways to split this 3D search space into four space-filling partitions include splitting two dimensions into two ($1 \cdot 2 \cdot 2 = 4$) or splitting one dimension into four ($1 \cdot 1 \cdot 4 = 4$) (cf. fig. 3.3).

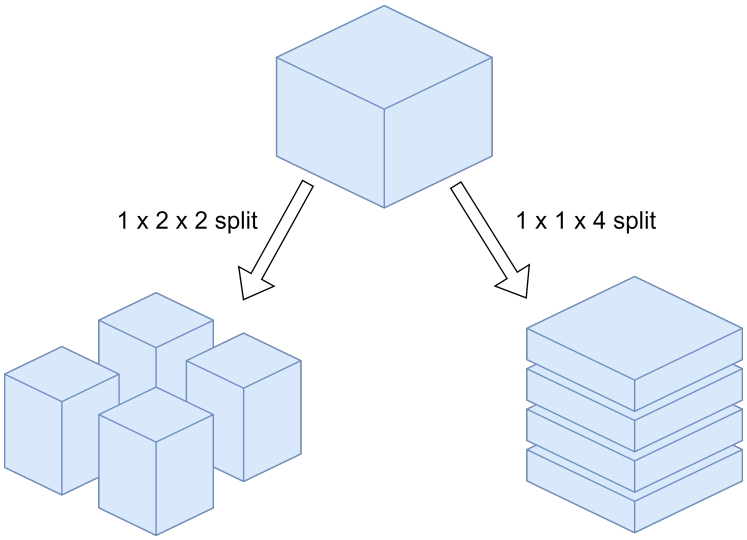

Figure 3.3: Two possible ways to split a 3D box, by splitting 2 dimensions into 2, or 1 dimension into 4. The order of factors is not significant in this example.

This factorization requirement is basically only a *constraint* of an optimization problem that we are about to formulate: There is no reason to prefer one split over another. As for our objective, if all we care about is minimizing the *volume* of each result search space partition, a reasonable initial proposal, we need to minimize the *product* of dimensional magnitudes in d with splits applied to them, leading to the following non-linear integer optimization problem:

$$\begin{aligned} \underset{x \in \mathbb{N}^n_{>0}}{\text{minimize}} \quad & \prod_{i=1}^{n} \frac{d_i}{x_i} \\ \text{subject to} \quad & \prod_{i=1}^{n} x_i = k \end{aligned} \tag{3.11}$$

For instance, one way of splitting $d = [1,\ 2,\ 4]$ with $k = 4$ to obtain a minimum volume is simply to split the longest dimension into four, and thus obtain the partition volume of $1 \cdot 2 \cdot \frac{4}{4} = 2$. Notice however that it verily *doesn't matter* from a minimization standpoint whether we do a $1 \cdot 2 \cdot 2$ or a $1 \cdot 1 \cdot 4$ split, or which

dimensions we spend our "budget" of k splits on, due to the associativity of multiplication! To demonstrate: $1 \cdot 2 \cdot \frac{4}{4} = \frac{1}{2} \cdot 2 \cdot \frac{4}{2} = \frac{1}{2} \cdot \frac{1}{2} \cdot (1 \cdot 2 \cdot 4) = \frac{1}{4} \cdot (1 \cdot 2 \cdot 4) = 2$.

Furthermore, even though the volume is technically minimal, a volumetric approach puts no preference on splitting the *longest* dimensions, which would arguably make heuristic sense. A solution of e.g. $x = [4,\ 1,\ 1]$ in our example feels off despite being minimum-volume, since the relative size of dimensions is now even more disproportionate, resulting in a very narrow rectangular slice of the search space.

Instead, we base the proposed approach on minimizing the resulting partition *perimeter*. Mathematicians might now exhale loudly and finally throw this report in the bin, since perimeters as a geometric concept admittedly make less and less sense with an increasing number of dimensions. There is no such thing as "n-perimeters", strictly speaking. Yet on our own terms, if we try to generalize the notion of calculating the sum of all sides of an n-dimensional rectangle with n-vector of side lengths d, we get the following formula:

$$\text{"}Perimeter\text{"}(d) = 2^{n-1} \sum_{i=1}^{n} d_i \tag{3.12}$$

Since each side repeats 2^{n-1} times in an n-rectangle. From an optimization standpoint we however don't care about this scale factor. For simplicity, and since "n-dimensional perimeters" is our own made-up concept to begin with, we will henceforth use our creative license to ignore it when talking about perimeters, since the same conclusions apply regardless. In the following alternate problem we instead just minimize the sum of sides with splits applied to them in a new perimetric objective:

$$\begin{aligned} \underset{x \in \mathbb{N}^n_{>0}}{\text{minimize}} \quad & \sum_{i=1}^{n} \frac{d_i}{x_i} \\ \text{subject to} \quad & \prod_{i=1}^{n} x_i = k \end{aligned} \tag{3.13}$$

Unlike in eq. (3.11), some feasible solutions are now better than others. In our running example, the minimum perimeter of 4 can be obtained from e.g. $x = [1,\ 2,\ 2]$. This is better than e.g. $x = [4,\ 1,\ 1]$ which would result in a perimeter of $\frac{1}{4} + 2 + 4 = 6.25$. This objective on one hand prefers splitting longer dimension, for larger perimetric reductions, while at the same time following a *law of diminishing returns* when "allocating" splits: E.g. setting $x_3 = 2$ for $d_3 = 4$ reduces its contribution to the total perimeter by 2 compared to setting $x_3 = 1$, while setting $x_3 = 4$ only reduces d_3's contribution by 1 compared to setting $x_3 = 2$. Under the present constraint regarding k, the objective will thus generate search space partitions of relatively even lengths, to spend its "budget" of k splits more

efficiently. Note that we accomplish all of this on top of still getting minimum-volume partitions, due to the previously mentioned associativity rule.

Solving the problem of eq. (3.13) is harder than it may seem, mainly due to its constraint requiring a special case of integer factorization, where we are only helped by having a known number of factors to be found. The balancing act of handling factorization constraints and diminishing returns makes it if not impossible, then at the very least too hard for me at present to come up with a greedy algorithm able to make the best global choices locally.

Still, disregarding the presence of discrete dimensions, the problem can be solved in polynomial time with dynamic programming, using the following recurrence:

$$p(i,\ j) = \begin{cases} d_1 \div j & \text{if } i = 1 \\ \min\{d_i \div s + p(i-1,\ j \div s)\ :\ s \mid j\} & \text{otherwise} \end{cases} \tag{3.14}$$

Where $s \mid j$ means that s (for "split") is a positive integer divisor of j. We assert that $p(i,\ j)$ is the minimum perimeter possible when splitting dimensions $[1..i]$ of dimensional magnitudes $d = [d_1,\ d_2, ...\ d_i] \in \mathbb{R}^i_{>0}$ by corresponding factors $[x_1, ...,\ x_i] \in \mathbb{N}^i_{>0}$ such that the product of factors $x_1 \cdot ... \cdot x_i = j > 0$. In other words, we assert that $p(i,\ j)$ is the objective value of a solution to the problem defined by eq. (3.13) with $i = n$ and $j = k$.

We prove this assertion by induction on i. Since j is only *non-increasing* in its quotient-based recursion (think of $j = 1$), we will address this variable by arguing that each proposition in the proof holds for all $j > 0$. Let a and b be integers, so we don't have to remind ourselves constantly. In the base case, we prove that the proposition holds for $p(1,\ b)$ for all $b > 0$. In the inductive step, we prove that for all $a > 1$, if the proposition holds for $p(a-1,\ b)$ for all $b > 0$, then it holds for $p(a,\ b)$ and all $b > 0$. This way, propositional dominoes fall such that the original assertion holds for all integers $a, b > 0$.

- **Base case**: When $a = 1$, we only need to consider the base case of eq. (3.14) and prove that it gives the right result for all $b > 0$. For $a = n = 1$ and $b = k > 0$ in eq. (3.13), the only feasible solution such that $x_1 = k$ is $x = [k]$ with objective value $d_1 \div k$. Since correspondingly $p(1,\ b) = d_1 \div b$ for all integers $b > 0$, the proposition holds for $a = 1$ and all $b > 0$.

- **Inductive step**: We can focus our efforts on the recursive case of eq. (3.14) here. For some $a > 1$, suppose that $p(a-1,\ b)$ gives the correct result for all $b > 0$. Is this then also the case for $p(a,\ b)$ for all $b > 0$?

  For $p(a,\ b)$ when $a > 1$ and $b > 0$, there will be one or more elements in the set over all divisors of $j = b$ in the recursive case of eq. (3.14), corresponding

to every possible split of d_i by s such that $s \mid j$, with at least one element for $s = 1$. As a shorthand, we will refer to this non-empty set as the *split set*, noting that it by definition contains one element for *every divisor* of j.

For $a = n > 1$ $b = k > 0$ in eq. (3.13), the solution is an n-vector x such that $x_1 \cdot ... \cdot x_n = k$ and the split perimeter $d_1 \div x_1 \ + ... + \ d_n \div x_n$ is minimal. Here, we note that *each component of x must necessarily be a divisor of k*.

For $a = i > 1,\ b = j > 0$ in eq. (3.14), we can use our inductive hypothesis for each element in the split set, corresponding to some divisor s of j, to assume that its corresponding $p(i-1,\ j \div s)$ term equals the minimum split perimeter $p^s_{i-1} = d_1 \div x_1 \ + ... + \ d_{i-1} \div x_{i-1}$, such that $x_1 \cdot ... \cdot x_{i-1} = j \div s$. Each element of the split set, of the form $d_i \div s + p^s_{i-1}$, therefore equals the minimum split perimeter of dimensions $[1..i]$ such that $x_1 \cdot ... \cdot x_{i-1} \cdot s = j$.

Note from before that the split set by definition exhausts every divisor of j, and that all solution components must be divisors of k in eq. (3.13). The minimum of the split set in eq. (3.14) is therefore the minimum split perimeter for dimensions $[1..i]$ among all possible ways of splitting dimension i - in other words being the minimum objective value of a solution to eq. (3.13) where $a = i = n$, $b = j = k$. Under the assumption that $p(a-1,\ b)$ gives the correct result for some $a > 1$ and all $b > 0$, we have thus proved that this is also the case for $p(a,\ b)$ for all $b > 0$.

Hence we have proved that the original assertion holds for all integers $a, b > 0$. □

We will however sometimes not find the *intuitively* best possible perimeter from $p(n,\ k)$ though. Consider the initial example given in this section, where the user demands a dynamic split into 5 local SearchSpaces. Note that 5 is a *prime number*. Then consider an example where $d = [1,\ 1,\ 1]$. The only feasible way to factorize these magnitudes with respect to eq. (3.13) is to split one dimension into five pieces, thus making $1 + 1 + \frac{1}{5} = 2.2$ the minimum possible perimeter, Note that we could have actually obtained a smaller perimeter with $k = 4$, using a solution where $1 + \frac{1}{2} + \frac{1}{2} = 2$. This is due to the previously mentioned law of diminishing returns. Prime number splits and the like are generally problematic for obtaining partitions with evenly sized dimensions. If we want to commit to minimizing search space perimeters, then we want to use an *inequality constraint* for optimization instead:

$$\begin{aligned} \underset{x \in \mathbb{N}^n_{>0}}{\text{minimize}} \quad & \sum_{i=1}^{n} \frac{d_i}{x_i} \\ \text{subject to} \quad & \prod_{i=1}^{n} x_i \leq k \end{aligned} \tag{3.15}$$

Note that while the problem now looks a bit more like a Knapsack problem than before, 1) our constraint still concerns a *product* of factors, which is a significant

departure, and 2) knowing that all components of x must be positive (i.e. "all items must go in the knapsack") eliminates a key element of choice, making the problem a bit easier.

We use this alternative problem formulation to first obtain an initial partitioning of the global search space into a number of space-filling, non-overlapping, rectangles with identical dimensions, i.e. a bog-standard rectangular honeycomb. We can still use dynamic programming with the recurrence above, but just need to look for the best possible value for j when $i = n$ in our lookup tables, knowing from our proof that $p(n,\ j)$ is indeed the minimum split perimeter for any assignment of $j > 0$.

The initial split by dynamic programming may or may not result in us having leftover partitions available, which we provide by filling in a number of *subdimensional slices* (to be explained). This approach implicates that we sometimes won't obtain minimum-volume partitions with the same split factors along all dimensions. However, we will generally come very close (as we shall see), and this solution was ultimately deemed better than making users google what a "highly composite number" is.

We start out with how the initial honeycomb split is found. Confer algorithm 4. Using two $n \times k$ matrices (lines 3-4) for memorizing $p(i,\ j)$ and optimal s's in all subproblems respectively, the problem can be solved in $\mathcal{O}(n \cdot k^2)$ time by filling these matrices in three embedded for loops, iterating over possible divisor s's with modulus checks in the innermost one (line 12). We need to iterate over all possible divisors of all possible k's to be able to find the best total split factor at $i = n$ in the end. The optimal splits for all dimensions are found in linear time by going in reverse from the split matrix component corresponding to the minimum perimeter when $i = n$ in the perimeter matrix (lines 18-22). While asymptotic time and space complexity might seem a bit hefty, note that we neither expect k or n to be particularly large in our setting, for each their own reason.

We had the special predicament that discrete variable dimensions are only finitely splittable in our application - e.g., a Categorical variable with tree categories in it can at most be split by a factor three. It is however simple to extend algorithm 4 to handle this: On line 7 and 13, we just use extra checks to ensure that we won't recur to undefined splits or breach the upper limit for splittability for any Variable type, with no upper limit for Real variables, and limits decided by finite cardinality for others. Since it is always at least possible to fall back on using ones in our solution vector, we won't get into trouble from having *None* values in the *splits* table, and there will always be a solution split ready at the end.

The result of running algorithm 4 is a vector of factors $[x_1\ ,...,\ x_n]$. These specify a perimetrically optimal and isometric split of all dimensions. In the event that the product of these factors equals k, specifying a complete isometric split of the search space according to user specification, we can just calculate and cross all

Algorithm 4 Min-Perimetric Up-to-k n-Honeycomb

```
1:  procedure MPUkNH(dims, k)
2:      n ← dims.length
3:      perims[1..n, 1..k] ← ∞                          ▷ Tabulates p(i, j) of eq. (3.14)
4:      splits[1..n, 1..k] ← None                       ▷ Tabulates optimum s for p(i, j)
5:
6:      for i = 1 to k do                               ▷ Handles base case
7:          perims[1, i] ← dims[1] ÷ i
8:          splits[1, i] ← i
9:
10:     for i = 2 to n do                               ▷ Handles recursive case progressively
11:         for j = 1 to k do
12:             for s = 1 to j if s | j then do         ▷ Tries out possible divisors
13:                 p = dims[i] ÷ s + perims[i − 1, j ÷ s]
14:                 if p < perims[i, j] then
15:                     perims[i, j] ← p
16:                     splits[i, j] ← s
17:
18:     j ← arg min_{i ∈ [1..k]} (perims[n, i])         ▷ Finds best isometric split ≤ k
19:     sol[1..n] ← None
20:     for i = n down to 1 do                          ▷ Reconstructs solution by backtracking
21:         sol[i] ← splits[i, j]
22:         j ← j ÷ sol[i]
23:
24:     return sol
```

splits as usual, and we are done.

For the case when we still need to allocate $\kappa > 0$ splits, we can now follow a greedy approach, since we only need to make a greedy choice once, i.e. with no possible backtracking (to be explained). To explain the greedy choice first, consider a vector of magnitudes $d = [d_1,\ d_2, ...,\ d_n] \in \mathbb{R}^n_{>0}$ and factors $x = [x_1, ...,\ x_n] \in \mathbb{N}^n_{>0}$. Then, taking diminishing returns into account, the dimension eliciting the maximum possible perimetric decrease by increasing its corresponding split factor further by one is:

$$\underset{i \in [i..n]}{\arg\max} \left(\frac{d_i}{x_i} - \frac{d_i}{x_i + 1} \right) \tag{3.16}$$

That is, we just assess how much each magnitude is affected by a further split, and pick the dimension with the most bang for the buck.

To complete partitioning, we can as mentioned make such a greedy choice *just once*, finding the dimension with the maximum further perimetric decrease possible, if such a dimension exists, regarding the finite splittability of certain Variable types. We add the remainder partitions of the search space by making further splits along this dimension, yet only to one or more of its *sub-dimensional slices*. This idea is probably best to visualize first, before further details are given. Confer fig. 3.4. If we for instance already have a 2×2 split in 2 dimensions at this point, but need to fill in one more partition for a total of 5, we just cut one of the slices along one dimension a bit more more finely, obtaining an additional partition.

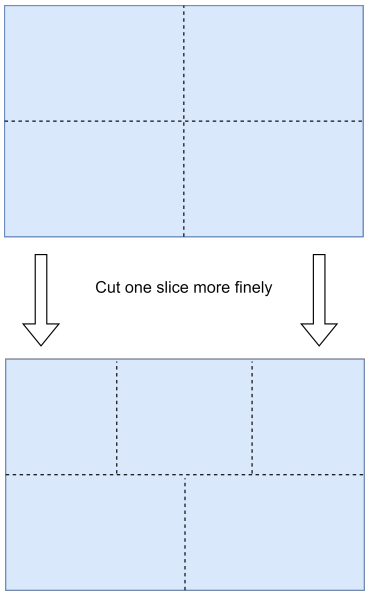

Figure 3.4: Filling in leftover partitions after an isometric split. The user wants five partitions, so we need to add one more after the initial split into four.

Note that if the originally assigned split budget were 6 in this particularly proportioned example, then the solution of algorithm 4 would have been a 2×3 split. More generally, for any factor x_i in a solution x to the problem in eq. (3.15), $x_1 \cdot ... \cdot (x_i + 1) \cdot ... \cdot x_n \leq k$ is impossible, since this would imply that the perimetric objective could be reduced by further factorization, and that the solution therefore isn't optimal.

Note that multiplication is in a way just repeated addition, i.e. $(a + 1) \cdot b = a \cdot b + b$. The effect on the total product of x by increasing a factor in it by one, is therefore to add the product of all other factors to it. Since such increases are impossible for the reasons mentioned, we know that no matter which split dimension i we pick for further partitioning, there is an upper bound on the remaining number of splits κ:

$$\kappa < \frac{\prod_{j=1}^{n} x_j}{x^i} \tag{3.17}$$

Turns out the right side of this inequality is also the number of slices of dimension i! Since we know from the above discussion that no dimension can be "fully factorized" further at this point, i.e. with all of its slices split more finely, we just need to pick *one* dimension greedily, and split it more finely in *some*, but conclusively *not all* of its subdimensional slices.

This conclusion had the practical consequence of greatly simplifying the spatial reasoning required for completing the implementation. After having made our greedy choice of fill-in dimension, we just pick κ of its slices to be cut into $x_i + 1$ pieces instead of x_i:

```
1 crossDims(vars, splits)
2   .sortBy(slice => -slice.map(_.range).sum)
3   .zipWithIndex.flatMap{ case (slice, i) =>
4       val fills = if (i < kappa) fineSplit else normalSplit
5       for (f <- fills) yield { slice :+ f }
6     }
```

Line 1 crosses all splits of all dimensions except the one where we need to fill in extra partitions. On line 2, we enforce a preference for slices which might have ended up with slightly larger perimeters than others due to binning of discrete variables (a very minor optimization, admittedly). On lines 3-5 we loop over all slices of our chosen fill-in dimension, and complete the Cartesian product on line 5, with a more or less finely cut version of the greedily chosen dimension found on line 4.

In case you are wondering if dimensions get jumbled around, variables in the obtained SearchSpaces are subsequently ordered lexicographically by name, a universally enforced library core convention, assuming that I didn't flub the job.

3.7.3 SHAPely search space partitioning

The reason for nebulously talking about generic "dimensional magnitudes" instead of just "variable ranges" above is that the library offers dynamic partitioning based on alternative metrics. By process of elimination, this must be the part where cooperative game theory and Shapley values (cf. section 2.5) come into play. Users can specify alternative metrics for heuristic search space partitioning:

```
df.howTo(
  minimize (foo),
  subjectTo (bar),
  forTrials (baz),
  withOptions ("partitioning factor" -> 5,
               "partitioning metric" -> "shap",
               "checkpoint path" -> "my/favourite/path")
).show()
```

Or something to that effect with the Transformer API. As implied, partitioning can take place based on SHAP values, leveraging what is known about the search space by vertical transfer.

So far, SHAP values have generally only been used for *explanatory* purposes within the field of ML, to measure contributions of different features to predictions on a local or global level (cf. section 2.5). Within BBO, to the best of my knowledge, only one framework and a couple of (student) projects have used SHAP values, yet again, only to *explain* the contribution of decision variables to objective values, as a pre- or postprocessing step (e.g. [101, 45]. (Not to say that SHAP values aren't extremely useful in this regard.)

Beyond merely *understanding* problems, we will propose a search heuristic for *solving* them, based on SHAP values. First, there is the motivation as to why this might be a good idea, and what the connection between SHAP values and BBO might be. Consider this example from previously:

```
df.howTo(
  minimize (foo),
  subjectTo (-1000000.0 <= hcol("x") <= 1000000.0,
             -100.0 <= hcol("y") <= 100.0),
  forTrials (baz),
  withOptions ("partitioning parallelism" -> 5,
               "partitioning metric" -> "range")
).show()
```

One dimension clearly has a larger range than the other, which invites the heuristic notion that it would be beneficial to the final result if we emphasized search here, depending on our intuitions about the problem-algorithm pair, as always. But what if the objective function is actually something like this within the black box:

$$f(x, y) = x^{-42} + y^{42} \tag{3.18}$$

In other news, variable x's influence on the objective value is actually quite insignificant compared to the one of y, regardless of their respective range. Intuitively speaking, even if the chosen algorithm is much better at finding the "right value" in a dimension when it is smaller, emphasizing search in dimension x might not be that rewarding in the end in terms of solution quality. Within an ML context, we of course also have the extreme example of e.g. decision trees ignoring some features, with said features having zero influence, and thus SHAP values of zero [103].

Global SHAP values (cf. eq. (2.46)) are used as a measure of overall *feature importance* in a predictive modelling context. Recall that each global coefficient estimates how much particular feature values contribute to predictions beyond an expectation over the domain. In our BBO context, we will use global SHAP values to estimate the *importance of choice*, with decision variables as features, and the penalized objective function as the model.

We will regard the *domain* of our model as all design points having *meaningful* objective and constraint function values, i.e. all design points feasible with respect to the subset of *unrelaxable* constraints (cf. section 2.4.1), with Relaxable constraints instead being incorporated into a penalty term. As per our library design, this just means that points need to be feasible with respect to bound and integer constraints. It is a practically motivated choice, since unlike with other constraints, we can easily generate feasible design points. As a shorthand, we will name our chosen domain for integration the *relaxable domain*.

Translating ideas from the ML field directly, our interpretation of global SHAP values within BBO is as follows: If a decision variable has a *small* global SHAP value with respect to the objective, it means that choosing one assignment or another for it in a design point generally makes little difference to the objective value and/or feasibility of said design point.

While anything goes in the black-box objective, reasons for this might for instance include that the average contribution of the variable is small, or that concrete assignments are *interchangeable*, with their generic contribution level being rolled into the domain expectation that SHAP values offset from (cf. section 2.5.2).

Note from the latter example that while a low global SHAP value doesn't necessarily imply that the decision variable is *unimportant* inside the black box, choosing one assignment or the other might still matter little to the output (penalized) objective value, which is ultimately what we care about in optimization.

As for decision variables with large global SHAP values, the same conceptual translation can be presented, yet of course with inverse conclusions.

Despite being able to capture nonlinear dynamics among coalitions of players, i.e. nonlinear decision variable interactions, SHAP values are still fundamentally

additive (it's literally in the name). This coincidentally fits our creative license over the make-believe concept of n-perimeters well. Let $\Phi = [\Phi_1, ..., \Phi_n] \in \mathbb{R}^n_{\geq 0}$ be a vector of global SHAP values (cf. eq. (2.46)) for n decision variables with respect to a penalized objective function over its relaxable domain. Then the *SHAP Perimeter (SHAPe)* of the search space is:

$$SHAPe(\Phi) = \sum_{i=1}^{n} \Phi_i \tag{3.19}$$

When meaningful comparisons are possible, we will say that some search spaces are more or less *SHAPely* depending on their relative SHAPe. Note that compared to the local SHAP formula (cf. eq. (2.45)), we have just averaged local SHAP values per dimension (obtaining global estimates as per eq. (2.46)) and removed the expectation term, it being irrelevant for perimetric minimization. Using the SHAPe as a measure of dimensional magnitude instead of range, we can k-partition the original search space with the same strategy as described in the previous section - but now with an allocation of splits heuristically weighted by the *significance of choice* within each dimensional range. In our leading example of this section, variable y would then be preferred for splits, even though it has a much smaller range, since its estimated influence on the objective value is astronomical compared to variable x.

Note that this partitioning scheme won't necessarily make the obtained search space partitions less SHAPely than the original one, or even ensure that partitions are equally SHAPely. The idea is not (erroneously) to try to make choices within each subproblem more or less significant. Instead, the idea is to limit the range of options within subproblem dimensions, heuristically weighted by their significance for solving the global problem, which is what we ultimately want to do.

The utility of this partitioning heuristic, as opposed to e.g. not using partitioning at all, of course depends on the problem-algorithm pair under consideration. Yet my intuition would be to consider it for e.g. high-dimensional, complex problems, where the dimensions might be evenly sized, but global SHAP values (as estimated by the library), indicate large discrepancies among variables in terms of influence on the objective. In such cases, SBO algorithms will likely have to spend a large chunk of the overall trial budget on the DoE for proper interpolation, and might have to spend a lot of computational resources to provide remotely useful solutions to auxiliary problems. Under a limited trial budget, my best bet would then be to run a fast local optimizer like MADS in several search space partitions, emphasizing importance of choice to get the most bang for the buck.

The final piece of the puzzle is how we can estimate global SHAP values with the library, which is all we need to feed to algorithm 4 etc. Upon detecting that the user has opted for "shap" partitioning with a non-empty TrialHistory dataset, we

go through the following steps:

1. Reproject objective values in the TrialHistory dataset with a consistent measure of penalty.
2. Fit a surrogate model to the projected dataset, standing in for the true objective function.
3. Employ the Spark cluster to estimate global SHAP values, using the Monte Carlo approach from eq. (2.49).

As for step 1, note that unlike existing partitioning schemes within BBO [125], we don't need an arbitrary number of extra objective evaluations to e.g. estimate all possible variable interactions in our approach: We just use whatever knowledge we already have about the search space. It thus extends algorithms already using e.g. LHS for initial solutions or interpolation sets "for free", squeezing out some extra value from the samples we already have on hand. The library default is to use the augmented penalty function of HRM (cf. section 3.5.6), thus enforcing a strict, yet numerically stable, SFS ranking in the interpolation dataset, when problems are constrained.

As for the surrogate model in step 2, we use a cubic-kernel RBF of the form in eq. (2.15) by default - the reason being that it is relatively cheap to do draw our desired number of Monte Carlo samples with it, and that such high-dimensional RBF models have been successful within other partitioning strategies based on interpolation [155].

In step 3, we run a number Spark tasks on the cluster. For each task, an executor allocates its cores to each estimate the Shapley value of variable i in design point j, by drawing a number of Monte Carlo samples. More fine-grained alternatives might of course have been possible, in which we just draw all of our Monte Carlo samples as result rows of one big query and then aggregate everything in the end. Yet I opted for the present level of granularity to keep tasks reasonably small, while avoiding the overhead introduced by doing a potentially massive amount of UDF calls for interpolation with the surrogate model on each row.

The Spark SQL query for estimating global SHAP values on the cluster goes as follows:

```
his.trials.map(_.solution).toDF("x")
  .crossJoin(spark.range(ss.nVars))
  .withColumn("seed", rand(seed))
  .withColumn("shap", shap($"x", $"id", $"seed"))
  .groupBy("id")
  .agg(expr("mean(abs(shap)) AS shap"))
  .select("id", "shap").as[(Long, Double)]
  .collect()
```

So, on line 1 we project the solution vectors of TrialHistory to obtain our points for Shapley value estimation. We then on line 2 cross join with a range of integers, denoting variables $[1..n]$, and thus which variable to estimate the Shapley value for in each task. After adding a random seed to each task on line 3, we use a UDF named $shap(\cdot)$ to do the Monte Carlo experiments with the RBF surrogate. Lines 5-6 we aggregate local Shapley values by variable i's to obtain the global SHAP values (eq. (2.46)), before collecting them in an array of n tuples in lines 7-8, containing each variable ID with its corresponding global SHAP value estimate.

As for what happens inside $shap(\cdot)$, this in summarized in algorithm 5. It is actually a functional closure, encapsulating the RBF surrogate f and global SearchSpace ss. Apart from these hidden parameters, various hyperparameters regarding the calculation of batches, along with an early stopping criterion, can be specified.

Algorithm 5 Monte Carlo Shapley Estimation

1: **procedure** MCSE(x, i, $seed$, f, ss, $maxSamples = 50000$, $batchSize = 1000$, $z = 1.96$, $threshold = 0.01$)
2: $rng \leftarrow Random(seed)$
3: $numBatches \leftarrow maxSamples \div batchSize$
4: $batchNum \leftarrow 0$
5: $stats \leftarrow BatchStatistics()$ ▷ Used for efficient descriptive statistics
6: Runs until maximum samples or convergence reached:
7: **while** $batchNum < numBatches$ **and not** $converged(stats, z, threshold)$ **do**
8: $sample \leftarrow genSample(batchSize, x, i, ss, rng)$ ▷ i.i.d. points, cf. eq. (2.48)
9: $masked \leftarrow f(sample)$ ▷ Objective value without x_i participating
10: $sample[:, i] \leftarrow x_i$ ▷ Assign x_i to entire column i
11: $unmasked \leftarrow f(sample)$ ▷ Objective value with x_i participating
12: $marginals \leftarrow unmasked - masked$ ▷ Marginal estimates, cf. eq. (2.49)
13: $stats.update(marginals)$ ▷ Update running mean and variance
14: $batchNum \leftarrow batchNum + 1$
15: **return** $stats.mean$

As shown in line 5 and 13, we use running means and averages to avoid needless recomputations of these statistics from scratch. The final running mean is ultimately what is returned by the algorithm (line 15) as the estimated SHAP value for value i in design point x. Otherwise we enter the main while loop, calculating marginal estimates in batches (cf. eq. (2.49)). As for what our "reference dataset" is here (i.e. Z in section 2.5.3), the domain over which we calculate expectations is as previously mentioned design points in the relaxable domain of f. We don't ever need to construct this dataset *explicitly* when generating samples on line 8, but can just use SearchSpace helper methods to quickly generate random bounds-feasible

design points.

As described previously, we can use the normal distribution to reason about the possibility of early stopping in this approach (cf. section 2.5.3). At M collected samples, with the current mean and standard deviation being $\hat{\mu}$ and $\hat{\sigma}$, respectively, our confidence interval for the chosen z score z is [182]:

$$\hat{\mu} \pm z \cdot \frac{\hat{\sigma}}{\sqrt{M}} \tag{3.20}$$

The rightmost factor is merely our estimate of the population standard deviation at M samples, as discussed in section 2.5.3. If for instance $z = 1.96$, as is the default of algorithm 5, we are 95% statistically confident that the true mean lies in this interval at M samples. The early convergence check in $converged(\cdot)$ on line 7, inspired by a solution I found elsewhere [181], simply checks whether the range of this interval, denoting possible error, becomes so small that it maybe isn't "worth it" to draw more samples, and we should just stop pronto. Concretely, for $M > 0$ and $\hat{\mu} \neq 0$ it checks whether:

$$\frac{z \cdot \frac{\hat{\sigma}}{\sqrt{M}}}{|\hat{\mu}|} < threshold \tag{3.21}$$

So, using library default values, if we are 95% statistically confident that the true mean is off by at most 1% of the estimated mean. Note that this approach isn't statistically immaculate, since there is some additional estimation error involved in using the *sample* mean in the formula above. We nonetheless accept this predicament, since any small biases will need to be corrected anyway by taking a global mean over all local Shapley values with the returned result of algorithm 5.

This concludes the story of what I've played around with this semester.

Chapter 4

Related work

This chapter goes over relevant existing technical solutions that DIBBOlib relates to, of which several are more or less antagonistic sources of inspiration.

4.1 PA systems

As described in chapter 1, PA incorporates forecasting into optimization models. The need for integrating predictive methods, such as ML, and optimization methods, such as LP, into efficient and user-friendly workflows motivates the development of prescriptive frameworks and DBMSes - of which only few exist as of today. The most full-fledged solutions, integrating both prediction and optimization facilities into DBMSes, are *prescriptive DBMSes* [58].

While the end result of this project is perhaps better described as a prescriptive *framework* or *tool*, integrating directly with MLlib for forecasting, DIBBOlib took its onset in a project about PA [115], and therefore relates to such solutions in several ways.

SolveDB+ is an extensible prescriptive DBMS built on top of PostgreSQL, and is arguably state-of-the art within its class [160]. It offers an entirely SQL-based workflow for PA. The overall architecture is based on providing an extensible way of integrating existing optimization solvers (such as CPLEX) and predictive tools to run in-DBMS [159]. Users can specify advanced *data-driven* analytics queries, including optimization queries, through an extension of SQL known as *solve queries* Using an example drawn from [58, p. 589]:

```
SOLVESELECT r(stock) AS
  (SELECT itemID, profit, null :: integer AS stock
   FROM ItemFacts)
MAXIMIZE (SELECT sum(profit * stock) FROM r)
SUBJECTTO (SELECT sum(stock) <= 70 FROM r)
USING solverlp();
```

The new *SOLVESELECT* clause on line 1 introduces an SQL view *r* of an input data table *ItemFacts* with an integer decision variable column *stock*, which is queried on line 4 and 5 to specify the objective and constraints of the problem model, respectively (cf. 2.1). The *USING* clause specifies a preferred optimization solver to use, in this case one using LP. Yet SolveDB+ notably offers an *automatic algorithm selector* making this clause optional in most cases [159]. On top of this, SolveDB+ offers support for *model management*, i.e. a way to store and reuse predictive and prescriptive model components [160].

DIBBOlib draws inspiration from SolveDB+ in also offering *data-driven analytics* facilities with support for complete prescriptive workflows, to be used alongside an *SQL-based* system core (Spark SQL) (cf. section 2.1). DIBBOlib also similarly emphasizes *extensibility*, allowing clients to implement their very own sBBO and constraint handling methods at their desired level of involvement. In some capacity, DIBBOlib additionally offers (an albeit limited kind of) *automatic algorithm selection* and *model management*, through the how-to wizard and by interoperating with the MLlib pipeline system, respectively section 3.4.2.

A significant departure is however that SolveDB+, being built on top of PostgreSQL [159] is designed as a *single-node* system, not outright made for the Big Data use case, which is the role played by Spark in the presented library. To my knowledge, current analytics solutions built around the data-intensive use case generally only support forecasting or optimization out of the box, but not both [58, 115].

SolveDB+ also more ambitiously aims to support a broad range of different optimization approaches, from LP to population-based BBO [164], while the proposed library limits itself to a fairly niche group of methods.

Tiresias is another single-node PA system, offering a Datalog-based query syntax [112]. Of note, DIBBOlib borrows a few ideas from Tiresias in terms of problem modelling: Tiresias being the inspiration behind the key concepts of *hypothetical columns* ($hcol(\cdot)$), *how-to queries*, and *what-if models*.

A *what-if query* makes a hypothetical change to a database and assesses the outcome, without the user having to manually make or unmake said changes [47, p. 252]. The goal is usually to forecast some hypothetical performance metrics under a set of assumptions related to past data [16]. In DIBBOlib, notice that what-if model evaluations, in which we change a set of columns on an input dataset to assess the effect on objective and constraint function values, fit this description (cf section 3.3).

What-if queries were ”invented” some 20+ years ago in an OLAP setting [96]. The people behind Tiresias later on came up with the concept of *how-to queries*, which are in a sense ”reverse” what-if-queries [111]. How-to queries instead specify a desired outcome of a database update, and obtain a hypothetical change to

the database achieving this effect [110]. The Tiresias system offers the TiQL language, used alongside SQL, essentially specifying mixed-integer LP problems to be solved. A key concept in TiQL is the one of a *hypothetical table*, e.g. [112, p. 3]:

```
HLineItem(ok, pk, sk, q?)
```

HLineItem is a relational table, with some attributes merely containing problem input data in their column, and other attributes being *unknown*, marked by a trailing "?" for q in the example, denoting decision variables [112]. Unknown attributes have *nondeterministic semantics*, there being several "possible worlds" for instantiating the relation in a logic programming sense, and a feasible set in an optimization sense - governed by a set of *rules* (constraints) as specified in the TiQl syntax. By optimization, Tiresias finds the best possible instantiation of hypothetical tables with respect to an objective function, and provides this as the output of how-to queries [112].

DIBBOlib offers a similar nondeterministic way of thinking about decision variables to Tiresias, them being specified as *hypothetical columns* ($hcol(\cdot)$) in an input dataset, for which we need to find the best possible feasible assignments. Of course, we only work with columnar constants in the library, due to our expected problem-algorithm types (cf. section 3.3).

In DIBBOlib, optimization queries at large, be it through the how-to syntax or Transformer API, similarly specify how-to queries, in the sense that users specify a desired outcome with respect to an input table with hypothetical attributes in the query, and get the best instantiation found in the output table.

4.2 BBO solutions for Spark

To my knowledge, DIBBOlib is the first ever attempt at building an sBBO tool for Spark. There is however a lot of existing research within BBO related to Spark, specifically implementations of various *population-based* BBO algorithms, such as Genetic Algorithms [37, 134], Differential Evolution [75], Particle Swarm Optimization [199, 5], and the like. The JMetalSP library offers a suite of such algorithms, all running on Spark [17].

These solutions leverage Spark for BBO quite differently compared to the presented library. In DIBBOlib, Spark's main role is to scale out objective evaluations, assumed to be relatively expensive, with the core optimization algorithm, assumed to be relatively cheap, generally running entirely on the driver machine - auxiliary solving on the cluster being regarded as a niche option (cf. section 3.6.1).

The aforementioned population-based algorithms grapple with quite different challenges in terms of scalability: Here, the objective function is generally assumed to be cheap to evaluate [42], while the core algorithm updates can get arbitrarily expensive as a function of the utilized population sizes [6]. High-dimensional and/or

highly constrained BBO problems, the "target audience" for these algorithms, tend to require very large populations, which can become a CPU-bound scalability issue [83].

Within research on population-based BBO for Spark, the general approach being pursued is to scale out expensive algorithmic updates, with each executor managing its own subpopulation (e.g [109, 7]). The most popular design pattern is to make executors run algorithmic iterations independently, while at regular intervals making them share their incumbent solutions with each other - the so-called *island topology* [105]. This furthers global progress while keeping I/O requirements relatively low. The role of the driver here is merely to track the number of algorithmic iterations and manage task completion.

Where does the presented library stand compared to alternative BBO solutions for Spark then? These solutions are arguably more *complementary* than anything else. As the above discussion would imply, the associated methods of each solution type are designed to handle different challenges within BBO: computationally expensive and high-dimensional problems models, roughly speaking. In conclusion, one should pick what fits the application, and hope that the problem model is either relatively cheap or simple, or that a hybrid solution [126] is on the table.

Their complementarity is even evident from a purely technical standpoint. Note that population-based solutions for Spark generally push algorithmic updates to executors, *including objective evaluations*. As per the master-worker structure of Spark applications, the driver process is the *single entry point* for executing Spark queries, through its session object (cf. section 2.1), with all references to the session object being set to *null* when tasks are serialized and relayed to executors [165]. It is therefore generally *technically impossible* to perform objective evaluations involving Spark queries with the aforementioned population-based solutions, due to platform limitations.

On the other hand, DIBBOlib *counts* on objective evaluations requiring Spark queries. This might of course cause some wasteful overhead if the objective function is a very cheap one-liner with no input dataset attached to it, and thus motivate looking for alternative solutions, like Genetic Algorithms with JMetalSP [17]. But for the converse scenario, DIBBOlib is otherwise the only ready-made solution to my knowledge.

4.3 Alternative sBBO solutions

Other sBBO solutions already exist today: frameworks, libraries, and solvers. They generally don't show up in yearly Stack Overflow surveys or in any comprehensive literature review, however.

As a small desktop research project, I therefore scoured Google (Scholar) and GitHub Topics for existing solutions, using pertinent keywords like "black-box",

”derivative-free”, ”gradient-free”, ”zeroth-order”, and ”model-based”, combined with ”optimization” and ”library”, ”solver”, ”framework”, ”platform”, ”system” and ”tool”. To limit the search scope and prioritize more prolific solutions, I only looked into repositories with 50+ stars on GitHub, or until Google search results were predominantly unrelated to sBBO. As always, we treat population-based BBO as a separate class of algorithms, and solutions dedicated to those were therefore not included in this survey. table 4.1 provides a visual overview of the polemics to come.

Type	Name	3+sBBO	|Ctx|	Ctx*	Parallel	Multi	Split	Transfer	Spark	Wizard
Solver	NOMAD [124]		✓	✓	✓	✓		✓		
	MISO [113]		✓							
	BFO [24]		✓					✓		
	DAKOTA Solvers [46]		✓							
	DFL solvers [48]		✓	✓						
	SNOBFIT [163]		✓		✓					
	TOMLAB Solvers [187]		✓	✓						
Library	scipy-optimize [129]		✓							
	PyBrain [140]		✓							
	scikit-optimize [153]		✓							
	GFO [61]	✓	✓							
	RoBo [151]		✓							
	blackbox [29]		✓		✓					
	Mystic [121]		✓	✓	✓			✓		
	pySOT [141]		✓	✓	✓			✓		
	RBFOpt [144]		✓		✓					
	Benderopt [20]		✓							
	Blackboxopt [29]		✓							
	GloMPO [62]		✓					✓		✓
	Optim.jl [131]		✓							
	Optimization.jl [130]	✓	✓							
	Surrogates.jl [184]		✓							
	BlackBoxOptim.jl [29]		✓	✓	✓					
	BlaBoO [28]	✓	✓							
	mlrMBO [114]		✓		✓	✓				
	bbotk [19]		✓							
	Blackbox package [29]		✓							
	Optim [128]		✓	✓						
	HOPSPACK [80]		✓	✓	✓					
Framework	Nevergrad [122]	✓	✓	✓	✓					✓
	Google Open-Source Vizier [4]		✓	✓	✓	✓		✓		
	OpenBox [127]		✓	✓	✓	✓		✓		✓
	DIBBOlib	✓	✓	✓	✓		✓	✓	✓	✓

Table 4.1: Existing solutions summarized. *3+sBBO*, denotes access to three or more sBBO method types within our taxonomy in a standard algorithm suite (cf. section 2.3.2). *|ctx|* denotes support for bound constraints for *all* algorithms. *ctx** denotes support for general constraints for *all* algorithms. *Parallel* denotes support for trial parallelism. *Multi* denotes support for bi- or multi-objective optimization. *Split* denotes support for search space partitioning. *Transfer* denotes support for any transfer learning, including checkpointing with ”hot restarts”. *Spark* denotes direct integration with the Spark platform and ecosystem. *Wizard* denotes access to an automatic algorithm selector. While we talk about DIBBOlib as a library in our assumed Spark setting, it may alternatively be regarded as a framework when including the Spark platform as part of the ”whole package”.

I categorize solutions found into 3 groups: Solvers, libraries, and frameworks. Of course, no one seems to agree on what anything is nowadays, but my own definitions are as follows: Solvers offer *high-quality implementations* of individual algorithms, usually in performant languages like C/C++. As a litmus test, mathematicians have a habit of calling these solutions "codes" [150]. Libraries are more focused on *breadth and user convenience*, and offer a suite of algorithms in popular data science or math languages like Python, R, MATLAB and Julia (e.g. [51, 184]). Frameworks are similar to libraries, yet additionally offer *deployment architectures* and/or *general workflow features*, such as benchmarking or visualization utilities (e.g. [21, 101]) - making up more opinionated proposals on how users should structure their applications than libraries.

As for solvers, there is one definite top dog within sBBO, in terms of GitHub stars, benchmark performance, and prominence within research, and that is *NOMAD (Nonlinear Optimization with the MADS Algorithm)*, implemented in C/C++ [13, 148]. As implied, this solver offers the MADS algorithm and variants thereof, with various extensions tailored to them, e.g. for general constraint handling, hybridization with surrogate models, and bi-objective optimization [124]. To illustrate, even though MADS is a Local algorithm at its core, NOMAD is still a contender in benchmark comparisons within Global sBBO [150].

Other solvers I found were much more obscure and limited in scope, with most of them seemingly no longer being actively maintained, or e.g. only offering support for bound constraints. Notables however include MISO, built around a hybrid of several SBO methods, which has previously outperformed MADS on higher-dimensional problems [120], and solvers in the relatively prolific TOMLAB toolbox, which offer algorithms like DIRECT and support for general constraints [188]. Both of these alternatives are "however" implemented in MATLAB.

As for where DIBBOlib stands with respect to these solutions, suppose that my late-afternoon student project implementation of e.g. MADS doesn't consistently outperform high-quality codes like NOMAD in most cases.

In principle, one could of course compile and run NOMAD on a remote driver or on a dedicated cluster machine, and make it run a series of Spark jobs through e.g. a Bash script, printing the obtained objective and constraint function values to stdout, as per the NOMAD API, which is indeed very generic, as per the black-box premise [124].

However, one might argue that integrating DIBBOlib into an existing Spark application or cluster is much more *convenient* than the alternative, or what would similarly be required for deploying the highlighted MATLAB solvers. The library already has handling in place for translating optimization problem models to Spark queries and back again - that is an important part of library core functionality. Without the library, users will have to figure out how to do all of these translations on their own and e.g. print all relevant information to *stdout* in the right format.

In conclusion: while high-quality codes might outperform the library suite, the value provided by the library core might outweigh this con in a practical development setting, where developer productivity is a priority.

As one might expect, Python is by far the most popular language among sBBO libraries, with Julia being the runner-op.

Among Python's data science ecosystem giants, *scipy.optimize* offers implementations of Nelder-Mead as well as DIRECT in its standard suite. Its support for constraint handling however varies by algorithm, with only bound constraints being supported for Nelder-Mead, for instance [129].

The Mystic library [121] also deserves a mention - while it by its design philosophy only aims to sport a handful of algorithms, with Nelder-Mead being the only one of interest to us, it offers the most elaborate support for generalized general constraint handling among any alternative solution I've found, full stop. It combines a symbolic constraint parser with various penalty and barrier methods to provide generic handling across its algorithm suite. As the reader was previously made aware (cf. section 3.5.5), DIBBOlib's approach to constraint handling, including its (ab)use of Catalyst as a constraint function interpreter, actually drew a bit of inspiration from Mystic.

Various other libraries in Python and Julia offer dedicated support for particular branches of sBBO methods, e.g. pySOT [141] and Surrogates.jl [184] in Python and Julia, respectively, for SBO, with pySOT being developed by the creators of SRBF (cf. section 2.3.6).

As for where DIBBOlib stands versus other libraries also offering sBBO, there is of course always the possibility of a potential customer seeing exactly what they need in an alternative tool to ours, and just going with that instead.

However, users regularly using sBBO may not appreciate having to learn how to use a new library for every problem. As a more *general-purpose* sBBO library, DIBBOlib arguably has the upper hand, by sheer *feature disparity*, regardless of its gimmick of leveraging the Spark platform.

As illustrated in table 4.1, *none* of these alternative libraries combines an extensive standard suite of sBBO algorithms with support for general constraint handling - what might be considered *bare-bones* features of any sBBO solution to be deemed general-purpose. In fact, I selected algorithms for the standard suite, so I could say just that, as I aspired to cover all commonly offered algorithms among alternative solutions. Having a number of useful options on hand is essential under the No Free Lunch theorem, since one algorithm won't perform better than average across all problems [197].

Furthermore, as also depicted in table 4.1, DIBBOlib offers some meat on top of bare-bones essentials for a general-purpose solution, apart from offering several unique features not found in any other tool (e.g. section 3.5.6), along with its

complete integration with the Spark analytics ecosystem.

To conclude, DIBBOlib forms a unique niche among others as being general-purpose, combining ideas from disparate solutions to create a sum larger than its parts, while at the same time offering features not found elsewhere.

As for frameworks, there are three main contenders to be aware of: *Nevergrad* by Meta [21], *Open Source Vizier* by Google [63], and *OpenBox* by the AutoML team from the DAIR Lab at Peking University [101].

These all offer deployment options for running optimization in a distributed master-worker architecture. Here, workers do several trials in parallel under a central coordinator requesting each evaluation client-server style [127, 4, 122].

If we consider the whole package of DIBBOlib running on the Spark platform as a framework, then there is clearly some conceptual overlap with the other solutions. Note however, that a crucial difference in focus is also evident from how work is parallelized in other frameworks versus ours.

The parallelism of Spark, and our library by extension, is designed for the *data-intensive* case, in which subtasks of processing a huge distributed dataset are (ideally) allocated to workers for completely data-local processing [35].

While we indeed support trial parallelism (cf. section 3.6.1), we are actually mainly interested in splitting *one* data-intensive trial into several data-local subtasks processed in parallel on the Spark cluster, so it can complete much faster than otherwise possible. The way I see it, this is how we want to distribute units of work in the data-intensive case.

The alternative frameworks are seemingly more designed with the *compute-intensive* case in mind, in that they hand off entire objective evaluations as subtasks to workers without regard for data-local processing - likely not due to neglect, but due to different workload assumptions [127, 4, 122].

To achieve parallel optimization functionality similar to our framework, one could of course run e.g. OpenBox by installing its distributed architecture *alongside* an already running Spark cluster, and make OpenBox workers act as drivers in separate Spark applications for each objective evaluation (noting again local session objects being the single entry points for Spark queries, cf. section 2.1). Yet all we have achieved is just a higher number of driver processes bothering the cluster manager - and such a deployment option, running two distributed frameworks alongside each other to achieve what could be accomplished with one, just seems needlessly convoluted compared to deploying e.g. DIBBOlib, which comes with batteries included for that use case.

We are thus again at a point, where our solution is more complementary to others than anything else, with our distributed architecture being more suited for the data-intensive case, and others possibly being better suited for the compute-intensive case, depending on the needs of the application.

Not to say that there aren't any clear pros and cons. While all alternative frameworks offer a suite of algorithms, only Nevergrad and DIBBOlib offer both Local and Global sBBO out of the box, with the others only offering Global methods [122].

Alternative frameworks all also offer some level of support for general constraint handling, yet with widely varying levels of elaborateness. As also pointed out by the OpenBox developers [127], Vizier only offers *very limited* support for general constraint handling: a single method operating similarly to the death penalty [82] (cf. eq. (2.34)). After all, they are mostly focused on the hyperparameter tuning use case for sBBO, which generally only requires simple bounds handling [63].

As for Nevergrad, it uses the same penalty formula to handle *all* Relaxable constraints, with a default of just factoring 10^5 into infeasible objective values, regardless of scale [123]. Again, this is likely just a product of having a different focus than ours, which is more general-purpose than e.g. hyperparameter tuning.

As for OpenBox' own constraint handling facilities, they follow a different design philosophy than ours, using more advanced methods per-algorithm with varying levels of support across the board [127], as opposed to DIBBOlib's aim for a generic solution accommodating more algorithms, including client implementations of sBBO.

The three alternative frameworks do offer some nice features, that DIBBOlib doesn't have: Support for multi-objective optimization [127], log-scale variables (e.g. the range of 0.1, 1, 10...) [4], algorithm benchmarking [21], horizontal transfer learning (for algorithms that support it) [101], and more.

But at the same time, our framework offers features that the others don't, including e.g. (dynamic) search space partitioning (cf. section 3.6.2), and a new alternative to the staple penalty method (cf. section 3.5.6).

In sum, DIBBOlib, regarded as a framework when combined with its assumed platform, can be regarded as being complementary to other prominent sBBO frameworks, having a unique focus on data-intensive processing. While mutual feature disparity entails that no solution is strictly better for every conceivable application, this nonetheless fits recurrent themes within optimization well [197].

Chapter 5

Experiments

There were many possible directions to pursue for experiments, and ultimately not enough time to cover all bases. Among these possibilities, trying to demonstrate situational superiority of the library compared to alternative solutions, was indeed on the table. One could for instance investigate whether NOMAD [124] indeed outperforms my student project implementation of MADS on *all* problems.

The ”burning questions” with respect to scalability and usability within the chosen setting (cf. chapter 1) were however found elsewhere. Of note, DIBBOlib offers some new ”tricks” related to three fundamental issues within BBO research, and the qualities sought after in the problem statement: *Trial parallelism*, *constraint handling*, and *problem partitioning* - corresponding to the dynamic load balancing algorithm based on directional sBBO (cf. section 3.6.1), HRM as a data-driven method for constraint handling (cf. section 3.5.6), and heuristic search space partitioning schemes combining dynamic/greedy programming with e.g. SHAP values (cf. section 3.6.2), respectively.

The first feature relates to scalability and usability both, in that it is meant to provide a way for library performance to scale better with increasing trial budgets, without users having to guess the right level of parallelism a priori (cf. section 3.6.1). HRM was largely motivated by usability concerns, in that it eliminates otherwise difficult hyperparameter choices, as seen with e.g. penalty methods (cf. section 3.5.6). The idea behind the proposed search space partitioning strategies were similarly to offer an alternative to existing approaches that was relatively simple to reason about heuristically, i.e. in terms of boxes (cf. section 3.7).

All of this is fine and well... but are these features any good, performance-wise? The actual merit of these new ideas compared to baseline solutions were certainly entirely unknown to me, thereby warranting experiments that weren't just for show. Three experiments, to be described below, will therefore investigate this. Other areas of the library, like our part-time employment of Catalyst as a constraint interpreter to boost performance (cf. section 3.5.4), as well as the potential usage

of the Spark cluster as an auxiliary problem solver for Model-based methods (cf. section 3.4.4) might of course also have been relevant to look into. But if I had to pick three library core features for experiments, it would be the others just described.

In the following sections, we will first go over the cluster setup before moving on to describe each experiment in turn.

5.1 Cluster setup

A Spark cluster was set up, using three virtual machines. Each were equipped with 32 logical CPU cores and 268 GB of RAM, with AMD EPYC™ 7302P 3.3 GHz CPU's, and a sum total of 295.57 GB disk space among all machines. These machines were provided by the university, of which two were shared with other users - while this provided a nicely naturalistic setting for testing our dynamic load balancing, do note that it was practically infeasible for me to supervise that no run among more than 1000 was impacted by the activities of other users.

Since this cluster was relatively small and single-tenant, and for deployment simplicity, it was decided to use the Spark Standalone cluster manager [177], and not e.g. YARN, but only the HDFS component of Apache Hadoop. Relevant numbers of all main cluster elements are summarized below, reflecting when the cluster came to be last December:

Component	**Version**
Ubuntu Linux (Guest OS)	22.04.4
Java JDK (Oracle)	17.0.5
Apache Hadoop	3.3.4
Scala	2.13.10
Apache Spark	3.3.1
Spark MLLib	3.3.1
Breeze NLP	1.2

We highlight MLlib and Breeze NLP (for linear algebra) due to their status as key dependencies of our library, but note that they come prepackaged with Spark distributions by default [173].

One machine was employed as the Spark cluster manager and HDFS name node, while all three were employed as HDFS data nodes, to make the best use of limited disk space. This was deemed a reasonable solution in our relatively small setup.

Following generic recommendations [50], the number of CPU cores per executor was limited to 5, to optimize HDFS throughput, and the default memory allotment of 1 GB was upped to 30 GB, using the available space with some leftover

headroom. In terms of available cores on each machine, experiments thus ran with a maximum of $\lfloor 32 \div 5 \rfloor = 6$ executors active per machine, or 18 in total, each with 5 CPU cores and 30 GB's of RAM.

All runs were completed by running Spark applications in *client mode* on the cluster master node, which launches a driver JVM through *spark-submit* [177, 166], *separate* from the up to 18 executors. While the driver could similarly be configured for a 30 GB RAM allotment, Spark Standalone offers no standard option for configuring the number of driver cores in client mode. The driver process is simply spawned as a standard JVM, which can use all cores by default. The overt disadvantage of this setup is that the driver process shares CPU cores with its up to 6 local executors, possibly impacting their performance. The alternative would be to run in cluster mode, spawning the driver process in one of the executor processes on any machine.

Still, practical concerns took priority: The driver process running on the same machine on each execution meant that it was easier to frequently log e.g. driver memory usage through local file system writes to the same directory, avoiding alternatives with direct impact on HDFS throughput. It was therefore decided to go with the client mode option - noting also that this condition is the same across all executions and is naturalistic to how Spark applications are usually run in this mode.

5.2 Experiment 1: Load balancing

The purpose of this experiment is to see how the *dynamic* load balancing algorithm of the library used for trial parallelism with SBO algorithms (cf. section 3.6.1) performs compared to a *static* baseline and a fully *sequential* run.

As a secondary purpose, it is also demonstrates the library in its element, with the example problem involving a dataset of non-trivial size.

5.2.1 Example problem

To assess performance under different loads, we use a modified version of a problem from the TPC-DS benchmarking suite (version 3.2.0) for this experiment [190], it being simple to scale the size of the input dataset at will.

We specifically use query 48, due to it having a decent number of substitution parameters [189, p. 117] easily re-framed to decision variables, and due to it querying one of the largest fact tables in the star schema used by the suite. I used a similar problem in my previous semester project [115], but we now add some non-trivial changes and extensions to the original story.

In the spirit of sBBO, we additionally enrich our what-if model with an MLlib decision tree in an extra pipeline stage, doing predictions on output rows [41],

as well as a black-box input constraint with a Spark SQL UDF taking decision variables as input arguments, thus requiring a separate small Spark query for evaluation upon each trial (cf. section 3.5.5).

The pipeline used for the what-if model is as follows:

```
val pipeline = new Pipeline()
  .setStages(Array(tpcds48Modified, mlModel, finalAggregation))
```

tpcds48Modified is simply an *SQLTransformer* with the original TPC-DS 48 query in it, but with the big *store_sales* table instead assumed present in the pipeline input table *__THIS__* - confer appendix A.1.1 to see the modified query in full. We try to be prudent here, and use it as the input Dataset to the optimization query, such that it is cached to default memory levels during the run when possible (cf. section 3.4.2). TPC-DS 48 simply selects a number of sales rows from the *store_sales* table, from a number of different states in the USA, based on demographic information such as marriage and educational status of customers. We use these substitution parameters as decision variables, for workload variety depending on predicate selectivity.

The *mlModel* step is a pre-trained MLlib *DecisionTreeClassifier* doing predictions on all rows of the previous pipeline steps [41]. It uses information from these, along with a made-up decision variable named (customer) *treatment*, to predict for each sales row whether the sale will generate *buzz* online or not (1 or 0).

The final aggregation step calculates the objective for maximization:

```
val finalAggregation = new SQLTransformer().setStatement(
  """SELECT count(*) as objective
     FROM __THIS__
     WHERE predicted_buzz = 1;"""
)
```

So, it just measures how much buzz we expect to get out of engaging with a particular set of customer segments, using a particular treatment in a targeted campaign. We want to maximize this quantity as our objective.

While I have limited experience with workplace politics, I suppose that any nation-wide campaign will likely require approval from upper management. Management is however subject to capricious whims, so we use an additional decision variable to decide when we should present the plan to upper management for approval: No approval renders any proposal infeasible, no matter how good the predicted outcome. I just implemented this constraint function as a Spark UDF assessing the popularity of ice cream and the hotness of weather in different states, aggregated with a business acumen score based on a variable hash, in constant time.

Using *RBFOptimizer* with 1000 trials and a preset random seed, the complete optimization model for our example problem looks like this:

```
val optimizer = new RBFOptimizer()
  .setWhatIfModel(whatIf)
  .setVariables(hcol("treatment") in treatments,
                hcol("MS1") in marriageStats,
                hcol("ES1") in educationLvls,
                hcol("MS2") in marriageStats,
                hcol("ES2") in educationLvls,
                hcol("MS3") in marriageStats,
                hcol("ES3") in educationLvls,
                hcol("STATE1") in westAndPacificStates,
                hcol("STATE2") in midwestAndNortheastStates,
                hcol("STATE3") in southStates,
                0.0 <= hcol("approval_time") <= 28800.0)
  .setInputConstraints(approval($"STATE1",
                                $"STATE2",
                                $"STATE3",
                                $"approval_time") >= 8)
  .setMaximize(true)
  .setNumTrials(1000)
  .setCheckpointPath(path)
  .setSeed(42)
```

We thus have a non-trivial optimization problem on our hands, forming a basis for benchmarking.

5.2.2 Experimental setup

We run experiments under two different workload characteristics and a set of different strategies for handling them, for a total of three experimental variables.

First, we vary the TPC-DS *scale factor* between different powers of two: 1, 2, 4, 8, 16, 32, 64, and 128, using an exponential scale to possibly get more information out of fewer runs. This factor denotes the size of the *entire* TPC-DS dataset in GB's, as created by the official data generator, and is not the actual size of our the five tables used in our example query, which actually make up about 50% of the total dataset size. I used a home-made fork of Databrick's own TPC-DS benchmarking framework [176] to generate the required tables for TPC-DS Q48 at the required scale factors, ultimately stored as compressed Parquet files in HDFS, Spark's default file format [88]. The original plan was to also cover scale factor 256 and 512, but I found out at a later point that experiments would then likely not complete before deadline for hand-in.

As the second experimental variable, we change the *number of executors* per cluster machine from 1 to 6, the total number of executors thus varying from 3 to 18 among these conditions. This is used to assess how different load balancing strategies perform under different levels of parallelism and available computing

resources, regarding e.g. speedup. To achieve this effect, we use the *–total-executor-cores* option with *spark-submit* [50]. Since the Standalone cluster manager always allocates executors to different machines evenly based on the total number of cores available [177], we can just set this option to e.g. 30 to get two executors on each machine with 5 cores each, for a total of 6 executors (I checked executor IP's to make sure).

Finally, we have three load balancing *strategies*. The first one is a fully *sequential* run, with no driver multi-threading, used as a baseline for the others. The second one is the *dynamic* strategy, using sBBO for load balancing on runtime (cf. section 3.6.1). The final one is a *static* strategy, in which we try to determine the right number of threads analytically before running. As previously mentioned, I know nothing about established approaches for load balancing, but this nonetheless makes the proposed strategy truly naive. After some manual testing with 1 executor per machine and the most extreme scale factors at the time, 1 and 512, I observed that a static thread pool size of about 20 and 2 was best for scale factor 1 and 512, respectively. By interpolating the rest of the scale factors between these numbers, and scaling the expected level of parallelism linearly with the number of executors per machine, I then arrived on the following formula: $p = e \cdot (20 - 2 \cdot \log_2(s))$, where p is the assigned thread pool size, e is the number of executors per machine, and s is the TPC-DS scale factor. So by increasing scale factor size, we run with 20, 18, 16... threads as a baseline, scaled up linearly by the number of executors per machine. E.g., at scale factor 32 with 3 executors per machine (9 in total) we run with $3 \cdot (20 - 2 \cdot \log_2(32)) = 3 \cdot (20 - 2 \cdot 5) = 3 \cdot 10 = 30$ threads in the pool.

You might have noticed that there is a substantial number of cases, so I tried to keep metrics simple. To assess relative throughput and the like, we just measure the *total response time* per run, using *System.nanoTime()* before and after calling $optimize(\cdot)$ in the driver. We additionally measure the *mean driver memory usage* over each run, as we are interested in whether all of these parallel workloads are pressuring it in this regard. We use Spark's internal instrumentation for this, which can log the total memory usage of the driver JVM per second to a chosen sink, with us opting for **.driver.jvm.total.used.csv* [118] - accepting that all runs have a few seconds of measurements irrelevant to sBBO in common at startup.

For all runs, we use RBFOptimizer (supporting the dynamic strategy) with the same trial budget of 1000, and with the same (reset-between-runs!) LHS in TrialHistory every time. With 8 different scale factors, 6 different executor amounts, and 3 load balancing strategies, we have a total of $8 \cdot 6 \cdot 3 = 144$ cases to cover.

Since multi-tenant virtual machines with wonky network settings are not always reliable performance-wise, I ran each of these cases five times, as separate Spark applications, using a Python script for managing runs automatically. This

process took about two weeks, give or take. Result metrics for each case are reported below as the average of all runs, with the minimum and maximum measurement discarded.

5.2.3 Results and discussion

Due to the large number of cases, we will focus on the big picture in terms of what was gained from the static vs. the dynamic load balancing strategy. Plots for total runtimes and mean driver memory usage across runs are provided in figs. 5.1 and 5.2, while raw results can be found in appendix A.1.1.

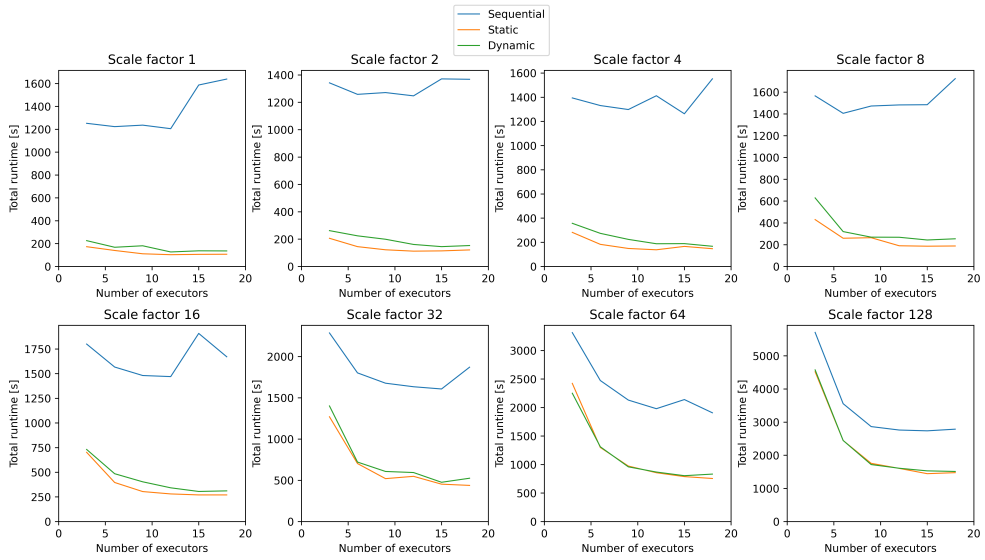

Figure 5.1: Total runtimes for the different strategies at different configurations.

Since all runs were completed with the same workload (query, seed and trial budget), we can talk about speedup by comparing total runtimes. Confer tables 5.1 to 5.4. Row and column indices correspond to number of Spark executors and TPC-DS scale factors, respectively.

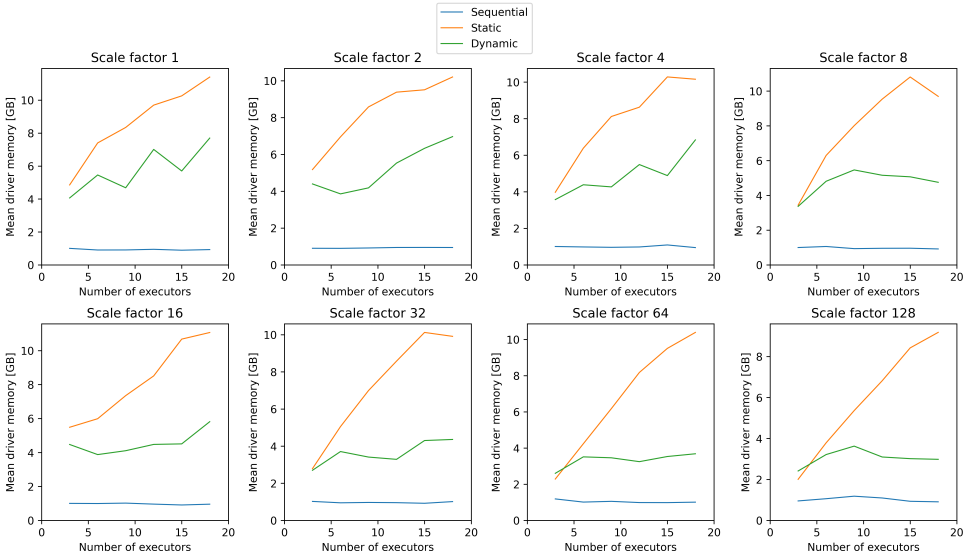

Figure 5.2: Driver memory usage for the different strategies at different configurations.

Num. executors/Scale factor	1	2	4	8	16	32	64	128
3	7.24	6.51	4.94	3.64	2.56	1.8	1.37	1.26
6	8.74	8.68	7.27	5.42	3.96	2.56	1.9	1.45
9	11.14	10.42	8.71	5.58	4.87	3.23	2.19	1.63
12	11.7	11.13	10.23	7.8	5.25	2.98	2.31	1.72
15	14.97	12.03	7.61	7.98	7.07	3.55	2.71	1.9
18	15.32	11.31	10.56	9.16	6.19	4.27	2.53	1.89

Table 5.1: Speedup achieved with the static load balancing strategy, compared to the corresponding sequential runs.

Num. executors/Scale factor	1	2	4	8	16	32	64	128
3	5.54	5.12	3.9	2.49	2.46	1.63	1.47	1.25
6	7.28	5.62	4.86	4.39	3.23	2.5	1.89	1.45
9	6.83	6.39	5.79	5.47	3.67	2.76	2.22	1.67
12	9.49	7.75	7.51	5.53	4.3	2.75	2.28	1.72
15	11.58	9.46	6.68	6.11	6.26	3.38	2.66	1.79
18	12.05	8.94	9.29	6.78	5.37	3.56	2.29	1.84

Table 5.2: Speedup achieved with the dynamic load balancing strategy, compared to the corresponding sequential runs.

Num. executors/Scale factor	1	2	4	8	16	32	64	128
3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
6	1.24	1.42	1.54	1.66	1.77	1.81	1.87	1.85
9	1.56	1.69	1.89	1.63	2.31	2.44	2.49	2.57
12	1.68	1.84	2.04	2.26	2.51	2.32	2.83	2.82
15	1.63	1.81	1.7	2.31	2.6	2.8	3.07	3.13
18	1.62	1.7	1.92	2.29	2.6	2.9	3.21	3.06

Table 5.3: Speedup achieved with the static load balancing strategy, compared to the corresponding static runs with only 3 executors.

Num. executors/Scale factor	1	2	4	8	16	32	64	128
3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
6	1.35	1.17	1.3	1.97	1.51	1.94	1.72	1.87
9	1.25	1.32	1.59	2.34	1.81	2.31	2.35	2.66
12	1.78	1.63	1.9	2.35	2.14	2.36	2.59	2.85
15	1.65	1.81	1.89	2.59	2.4	2.94	2.8	2.99
18	1.66	1.71	2.14	2.48	2.35	2.67	2.7	3.03

Table 5.4: Speedup achieved with the dynamic load balancing strategy, compared to the corresponding dynamic runs with only 3 executors.

The first set tabulates speedup for the static and dynamic strategies, compared to the sequential strategy. So, e.g. cell $e,\ s$ in the dynamic table denotes the total runtime at $e \cdot 3$ executors and scale factor 2^{s-1} for the sequential strategy, divided by the corresponding runtime for the dynamic strategy, as a measure of speedup. In other words, it tells us how much faster it was to pick the static/dynamic multi-threaded strategy for each case, as compared to evaluating all trials in sequence.

The second set tabulates speedup when the number of executors is increased with each multi-threaded strategy. So cell $e,\ s$ in these table is the total runtime at settings 3, 2^{s-1} (lowest number of executors with same strategy), divided by the runtime at settings $e \cdot 3, 2^{s-1}$. We see then how each multi-threaded strategy copes with having more computing resources available on the cluster.

From tables 5.3 and 5.4, it is immediately apparent that adding more executors, i.e. total number of cores, with either multi-threaded strategy did not lead to perfect linear speedup. This would however have been surprising to find, since our I/O bound workload and distributed computing environment don't facilitate a "true vacuum" amenable to such effects.

Increasing dataset sizes however generally led to better speedup from adding more executors, maxing out at about x3 at 18 executors for both multi-threaded strategies. This might be attributed to the fact that the total number of parallelizable tasks for processing the partitioned Parquet files, having e.g. 2,879,992 and 368,647,941 rows in the *store_sales* tables at scale factor 1 and 128 respectively, is

likely larger with increasing scale factors and there thus is a larger possible benefit from parallel processing. We note that the executors were likely not starving for RAM during processing, even when their total memory pool was ”only” 90 GB at 3 executors - at scale factor 128, the size of the uncompressed *store_sales* table was only about half of that, with the rest of the (dimensional) tables only taking up a few megabytes in total.

When we compare sequential runs to the multi-threaded runs (cf. tables 5.1 and 5.2), we see a different pattern of speedup being larger when the scale factor is smaller, maxing out at scale factor 1 and the maximum number of executors for both load balancing strategies. This result ”makes sense”, in that we can expect to be able to run more queries in parallel when processing is cheaper and we have more cores available.

Now for the question of whether static or dynamic was the ”best” strategy. As an aggregate measure *for gross comparison only*, we take the means of the speedup values in our speedup tables to see how each strategy performs overall, across all problem configurations. For tables 5.1 and 5.2, the mean speedup of the static strategy versus a sequential run is 6.03, with $\sigma = 3.87$. For the dynamic strategy, mean speedup is 4.78 with $\sigma = 2.81$. The same metrics for tables 5.3 and 5.4 are $\mu = 1.97, \sigma = 0.66$ for the static strategy, and $\mu = 1.91, \sigma = 0.63$ for the dynamic one.

It may seem like the static strategy is plainly better from the mean values alone. However, an independent t test [54] comparing the set of total *runtimes* of both strategies (using the raw data from table A.1) shows that the difference between them is actually *not statistically significant* by any commonly used threshold ($t(94) = -0.2721, p = 0.7862$) - i.e. no statistical tendency implies that these two samples were even drawn from different populations. fig. 5.1 seems to corroborate this conclusion visually.

While the static and dynamic strategy performed similarly in terms of total runtime, the same thing cannot be said in terms of mean driver memory usage. If we do another t test comparing this memory usage metric across all cases for the two strategies, we see that the dynamic strategy uses significantly less driver RAM than the static one ($t(94) = 7.9170, p < 0.0001, Cohen's\ d = 1.6161$). On top of there being a strong tendency of the dynamic strategy using less memory, we see from Cohen’s d that the absolute impact of picking either is (beyond) *strong* (d > 0.8) [54].

Runtimes being about the same, yet not memory usage, might imply that the static strategy allocates many more threads than needed, with practically no benefit. We can use a concrete example to investigate this hypothesis. For instance, at scale factor 64, the static strategy will allocate 8 and 48 threads with 3 and 18 executors, respectively. Confer figure, illustrating the polled thread pool size used for each trial at the same cases, over all five runs.

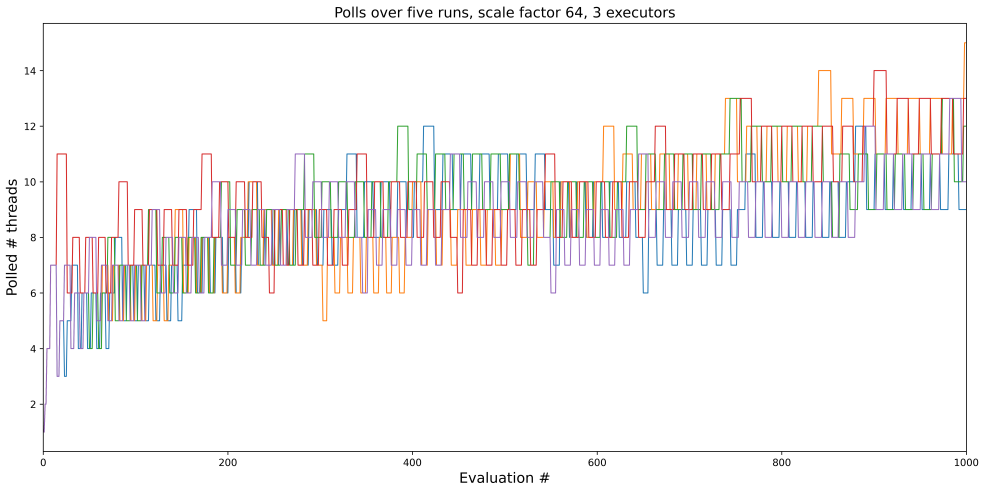

Figure 5.3: A serialized representation of what the thread pool size was for each trial over the course of five runs, with only 3 executors.

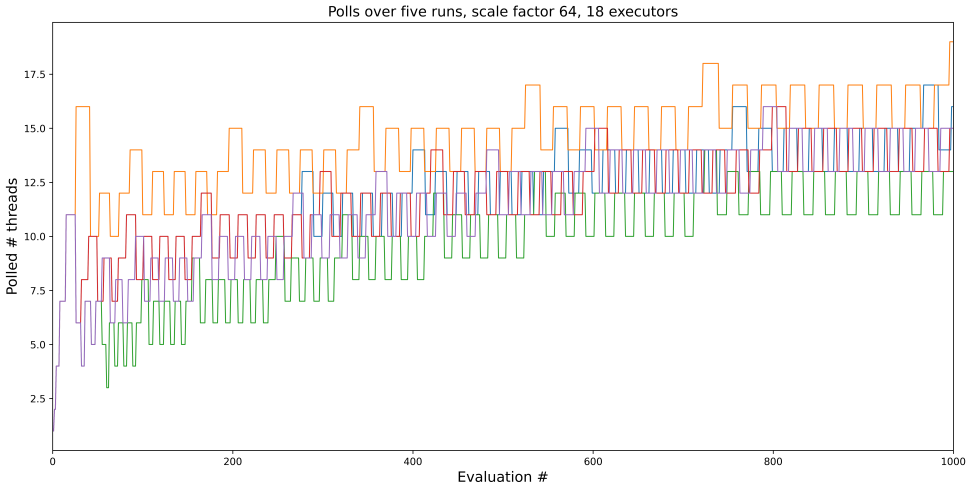

Figure 5.4: A serialized representation of what the thread pool size was for each trial over the course of five runs, with 18 executors.

First, we note that the thread pool size is steadily increasing, which is likely due to the design flaw of algorithm 3 that it is not too fond of decreasing the pool size, unless there is a possible speedup benefit attached to this change (cf. section 3.6.1).

More importantly, however, we see how the dynamic strategy hovers around 6-10 threads for most of the run at 3 executors (cf. fig. 5.3), and 10-14 threads at 18 executors (cf. fig. 5.4). If we look at tables 5.1 and 5.2, speedup compared to the

sequential strategy is consistently lower, yet very similar to the dynamic strategy. However, the story is quite different in terms of memory usage. Confer tables 5.5 and 5.6: At 18 executors and scale factor 64, the dynamic strategy uses 3.68 GB RAM on average while the static one uses 10.39 GB RAM on average - with no significant speedup benefits, as described. While the static strategy is indeed very naive, this exemplifies how difficult it can be to reason about the optimal number of threads outside of a vacuum, based on available computing resources alone, and that it is very easy to overshoot with no significant benefit in terms of speedup.

Num. executors/Scale factor	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
3	4.86	5.18	3.97	3.44	5.49	2.79	2.29	2.01
6	7.4	6.95	6.38	6.3	5.99	5.05	4.24	3.79
9	8.34	8.58	8.12	8.01	7.35	6.99	6.18	5.36
12	9.7	9.38	8.63	9.54	8.51	8.58	8.17	6.82
15	10.26	9.51	10.28	10.81	10.68	10.12	9.51	8.42
18	11.4	10.21	10.16	9.7	11.07	9.91	10.39	9.18

Table 5.5: Average driver memory usage at each different problem configuration, for the static load balancing strategy.

Num. executors/Scale factor	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
3	4.07	4.4	3.57	3.37	4.47	2.7	2.6	2.41
6	5.46	3.86	4.38	4.81	3.88	3.71	3.51	3.22
9	4.68	4.18	4.27	5.46	4.11	3.41	3.46	3.63
12	7.01	5.53	5.49	5.16	4.48	3.29	3.25	3.1
15	5.7	6.33	4.89	5.07	4.51	4.3	3.53	3.02
18	7.7	6.97	6.84	4.75	5.82	4.36	3.68	2.98

Table 5.6: Average driver memory usage at each different problem configuration, for the dynamic load balancing strategy.

So, to summarize, trial parallelism actually brought some decent speedup improvements with in this experiment. While no significant difference in runtimes was found between the static and dynamic strategies, the dynamic strategy overall achieved similar speedup benefits with significant less memory usage involved.

5.3 Experiment 2: Constraint handling

HRM was proposed as an alternative to scenarios in which one would otherwise have used e.g. penalty methods: For when users are not sure about hyperparameter settings for the algorithm-problem pair at hand, or when there is simply no other option available section 3.5.6. I reiterate this point to highlight that HRM is not supposed to compete with more tailored methods for particular problems or

algorithms: Comparisons will therefore instead be based on its closest competitor, a traditional penalty method equivalent.

The purpose of this exercise is not to investigate whether HRM is *always* better than penalty methods or vice versa: averaged over all problems without prior, we already know the answer [197]. Also, figuring out whether HRM is ”often mildly usefull”, where ”often” means studying a lot more than just e.g. 2-3 example problems in my estimation, simply requires an amount of experimental work beyond what this one-man show can accomplish before the deadline.

Instead, our more modest ambition with this experiment is to learn whether or not HRM can *sometimes* be useful, and most importantly *why*, such that possible improvements can be considered and more informed choices of method can be made.

5.3.1 Example problem

From my own readings, I notice that example problems within sBBO research are often ”easy” in terms of constrainedness, usually only involving a couple of constraints beyond bound constraints, if any [191, 150]. Furthermore, there are to my knowledge no standard benchmark suites for constrained problems within this field.

To make things more interesting, I instead looked for possible example problems within the field of population-based BBO research, in which there are basically two important lineages of benchmark suites: The test environments of IEEE’s Congress on Evolutionary Computation (CEC) and the COmparing Continuous Optimizers (COCO) project [77]. As implied in the names, CEC is more generally scoped, and I therefore looked further in this direction.

I ultimately settled on a problem from the CEC2020 suite, which notably had a focus on *real-world* problems [95]. I didn’t want to ”game” the choice of problem too much, ruining the integrity of the study. Picking a problem that was way too easy or hard for any method in the presented library would however only allow for trivial analysis. As a rule of thumb, I therefore steered away from problems with e.g. hundreds of variables and constraints, and looked for something in the ballpark of a total of 20 variables and general constraints instead, which is what the creators of NOMAD estimate the solver can ”efficiently” handle [124].

I picked RC28, the *Rolling element bearing* problem [95], which had a decently complex problem model in my estimation. It is a real-world mechanical engineering problem, in which we need to maximize the dynamic load-carrying capacity of rolling element bearings, as found in e.g. household appliances along with aeronautical and nano-machine applications [66]. Beyond paraphrasing others, I can however not say anything remotely insightful about this problem domain, and will therefore stick to the raw guts of what we are dealing with. The problem definition

can be found in eq. (5.1).

$$
\begin{aligned}
\underset{x\in\mathbb{R}^{10}}{\text{maximize}} \quad & \begin{cases} f_c Z^{\frac{2}{3}} D_b^{1.8} & \text{if } D_b \leq 25.4 \text{ mm} \\ 3.647 f_c Z^{\frac{2}{3}} D_b^{1.4} & \text{otherwise} \end{cases} \\
\text{subject to} \quad & 0.5(D+d) \leq D_m \leq 0.6(D+d) \\
& 0.15(D-d) \leq D_b \leq 0.45(D-d) \\
& Z \in [4..50] \\
& 0.515 \leq f_i \leq 0.6 \\
& 0.515 \leq f_0 \leq 0.6 \\
& 0.4 \leq K_{D_{min}} \leq 0.5 \\
& 0.6 \leq K_{D_{max}} \leq 0.7 \\
& 0.3 \leq \epsilon \leq 0.4 \\
& 0.02 \leq e \leq 0.1 \\
& 0.6 \leq \zeta \leq 0.85 \\
& g_1(x) = Z - \frac{\phi_0}{2\sin^{-1}(D_b \div D_m)} - 1 \leq 0 \\
& g_2(x) = K_{D_{min}}(D-d) - 2D_b \leq 0 \\
& g_3(x) = 2D_b - K_{D_{max}}(D-d) \leq 0 \\
& g_4(x) = D_b - \zeta B_w \leq 0 \\
& g_5(x) = 0.5(D+d) - D_m \leq 0 \\
& g_6(x) = D_m - (0.5+e)(D+d) \leq 0 \\
& g_7(x) = \epsilon D_b - 0.5(D - D_m - D_b) \leq 0 \\
& g_8(x) = 0.515 - f_i \leq 0 \\
& g_9(x) = 0.515 - f_0 \leq 0 \\
\text{where} \quad & x = [D_m,\ D_b,\ Z,\ f_i,\ f_0,\ K_{D_{min}},\ K_{D_{max}},\ \epsilon,\ e,\ \zeta] \\
& f_c = 37.91 \left[1 + \left[1.04 \left(\frac{1-\gamma}{1+\gamma} \right)^{1.72} \left(\frac{f_i(2f_0-1)}{f_0(2f_i-1)} \right)^{0.41} \right]^{\frac{10}{3}} \right]^{-0.3} \left[\frac{\gamma^{0.3}(1-\gamma)^{1.39}}{(1+\gamma)^{\frac{1}{3}}} \right] \left[\frac{2f_i}{2f_i-1} \right]^{0.41} \\
& \gamma = \frac{D_b \cos(\alpha)}{D_m},\ f_i = r_i \div D_b,\ f_0 = r_0 \div D_b,\ \alpha = 0 \\
& \phi_0 = 2\pi - 2\cos^{-1}\left[\frac{[(D-d) \div 2 - 3(T \div 4)]^2 + [D \div 2 - T \div 4 - D_b]^2 - [d \div 2 + T \div 4]^2}{2[(D-d) \div 2 - 3(T \div 4)][D \div 2 - T \div 4 - D_b]} \right] \\
& T = D - d - 2D_b,\ D = 160,\ d = 90,\ B_w = 30
\end{aligned}
\tag{5.1}
$$

Mind the "*maximize*"' in results to come. As shown, this is a mixed-integer, non-linear, generally constrained problem, sporting 10 variables and 9 general constraints. Variables consist of 5 "normal" decision variables and 5 design parameters, the latter of which are modelled as decision variables only present within constraint functions, and therefore have no inherent effect on objective values. A trial budget of 100,000 is allotted for solving this problem within the benchmark suite [95].

While 19 variables and general constraints in total might sound pitiful to some, note that solving these kinds of problems is an entirely different ballgame than e.g.

LP. Even if we were to use gradient information etc., our best option would likely be a multi-start strategy with local optimizers starting from random positions in the search space [91]. Solutions are tentatively reported in terms of being "the best known solution as of yet", without guarantee of them being the global optimum [95].

Following along the provided MATLAB code [133], I implemented the optimization model with the library, using mostly standard Spark SQL, while resorting to UDF's for more complicated expressions. The objective function went like this, for instance, using a UDF for f_c in eq. (5.1):

```
val objective = expr("""CASE WHEN Db <= 25.4
                            THEN fc(Db, Dm, fi, f0) *
                                 pow(Z, 2.0 / 3.0) *
                                 pow(Db, 1.8)
                            ELSE 3.647 * fc(Db, Dm, fi, f0) *
                                 pow(Z, 2.0 / 3.0) *
                                 pow(Db, 1.4) END""")
```

5.3.2 Experimental setup

As our two experimental variables, we use different optimization algorithms combined with different constraint handling methods to give RC28 a go.

First, to represent all axes within sBBO (section 2.3.2), I opted for directional method MADS (Local/Direct) as well as the basic SRBF SBO strategy (Global/Model-based), with the latter being chosen as a lightweight alternative to Bayesian Optimization since I was tired of running hour-long experiments at this point.

Second, we use one among two constraint handling strategies in our runs, both using library defaults: *HRM* (mixed penalty, ten iterations), as well as a traditional penalty method (also mixed penalty, ten iterations, base magnitude 1, and adjustment factor 2).

As for metrics, this is simple: We look at *solution quality* after a preset number of trials, with solutions being ranked with the SFS scheme (cf. section 3.5.6), as is also the standard within the CEC's competitions. Furthermore, we look at the total *response time*, by the same procedure as in the previous experiment, since we are interested in how much slower HRM runs than the alternative, given the time complexity of its more involved recalibration step. On the other hand, memory usage for e.g. storing Arrays with a few hundred Doubles in them wasn't as much of a concern, so I didn't bother with measuring it for this experiment.

We also follow CEC's standards in doing 25 runs for each condition with different seeds given to algorithms, since LHS, MADS and SRBF are all stochastic methods [95]. In a way, we thereby actually test 25 variants of each strategy in total. We report statistics over all 25 runs below.

For the trial budget, we assign a total of 1,000 evaluations, i.e. 1% of the allotted budget in the suite - this amount was just an initial cautious guess, since I didn't want to overshoot.

Following generic recommendations for constrained problems [57], we use a sizeable *third* of this budget on symmetric LHS's, providing interpolation sets and initial solutions for SRBF and MADS, respectively.

MADS and SRBF then have a total of 667 trials for optimization, split evenly into ten for different penalty setting with the two constraint handling methods

For analysis, I logged every trial evaluated for each run on an assigned checkpoint path.

5.3.3 Results and discussion

The found results are much more simple to analyze than in Experiment 1. Details for all runs are available in appendix A.1.2. Look to table 5.7, which summarizes the best solutions found among all 25 runs, as per an SFS ranking, for each paring of optimization algorithm and constraint handling method. While there are understandably no official CEC guidelines on what to do about LHS's, I ultimately chose to discard the initial 333 LHS trials when aggregating these metrics from each run. Results therefore reflect what was accomplished while constraint handling was "active". What was accomplished with LHS on its own across all runs is summarized separately in the table.

Optimizer	Strategy	Best	Median	Mean	Worst	Std	FR	MV
LHS	N/A	54295.9689	35979.3059	35513.8563	12853.1212	9236.1075	100	0
MADS	Historical	81091.3997	69551.6104	70589.0316	52199.8693	8168.2231	100	0
SRBF	Historical	70471.8534	50676.7498	49477.3814	21170.9085	12523.9587	100	0
MADS	Penalty	139225.6286	110887.4738	108285.5663	56477.8487	23874.3805	0	7.8014
SRBF	Penalty	111529.1367	8668.8394	40718.4443	5698.9814	45531.352	0	7.3236

Table 5.7: Statistics over the objective values of the best SFS-ranked solutions returned across different algorithms and constraint handling methods. FR = Feasibility rate, the proportion of runs eliciting feasible solutions. MV = Mean Violation, the mean summed constraint violation of solutions across runs.

We immediately note from the feasibility rate (FR) that the penalty method didn't elicit a single feasible solution during *any* MADS or SRBF run, while HRM always did for both. The higher average objective values for MADS are not connected to any feasible solutions. In fact, running LHS with no regard for constraints was a much better strategy than running MADS or SRBF with the penalty method!

The low mean constraint violation values of about $7 - 8$, obtained by summing all violation values (cf. eq. (2.5)), seem to suggest that the underlying reason for this poor performance is connected to the fact that violation values are on a much

smaller order of magnitude than objective values in this problem. A penalty magnitude of 2^{10} on the final penalty iteration therefore won't make infeasible solutions anyway near unappealing enough to the underlying optimization algorithm.

On the other hand, HRM scales constraint and objective functions to be on about the same scale, circumventing this problem. One might otherwise have to learn about these scaling issues by trial and error and fix them manually when using standard penalty methods. And even if one managed to find a sufficiently large penalty magnitude, it might have to be so large that it hijacks the significance of the objective value during optimization [55], and we thereby create a new problem by solving the original one.

One caveat here is that our mileage with HRM might have varied, depending on the size of the available LHS, and the distribution of violation values across the search space. In our case, this seemed to be no issue. Another caveat is that running HRM took about 10 seconds longer on average, i.e. 1 seconds per recalibration ($\mu = 10.4477, \sigma = 1.0816, n = 50$). Apart from the more expensive update calculations, this can of course also be attributed to the convenient, but likely inefficient, library design of using DataFrames etc. for generally very small datasets in TrialHistory.

To illustrate what difference the choice of constraint handling method makes during optimization, look to MADS's go at the problem on run 25 with the two different methods on figs. 5.5 and 5.6. The choice of 67 as the tick mark interval on the x axes might seem a bit quaint, but it simply reflects roughly when the pertinent constraint handling methods do their respective updates. With the penalty method (cf. fig. 5.5), violation values are a blip on the radar, so MADS seemingly dives deep into the infeasible region for better objective values. When using HRM (cf. fig. 5.6), MADS instead converges towards the best known feasible solution [66].

When running with HRM, MADS and SRBF overall performed quite differently, with MADS clearly being superior in terms of obtained solution quality (cf. table 5.7). Part of this might be explained by the fact that MADS, being a Local algorithm, can work more efficiently when the trial budget is as small it is, since SRBF spends additional effort on exploring the search space for more accurate interpolation in the future. With HRM transforming contours in the search space substantially across iterations, MADS will likely use different incumbents and paths across different iterations, effectively operating as a multi-start algorithm [91], which proved quite effective in the end.

The best known solution to this problem has the objective value of 81843.3 [95]. In 1% of the allotted trials for the problem, MADS finds feasible solutions with objective values over 80,000 in 20% of all runs, which is not too shabby in my estimation, and otherwise finds good-quality solutions across the board, with the mean and median objective value (cf. table 5.7) being very close to the best solution to the problem known by mechanical engineers, before people started to throw Genetic Algorithms with population sizes of 4500 at it [66].

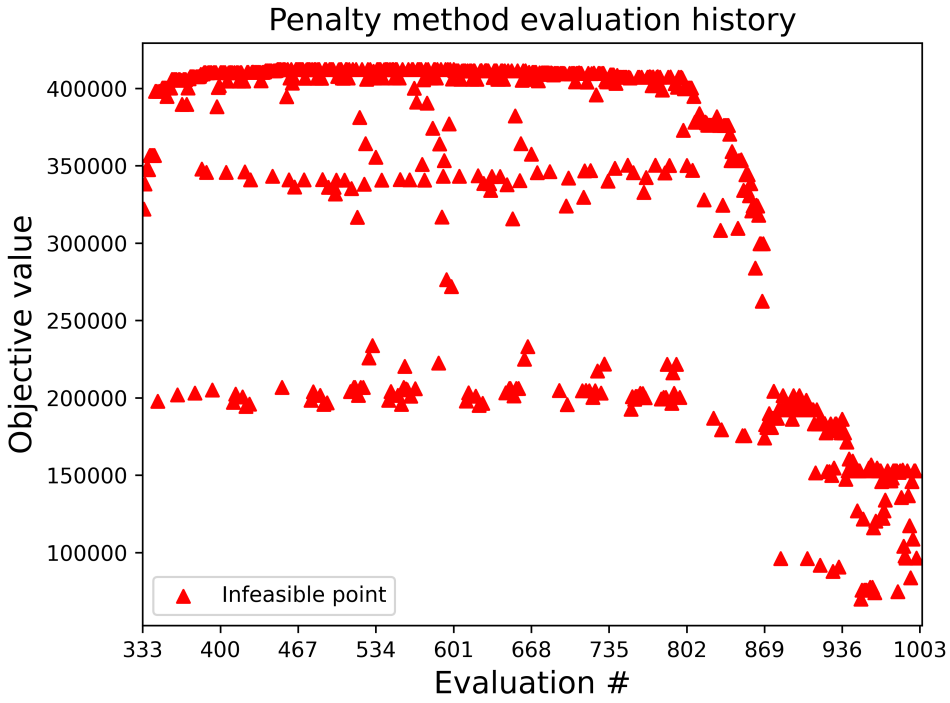

Figure 5.5: "Convergence plot" for run 25 with MADS and the penalty method.

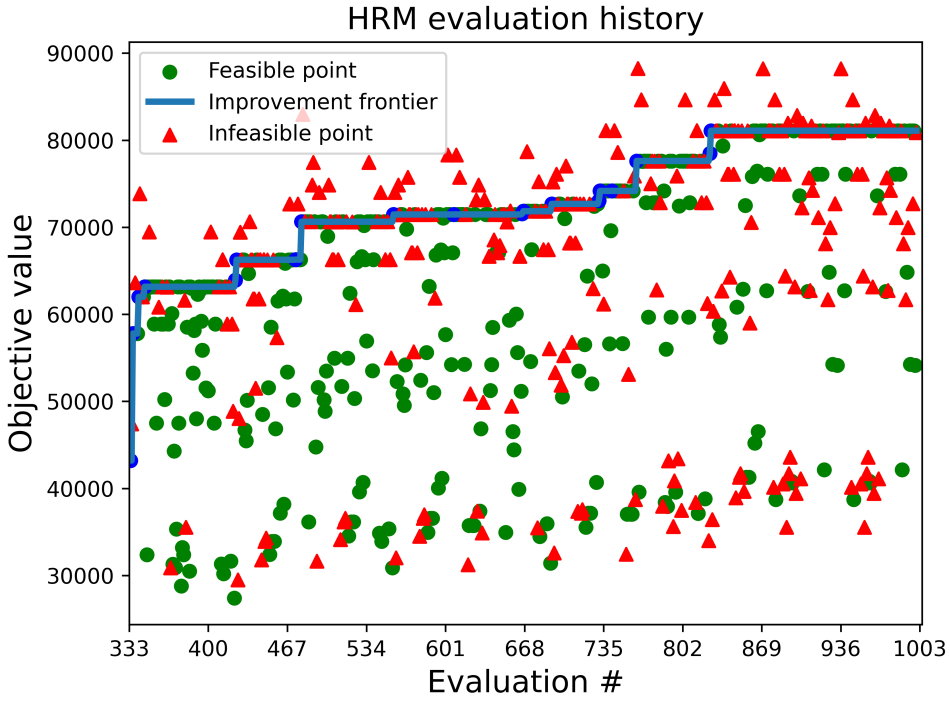

Figure 5.6: Convergence plot for run 25 with MADS and HRM.

5.4 Experiment 3: Search space partitioning

DIBBOlib offers a number of different search space partitioning strategies, including the option of minimizing the SHAPe of the global search space with a combined dynamic/greedy programming approach (cf. section 3.7). Similarly to the previous experiment, we will now look into the question of whether and how using such a strategy or the range-based one is any good compared to running with no partitioning at all.

5.4.1 Example problem

I looked to CEC's competitions again for an example problem. For us to have a reason to consider SHAP-based partitioning specifically, it was decided to pick a problem, in which some variables clearly have more influence on the objective value than others. Digging deep, I found just the right problem, F19 in the CEC2008 suite [186]:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \sum_{i=1}^{n} \left[\sum_{j=1}^{i} (x_i - z_i) \right]^2 \\ \text{subject to} \quad & x \in [-100,\ 100]^n \end{aligned} \tag{5.2}$$

The number of dimensions n can vary freely in the benchmark suite. For our purposes, we set $n = 20$, i.e. "the highest number a high quality implementation expects to efficiently handle" [124].

$z \in [-80,\ 80]^n$ is a uniformly random chosen *shift vector*. The shift vector makes the problem *fully non-separable*, in rough terms meaning that we cannot optimize any variable independently from the others [186]. The solution is z with the minimum objective value of 0. Turns out the problem even has an auspicious name for the SHAP value estimation that we are about to embark on: *Shifted Schwefel 1.2* [116]. To give an idea about the shape of this function, eq. (5.2) provides a surface plot for a 2-dimensional case.

Note that the number of times x_i repeats in the inner sum in eq. (5.2) is inversely proportional to i, with x_1 repeating n times, x_2 repeating $n - 1$ times, and so on. Analytically speaking, variables thus seem to have a very different impact on the sum total, in a non-trivial way, due to the shift vector.

We can however provide actual numbers for how much this skew matters, using the Monte Carlo approach in the library. With a random shift vector and 1,000-sample LHS, we get global SHAP values for a 20-dimensional problem as shown in table 5.8.

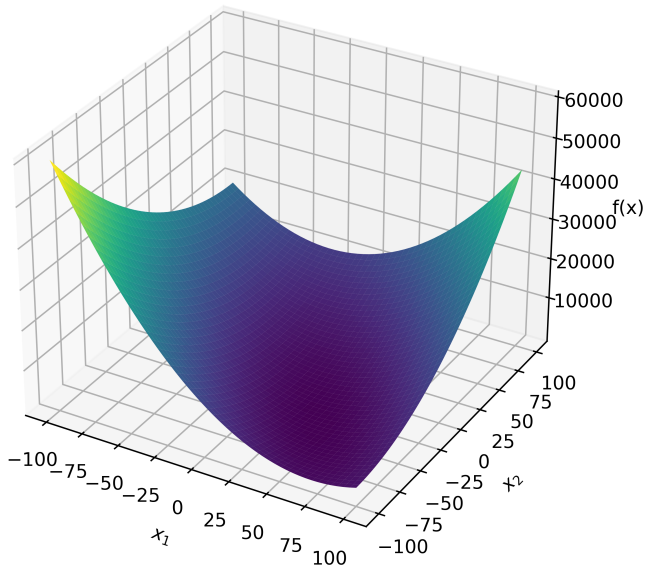

Figure 5.7: Surface plot for Shifted Schwefel 1.2, where $n = 2$ and $z = [42, -42]$, with z also being the location of the global minimum.

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
186238	185520	180357	173749	172334	164028	158650	151160	143458	139414
x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}	x_{19}	x_{20}
130831	121151	105511	90470	72532	59624	49961	35687	25152	15466

Table 5.8: Shifted Schwefel 1.2 global SHAP values from a 1000-sized LHS and a random shift vector.

Here it is evident that the choice of assignment for x_1 matters more than *ten times* as much for the objective value as the choice of assignment for x_{20}.

The question is then whether this seemingly good use case for SHAP value based partitioning actually walks the walk.

5.4.2 Experimental setup

In this setup, we combine different optimization algorithms with different partitioning strategies. This time, we pick algorithms with some idea about them benefitting from partitioning the search space of the example problem.

We ultimately run with *MADS* and *DIRECT* (cf. sections 2.3.3 and 2.3.4). While the objective function is unimodal, we are dealing with a non-trivial number of dimensions, so algorithms might take a while to converge. Local algorithm MADS giving different subregions a shot to improve global exploration overall under a limited trial budget is a simple heuristic borrowed from research literature [191].

As for DIRECT, recall that this algorithm does a lot of search space partitioning on its own (cf. section 2.3.4). It essentially uses a divide and conquer strategy to get as much information as possible out of few evaluations. Splitting dimensions for it beforehand is therefore a possible heuristic for expediting search in select dimensions. The example problem having a smooth objective function is also a good fit for DIRECT, which operates on assumptions of Lipschitz continuity.

As for partitioning strategies, we use one of four. We of course try running with *none*, as a baseline, using the full trial budget on one optimization run in one search space partition. Furthermore, we try out a *best* case, *average* case, and *worst* case split in terms of SHAP values: Splits into 2, 4, 8, or 16 partitions, to see if e.g. a high number of splits stretches the subproblem budget too thin.

For the best case split, we just specify the desired number of splits, and let the library do its Monte Carlo estimation, dynamic programming etc. This way we can also see, if we are actually practically able to get the optimal splits from smaller samples. Based on the large sample drawn, we estimate that the best obtainable partitions based on SHAPe will split dimensions $x1$ up to $x4$ into two, depending on which power of two we are working with.

We set the average and worst case splits statically, since if we told the library to e.g. dynamically split on range to obtain an average case, an artifact of how dynamic programming is implemented (cf. algorithm 4) would then actually assign optimal SHAPe splits to the evenly sized dimensions of the problem. The static split Maps were instead defined as follows:

```
val averageSplitMaps = Map(
  2 -> Map("x10" -> 2),
  4 -> Map("x10" -> 2, "x11" -> 2),
  8 -> Map("x9" -> 2, "x10" -> 2, "x11" -> 2),
  16 -> Map("x9" -> 2, "x10" -> 2, "x11" -> 2, "x12" -> 2)
)
val worstSplitMaps = Map(
  2 -> Map("x20" -> 2),
  4 -> Map("x20" -> 2, "x19" -> 2),
  8 -> Map("x20" -> 2, "x19" -> 2, "x18" -> 2),
  16 -> Map("x20" -> 2, "x19" -> 2, "x18" -> 2, "x17" -> 2)
)
```

So, for the average split, we just fill in dimensions around the mid-range in terms of SHAP values, and for the worst one, we prefer splitting dimensions with the lowest SHAP values before all others.

As the example problem is not generally constrained, our main metric for this experiment is simply the obtained *objective value* after a preset number of trials - 2000 this time, due to the higher number of problem dimensions. We use part of this trial budget on a small LHS sample to estimate global SHAP values, $2 \cdot (n + 1) = 2 \cdot (20 + 1) = 42$ samples in total, even when using static partitioning, to

make comparisons fair. This LHS is also provided as input to both optimization algorithms, for e.g. initial incumbents for MADS, which will otherwise default to a random point within the bounds of its subregion.

As in Experiment 2, we execute the same runs with different seeds 25 times and report statistics based on that, noting that DIRECT is not a stochastic algorithm, but the initial Monte Carlo estimation of SHAP values is, thus possibly impacting results.

We also measure the *total runtime*, to see how long dynamic splits with initial SHAP value estimation take compared to runs without it.

5.4.3 Results and discussion

Results aggregated over all 25 runs can be found in table 5.9. More details can be found in appendix A.1.3.

Optimizer	Strategy	Split	Best	Median	Mean	Worst	Std
LHS	None	1	52035.7585	82846.8922	88325.3542	128827.0857	23192.4497
DIRECT	None	1	14768.9843	14768.9843	14768.9843	14768.9843	0.0
DIRECT	Worst	2	10123.2826	10123.2826	10123.2826	10123.2826	0.0
DIRECT	Worst	4	13892.6043	13892.6043	13892.6043	13892.6043	0.0
DIRECT	Worst	8	13965.2474	13965.2474	13965.2474	13965.2474	0.0
DIRECT	Worst	16	14768.9843	14768.9843	14768.9843	14768.9843	0.0
DIRECT	Average	2	10817.2496	10817.2496	10817.2496	10817.2496	0.0
DIRECT	Average	4	9061.4397	9061.4397	9061.4397	9061.4397	0.0
DIRECT	Average	8	11797.9532	11797.9532	11797.9532	11797.9532	0.0
DIRECT	Average	16	20899.982	20899.982	20899.982	20899.982	0.0
DIRECT	Best	2	9735.7598	9735.7598	9735.7598	9735.7598	0.0
DIRECT	Best	4	10749.1504	10749.1504	10749.1504	10749.1504	0.0
DIRECT	Best	8	15072.2732	15529.879	15511.5748	15529.879	91.5212
DIRECT	Best	16	11853.7632	14408.4017	13636.7862	20664.3989	1931.2616
MADS	None	1	12846.6039	40070.7244	40467.0784	72468.8334	14376.1973
MADS	Worst	2	1955.4537	6501.1593	6986.996	14234.5443	3586.6184
MADS	Worst	4	5372.4197	17278.8681	16548.6229	25467.0845	5545.1577
MADS	Worst	8	9912.9516	25795.3438	27652.7121	43365.432	9809.0424
MADS	Worst	16	12846.6039	40070.7244	40467.0784	72468.8334	14376.1973
MADS	Average	2	1322.7695	5835.8203	6836.3401	14385.2613	3603.0957
MADS	Average	4	4320.1206	12359.3161	12896.9993	32690.3856	6717.4893
MADS	Average	8	9079.2299	28733.0937	27947.8295	53565.9208	11492.8005
MADS	Average	16	12846.6039	42056.4008	42209.1506	64202.4352	12302.4779
MADS	Best	2	2027.6514	5704.5172	6817.5759	14221.5542	3927.697
MADS	Best	4	5607.293	16821.7138	18588.9696	32690.3856	7245.8841
MADS	Best	8	8709.6542	30091.2277	29047.4038	53565.9208	10748.1271
MADS	Best	16	12847.1789	41486.3927	42358.5673	69629.6459	14091.5518

Table 5.9: Statistics over the objective values of the best SFS-ranked solutions returned across different algorithms and splits. The standard deviation of DIRECT is generally zero due to it being fully deterministic, but note that it sometimes operates with splits that are slightly off when doing dynamic splits based on SHAP value estimation.

Starting out with the good news, partitioning often elicited better solutions than running without it, up to a certain point. table 5.10 shows how if we aggregate all runs by splits (from none = 1 up to 16), then partitioning is better for solution quality on average, up to 16 splits, at which point we might consider whether we have stretched the optimization budget for each subproblem too thin. Standard deviations are however generally high among runs, so there is a lot of inconsistency involved.

Split	Mean	Std
1	36445.2971	23384.0398
2	16903.5394	6026.4475
4	23218.0181	11401.2366
8	30943.4504	17524.4772
16	38684.8801	22795.162

Table 5.10: Benefits of partitioning, aggregated across all strategies, algorithms and runs.

As visualized on fig. 5.8, there are also quite dramatic discrepancies between algorithms, in terms of benefits from partitioning, with DIRECT in all but one case seeing inconsistent, minor benefits on solution quality, while MADS exhibits consistent, dramatic improvements on lower partitioning factors, with diminishing returns from higher split factors.

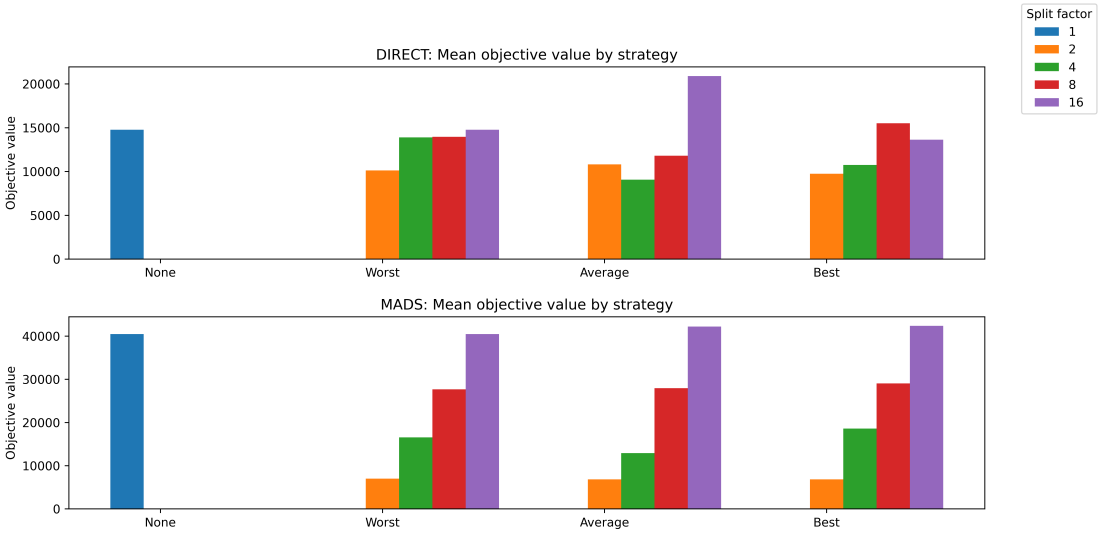

Figure 5.8: Mean objective values by optimizer, strategy and total split factor.

Based on what has been seen so far, the sweet spot seems to have been splitting into 2 partitions, which is also the option with the lowest standard deviation overall

(cf. table 5.10), and thus highest degree of consistency across conditions.

If we aggregate by each partitioning strategy and use t tests to compare objective values to running without any partitioning, then each strategy brings with it significant improvement when compared to running without any partitioning, with medium-size effects on solution quality (cf. table 5.11).

Strategy	**t(248)**	**p value**	**Cohen's d**
Worst	-4.2725	< 0.0001	0.63
Average	-4.5184	< 0.0001	0.66
Best	-4.6384	< 0.0001	0.66

Table 5.11: Effect on solution quality of each partitioning strategy, compared to running with None, comparing groups of samples with independent t tests. Cohen's d > 0.5 denotes a medium-size benefit across the board [54].

Now for the bad news for what might be considered our ulterior motives: It doesn't seem to matter what kind of partitioning strategy we use with our chosen problem-algorithm pairs. In fact, if we group solely by partitioning strategy, then the mean objective value of the worst-case split is actually a bit better than the "Best" one SHAPe-wise (cf. table 5.12).

Strategy	**Mean**	**Std**
None	27618.0313	16422.4223
Worst	18050.691	12069.6966
Average	17808.368	12982.7009
Best	18305.7235	13054.1222

Table 5.12: Benefits of splitting, aggregated across all algorithms, split factors and runs.

We can make this comparison more rigorous, using a one-way independent Analysis of Variance (ANOVA) test. This is basically like a t test, but it accounts for within and between-group variance, when we need to consider whether more than two samples are from the same population [54]. The ANOVA shows that there is no statistically significant difference between the aggregated results from either partitioning strategy by any commonly used significance threshold ($F(2, 597) = 0.0766, p = 0.9263$). So, they might just as well be a result of the same partitioning strategy - which they in a sense are: they all minimize the range perimeter of the search space.

If we aggregate total runtime by algorithm and whether they used SHAP value estimation or not, then we get the result that estimating the $n \cdot |LHS| = 20 \cdot 42 = 840$ Shapley values took about 20 seconds on average ($\mu = 19.2373, \sigma = 6.2861$). So, not a massive amount of extra time, but not really worth it either, as discussed.

If we look to the dynamic SHAP value estimation used in the best-case condition, then we see that it is actually quite accurate about selecting the right splits. Out of the $25 \cdot 4$ runs it needed to estimate SHAP values, all values were accurate enough for fully correct splits 84% of the time. Otherwise, estimates of the least important dimension was off by one or two places in the ranking. So e.g. at 16 splits, it would assign the final split to variable x_5 instead of x_4. The mean ranking error across these 16 runs was 1.125, with two runs displacing the lowest rank by two. Looking at the estimated SHAP values from a large sample, we see that the top values are all quite close to one another anyway (cf. table 5.8), so (at least in my estimation) a misplaced rank by one or two here and there doesn't account fully for, why the best-case splits did so "normally".

In conclusion, we see from this experiment that there may be consistent benefits to using search space partitioning with some problem-algorithm pairs, as long as the trial budget allows it, but did not demonstrate any particular benefit of using partitioning based on SHAP value estimation. Inferential statistics quickly dispelled these superstitions.

While we have not shown any benefits of SHAPe minimization splits, we have not shown any universal detriment either. It is possible that we just didn't find the right problem-algorithm pair to work with. But results nonetheless spawn the backpedaling notion that SHAP value estimation is perhaps best offered as a pre- and postprocessing facility in the library, to be used for its usual explanatory value, or possibly for variable elimination when the problem allows it.

Chapter 6

Conclusion and Future Work

This project took its onset in an observed lack of scalable, usable tools for data-intensive PA. The proposed solution, nipioacronymously named DIBBOlib, was defined under the problem statement: *How can a scalable and usable sBBO library for Spark be designed, implemented and tested?* (cf. chapter 1).

Since the developed library utilizes a cadre of methods that don't usually get the limelight, and certainly weren't known to me before this project, considerable effort was spent on understanding how they work, such that the design space of the library to be built around them was well understood (cf. 2). We observed how there are distinct types of sBBO methods within each quadrant of a proposed taxonomy, forming key design requirements for developing the library. General constraint handling, as well as how constraints are usually encountered within practical sBBO settings, furthermore formed some of the most difficult design challenges in the end.

In section 3.2 we moved on to provide an outline of how a library with the desired properties could be designed, including key architectural decisions and features. As for scalability, the main challenge beyond facilitating the efficient integration of sBBO with the Spark query engine, was to leverage any possible avenue for parallelism, to cope with load parameters specific to sBBO. Usability was worked with by using MLlib, and by extension Spark SQL, as a design analogy by proxy.

We then went ahead and outlined the implementation of DIBBOlib as is, describing how it basically just extends the MLlib Transformer class, being compatible with the pipeline system of MLlib. The pivotal *BlackBoxOptimizer* class offers a way of translating between a BBO problem model and a set of Spark SQL queries for solving said problem. For usability, the library offers an API matching exactly the one used in MLlib. Furthermore, it offers the additional option of using an algorithmic *wizard* API, mimicking vanilla Spark SQL queries, doing automatic algorithmic selection among options in the standard library suite, which implements

all the most commonly used sBBO algorithms.

An important core library feature is the one of *vertical transfer learning*, ensuring that no trial is left behind in our setting, where this would mean executing the same expensive Spark query yet another time (cf. section 3.4.3). TrialHistory's offer a flexible way of sharing information between optimization runs, and play a key role in supporting other library features.

Considerable effort was needed for implementing a library design for supporting *generally constrained optimization* (cf. section 3.5.5), interfacing with the Spark platform properly while ensuring extensibility with respect to existing approaches to constraint handling more or less specific to particular algorithms or constraint types. The implementation leveraged the internal Expression format of the Spark Column class, to rewrite user-specified constraints to quantifiable constraint function values and elide unnecessary Spark queries when possible.

To offer a better generic alternative to using classic penalty methods for users who would just like a plug-and-play option, the *Historical Revisionist Method* for general constraint handling was proposed (cf. section 3.5.6), forming a novel, data-driven approach to running with penalty functions - it essentially replaces the original objective function with a model fit to evaluation history so far, ensuring a ranking among points seen reflecting our priorities within constrained optimization.

The library also explored how various features built around *parallelism* could be supported, allowing for better scalability in settings in which there are available resources for it. First, the library offers the option of running several trials in parallel, when algorithms support it (cf. section 3.6.1). In this area, a novel dynamic load balancing strategy for SBO algorithms was proposed, based on directional sBBO. The library also offers the option of splitting the overall trial budget into several pieces, and do several optimization runs in parallel, using different random seeds or different search space partitions in separate subproblems (cf. section 3.6.2).

The proposed *heuristic search space partitioning strategies* run with a desired level of parallelism, as set by the user, possibly entirely in sequence (cf. section 3.7). The library core offers *dynamic* partitioning through a hybrid greedy/dynamic programming approach, minimizing perimetric objectives, based on overt search space geometry or estimated feature importance of problem dimensions, using Monte Carlo estimated Shapley values from game theory for the latter option.

The subsequent chapter then analyzed the merits of the library compared to adjacent solutions (cf. chapter 4). The library draws a lot of inspiration from existing solutions within PA systems, since the project took its onset in this setting. As for other BBO solutions for Spark, which are predominantly population-based, DIBBOlib can be said to form a separate niche, since sBBO and population-based BBO tend to be good for different problems. As for other solutions within sBBO, the library can be said to hold a unique spot, being the only *general-purpose* solution,

offering an extensive suite of algorithms and generic support for general constraint handling, that is furthermore designed for *data-intensive* sBBO workloads, unlike other frameworks, which seemingly assume compute-intensive ones.

The main focus of experiments (cf. chapter 5) was to assess the possible usefulness of 3 novel library features, which were variously designed with better scalability or usability in mind. Benefits of using the dynamic load balancing strategy, the new constraint handling method, and search space partitioning as opposed to none were found on example problems.

To conclude, DIBBOlib forms a novel proposal of how to support scalable, usable, and data-driven PA workflows in data-intensive settings, by using sBBO combined with the advanced analytics of the Spark ecosystem.

6.1 Future Work

There is no SW11 project, but I will still present a few thoughts about loose ends and hypothetical extensions to what was already accomplished.

An elephant in the room is of course to see if DIBBOlib is any good in one or more *real-world applications*, as opposed to the more or less artificial problems we have worked with so far to provide some breathing room for technical challenges (cf. section 1.3).

Implementing a *Python API* for the library would also be an obvious priority, since this is after all the client language of choice for Spark [88].

Providing support for *multi-objective optimization* is another possibly relevant extension for the library, since this was a notably absent feature from DIBBOlib, when comparing it to alternative solutions (cf. table 4.1). Supporting this would likely challenge some of the design assumption taken for given in the library so far - e.g., we might now need to work with *vectors* of objectives, the definition of optimality now becomes more complicated to handle from a user-centered standpoint [91], and individual algorithmic support for multi-objective optimization likely becomes a challenge.

A perhaps more surprising proposal from my end is actually to *ditch sBBO and Spark*, and transfer lessons learned from this project to a *single-node solution* for data-driven PA, using e.g. population-based BBO. We might for instance consider offering a similar kind of vertical transfer learning, HRM for data-driven constraint handling, search space partitioning with our dynamic/greedy approach, or the like. Such a library might want to leverage an efficient backend, like some Deep Learning libraries do with TensorFlow, scaling out computations as needed [38]. As observed by BBO researchers, there is currently a large focus within the field on making minuscule improvements to various benchmarks, as opposed to consolidating knowledge and developing domesticated solutions you would dare leave in the room with an unsuspecting user [126]. There is in other words a *general*

dearth of usable, scalable solutions for BBO, including single-node settings. Perhaps BBO research has reached a level of maturity, where it is time to focus on democratization as opposed to micro-optimization?

Bibliography

[1] *10 Gaussian Processes*. https://mc-stan.org/docs/stan-users-guide/gaussian-processes.html. Accessed: 2023-05-07.

[2] *1.7. Gaussian Processes*. https://scikit-learn.org/stable/modules/gaussian_process.html#gaussian-process-regression-gpr. Accessed: 2023-05-07.

[3] *A Conceptual Explanation of Bayesian Hyperparameter Optimization for Machine Learning*. https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f. Accessed: 2023-01-11.

[4] *AI Platform Vizier documentation*. https://cloud.google.com/ai-platform/optimizer/docs. Accessed: 2023-06-10.

[5] Jamil Al-Sawwa and Simone A Ludwig. "Parallel particle swarm optimization classification algorithm variant implemented with Apache Spark". In: *Concurrency and Computation: Practice and Experience* 32.2 (2020), e5451.

[6] Enrique Alba, José M Troya, et al. "A survey of parallel distributed genetic algorithms". In: *Complexity* 4.4 (1999), pp. 31–52.

[7] Mohammad Gh Alfailakawi, Maryam Aljame, and Imtiaz Ahmad. "Parallel and distributed implementation of sine cosine algorithm on apache spark platform". In: *IEEE Access* 9 (2021), pp. 77188–77202.

[8] *An introduction to Apache Hadoop for big data*. https://opensource.com/life/14/8/intro-apache-hadoop-big-data. Accessed: 2023-06-12.

[9] *Apache Arrow in PySpark*. https://spark.apache.org/docs/3.3.1/api/python/user_guide/sql/arrow_pandas.html. Accessed: 2023-01-06.

[10] Charles Audet and John E Dennis Jr. "Mesh adaptive direct search algorithms for constrained optimization". In: *SIAM Journal on optimization* 17.1 (2006), pp. 188–217.

[11] Charles Audet and Warren Hare. *Derivative-free and blackbox optimization*. Vol. 2. Springer, 2017.

[12] Charles Audet et al. “A surrogate-model-based method for constrained optimization”. In: *8th symposium on multidisciplinary analysis and optimization*. 2000, p. 4891.

[13] Charles Audet et al. “NOMAD version 4: Nonlinear optimization with the MADS algorithm”. In: *arXiv preprint arXiv:2104.11627* (2021).

[14] Tianyi Bai et al. “Transfer Learning for Bayesian Optimization: A Survey”. In: *arXiv preprint arXiv:2302.05927* (2023).

[15] Ishan Bajaj, Akhil Arora, and MM Hasan. “Black-Box Optimization: Methods and Applications”. In: *Black Box Optimization, Machine Learning, and No-Free Lunch Theorems*. Springer, 2021, pp. 35–65.

[16] Andrey Balmin, Thanos Papadimitriou, and Yannis Papakonstantinou. “Hypothetical queries in an olap environment”. In: *VLDB*. Vol. 220. 2000, p. 231.

[17] Cristóbal Barba-Gonzaléz et al. “Multi-objective big data optimization with jmetal and spark”. In: *International conference on evolutionary multi-criterion optimization*. Springer. 2017, pp. 16–30.

[18] Alexander Baur, Robert Klein, and Claudius Steinhardt. “Model-based decision support for optimal brochure pricing: applying advanced analytics in the tour operating industry”. In: *OR spectrum* 36.3 (2014), pp. 557–584.

[19] *bbotk GitHub*. `https://github.com/mlr-org/bbotk`. Accessed: 2023-06-12.

[20] *Benderopt GitHub*. `https://github.com/vthorey/benderopt`. Accessed: 2023-06-12.

[21] Pauline Bennet et al. “Nevergrad: black-box optimization platform”. In: *ACM SIGEVOlution* 14.1 (2021), pp. 8–15.

[22] David Benyon, Phil Turner, and Susan Turner. *Designing interactive systems: People, activities, contexts, technologies*. Pearson Education, 2005.

[23] Dimitris Bertsimas and Nathan Kallus. “From predictive to prescriptive analytics”. In: *Management Science* 66.3 (2020), pp. 1025–1044.

[24] *BFO docs*. `https://sites.google.com/site/bfocode/home?pli=1`. Accessed: 2023-06-12.

[25] Atharv Bhosekar and Marianthi Ierapetritou. “Advances in surrogate based modeling, feasibility analysis, and optimization: A review”. In: *Computers & Chemical Engineering* 108 (2018), pp. 250–267.

[26] *Big Data - GitHub Topic*. `https://github.com/topics/big-data`. Accessed: 2023-01-06.

[27] Mattias Björkman and Kenneth Holmström. “Global optimization of costly nonconvex functions using radial basis functions”. In: *Optimization and Engineering* 1 (2000), pp. 373–397.

[28] *BlaBoO GitHub*. https://github.com/kppeterkiss/BlackBoxOptimizer. Accessed: 2023-06-12.

[29] *blackbox GitHub*. https://github.com/paulknysh/blackbox. Accessed: 2023-06-12.

[30] Andrea Brilli, Giampaolo Liuzzi, and Stefano Lucidi. "An interior point method for nonlinear constrained derivative-free optimization". In: *arXiv preprint arXiv:2108.05157* (2021).

[31] Eric Brochu, Vlad M Cora, and Nando De Freitas. "A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning". In: *arXiv preprint arXiv:1012.2599* (2010).

[32] Rodney A Brooks. "Elephants don't play chess". In: *Robotics and autonomous systems* 6.1-2 (1990), pp. 3–15.

[33] Giovanni C Cattini. "Historical revisionism". In: *Transfer Journal of Contemporary Culture* 6 (2011), pp. 28–38.

[34] Alberto Ceselli et al. "Prescriptive analytics for MEC orchestration". In: *2018 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE. 2018, pp. 1–9.

[35] Bill Chambers and Matei Zaharia. *Spark: The definitive guide: Big data processing made simple*. " O'Reilly Media, Inc.", 2018.

[36] Hongwei Chen et al. "A spark-based distributed whale optimization algorithm for feature selection". In: *2019 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*. Vol. 1. IEEE. 2019, pp. 70–74.

[37] Fatemeh Cheraghchi, Arash Iranzad, and Bijan Raahemi. "Subspace selection in high-dimensional big data using genetic algorithm in apache spark". In: *Proceedings of the Second International Conference on Internet of things, Data and Cloud Computing*. 2017, pp. 1–7.

[38] Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2021.

[39] *Class AtomicInteger*. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html. Accessed: 2023-05-07.

[40] *Class Column*. https://spark.apache.org/docs/1.6.1/api/java/org/apache/spark/sql/Column.html. Accessed: 2023-05-07.

[41] *Classification and regression*. https://spark.apache.org/docs/latest/ml-classification-regression.html#decision-tree-classifier. Accessed: 2023-01-06.

[42] Andrew R Conn, Katya Scheinberg, and Luis N Vicente. *Introduction to derivative-free optimization*. SIAM, 2009.

[43] Aldina Correia et al. “Classification of some penalty methods”. In: *Integral Methods in Science and Engineering, Volume 2: Computational Aspects* (2010), pp. 131–140.

[44] Aldina Correia et al. “Direct-search penalty/barrier methods”. In: *Proceedings of the World Congress on Engineering 2010*. Vol. 3. International Association of Engineers. 2010, pp. 1729–1734.

[45] Federico Croppi. “Explaining Sequential Model-Based Optimization”. PhD thesis. 2021.

[46] *DAKOTA docs*. `https://dakota.sandia.gov/`. Accessed: 2023-06-12.

[47] Chris J Date. *SQL and relational theory: how to write accurate SQL code*. " O'Reilly Media, Inc.", 2011.

[48] *DFL docs*. `http://www.iasi.cnr.it/~liuzzi/DFL/`. Accessed: 2023-06-12.

[49] Sébastien Le Digabel and Stefan M Wild. “A taxonomy of constraints in simulation-based optimization”. In: *arXiv preprint arXiv:1505.07881* (2015).

[50] *Distribution of Executors, Cores and Memory for a Spark Application running in Yarn*. `https://spoddutur.github.io/spark-notes/distribution_of_executors_cores_and_memory_for_spark_application.html`. Accessed: 2023-01-06.

[51] David Eriksson, David Bindel, and Christine A Shoemaker. “pySOT and POAP: An event-driven asynchronous framework for surrogate optimization”. In: *arXiv preprint arXiv:1908.00420* (2019).

[52] *Executing Spark code with expr and eval*. `https://mungingdata.com/apache-spark/expr-eval/`. Accessed: 2023-05-07.

[53] *FACT CHECK: DID ABRAHAM LINCOLN SAY, 'THE BEST WAY TO PREDICT THE FUTURE IS TO CREATE IT'?* `https://checkyourfact.com/2019/07/24/fact-check-abraham-lincoln-best-way-predict-future-create/`. Accessed: 2023-06-10.

[54] Andy Field. *Discovering statistics using IBM SPSS statistics*. sage, 2013.

[55] Roger Fletcher and Sven Leyffer. “Nonlinear programming without a penalty function”. In: *Mathematical programming* 91 (2002), pp. 239–269.

[56] *fminsearch Algorithm*. `https://www.mathworks.com/help/optim/ug/fminsearch-algorithm.html`. Accessed: 2023-05-07.

[57] Alexander IJ Forrester and Andy J Keane. “Recent advances in surrogate-based optimization”. In: *Progress in aerospace sciences* 45.1-3 (2009), pp. 50–79.

[58] Davide Frazzetto et al. “Prescriptive analytics: a survey of emerging trends and technologies”. In: *The VLDB Journal* 28.4 (2019), pp. 575–595.

[59] Daniel Gartner, Elizabeth M Williams, and Paul R Harper. "Prescriptive healthcare analytics: a tutorial on discrete optimization and simulation". In: *2022 IEEE 10th International Conference on Healthcare Informatics (ICHI)*. IEEE. 2022, pp. 01–03.

[60] Sneha Gathani et al. "Predictive and Prescriptive Analytics in Business Decision Making: Needs and Concerns". In: *arXiv preprint arXiv:2212.13643* (2022).

[61] *GFO docs*. `https://github.com/SimonBlanke/Gradient-Free-Optimizers`. Accessed: 2023-06-12.

[62] *GloMPO GitHub*. `https://github.com/mfgustavo/glompo`. Accessed: 2023-06-12.

[63] Daniel Golovin et al. "Google vizier: A service for black-box optimization". In: *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 2017, pp. 1487–1495.

[64] Aanchal Goyal et al. "Asset health management using predictive and prescriptive analytics for the electric power grid". In: *IBM Journal of Research and Development* 60.1 (2016), pp. 4–1.

[65] Stewart Greenhill et al. "Bayesian optimization for adaptive experimental design: A review". In: *IEEE access* 8 (2020), pp. 13937–13948.

[66] Shantanu Gupta, Rajiv Tiwari, and Shivashankar B Nair. "Multi-objective design optimisation of rolling bearings using genetic algorithms". In: *Mechanism and Machine Theory* 42.10 (2007), pp. 1418–1443.

[67] H-M Gutmann. "A radial basis function method for global optimization". In: *Journal of global optimization* 19.3 (2001), pp. 201–227.

[68] Peter J Haas et al. "Data is dead... without what-if models". In: *Proceedings of the VLDB Endowment* 4.12 (2011), pp. 1486–1489.

[69] *Hacker New - We're all consenting adults*. `https://news.ycombinator.com/item?id=13292682`. Accessed: 2023-05-07.

[70] Raphael T Haftka, Diane Villanueva, and Anirban Chaudhuri. "Parallel surrogate-assisted global optimization with expensive functions–a survey". In: *Structural and Multidisciplinary Optimization* 54 (2016), pp. 3–13.

[71] Jiawei Han, Jian Pei, and Hanghang Tong. *Data mining: concepts and techniques*. Morgan kaufmann, 2022.

[72] Zhong-Hua Han, Ke-Shi Zhang, et al. "Surrogate-based optimization". In: *Real-world applications of genetic algorithms* 343 (2012).

[73] Haripriya Harikumar et al. "Prescriptive analytics through constrained Bayesian optimization". In: *Pacific-Asia conference on knowledge discovery and data mining*. Springer. 2018, pp. 335–347.

[74] Xin He, Kaiyong Zhao, and Xiaowen Chu. "AutoML: A survey of the state-of-the-art". In: *Knowledge-Based Systems* 212 (2021), p. 106622.

[75] Zhihui He et al. "A Spark-based differential evolution with grouping topology model for large-scale global optimization". In: *Cluster Computing* 24.1 (2021), pp. 515–535.

[76] Abdel-Rahman Hedar, Masao Fukushima, et al. "Derivative-free filter simulated annealing method for constrained continuous global optimization". In: *Journal of Global optimization* 35.4 (2006), pp. 521–550.

[77] Michael Hellwig and Hans-Georg Beyer. "Benchmarking evolutionary algorithms for single objective real-valued constrained optimization–a critical review". In: *Swarm and evolutionary computation* 44 (2019), pp. 927–944.

[78] Yu-Chi Ho and David L Pepyne. "Simple explanation of the no-free-lunch theorem and its implications". In: *Journal of optimization theory and applications* 115 (2002), pp. 549–570.

[79] *Honeycomb (geometry)*. https://en.wikipedia.org/wiki/Honeycomb_(geometry). Accessed: 2023-05-07.

[80] *HOPSPACK GitHub*. https://www.osti.gov/servlets/purl/1130394/. Accessed: 2023-06-12.

[81] Kerstin Hötte. "Demand-pull, technology-push, and the direction of technological change". In: *Research Policy* 52.5 (2023), p. 104740. ISSN: 0048-7333. DOI: https://doi.org/10.1016/j.respol.2023.104740. URL: https://www.sciencedirect.com/science/article/pii/S0048733323000240.

[82] *Infeasibility*. https://oss-vizier.readthedocs.io/en/latest/guides/user/search_spaces.html#infeasibility. Accessed: 2023-06-10.

[83] Piotr Jedrzejowicz and Izabela Wierzbowska. "Apache spark as a tool for parallel population-based optimization". In: *Intelligent Decision Technologies 2019*. Springer, 2020, pp. 181–190.

[84] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme programming installed*. Addison-Wesley Professional, 2001.

[85] Yuan Jin, S Joe Qin, and Qiang Huang. "Prescriptive analytics for understanding of out-of-plane deformation in additive manufacturing". In: *2016 IEEE International Conference on Automation Science and Engineering (CASE)*. IEEE. 2016, pp. 786–791.

[86] Donald R. Jones. "Direct global optimization algorithm: Direct Global Optimization Algorithm". In: *Encyclopedia of Optimization*. Ed. by Christodoulos A. Floudas and Panos M. Pardalos. Boston, MA: Springer US, 2001, pp. 431–440. ISBN: 978-0-306-48332-5. DOI: 10.1007/0-306-48332-7_93. URL: https://doi.org/10.1007/0-306-48332-7_93.

[87] Donald R Jones, Cary D Perttunen, and Bruce E Stuckman. "Lipschitzian optimization without the Lipschitz constant". In: *Journal of optimization Theory and Applications* 79 (1993), pp. 157–181.

[88] Holden Karau et al. *Learning spark: lightning-fast big data analysis*. " O'Reilly Media, Inc.", 2015.

[89] Ban Kawas et al. "Prescriptive analytics for allocating sales teams to opportunities". In: *2013 IEEE 13th International Conference on Data Mining Workshops*. IEEE. 2013, pp. 211–218.

[90] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.

[91] Mykel J Kochenderfer and Tim A Wheeler. *Algorithms for optimization*. Mit Press, 2019.

[92] Slawomir Koziel, David Echeverría Ciaurri, and Leifur Leifsson. "Surrogate-based methods". In: *Computational optimization, methods and algorithms* (2011), pp. 33–59.

[93] Slawomir Koziel and Xin-She Yang. *Computational optimization, methods and algorithms*. Vol. 356. Springer, 2011.

[94] Tipaluck Krityakierne, Taimoor Akhtar, and Christine A Shoemaker. "SOP: parallel surrogate global optimization with Pareto center selection for computationally expensive single objective problems". In: *Journal of Global Optimization* 66 (2016), pp. 417–437.

[95] Abhishek Kumar et al. "A test-suite of non-convex constrained optimization problems from the real-world and some baseline results". In: *Swarm and Evolutionary Computation* 56 (2020), p. 100693.

[96] Laks VS Lakshmanan, Alex Russakovsky, and Vaishnavi Sashikanth. "What-if OLAP queries with changing dimensions". In: *2008 IEEE 24th International Conference on Data Engineering*. IEEE. 2008, pp. 1334–1336.

[97] *LAPACK - Linear Algebra PACKage*. `https://www.netlib.org/lapack/`. Accessed: 2023-06-10.

[98] Jeffrey Larson, Matt Menickelly, and Stefan M Wild. "Derivative-free optimization methods". In: *Acta Numerica* 28 (2019), pp. 287–404.

[99] Katerina Lepenioti et al. "Machine learning for predictive and prescriptive analytics of operational data in smart manufacturing". In: *International Conference on Advanced Information Systems Engineering*. Springer. 2020, pp. 5–16.

[100] Katerina Lepenioti et al. "Prescriptive analytics: Literature review and research challenges". In: *International Journal of Information Management* 50 (2020), pp. 57–70.

[101] Yang Li et al. "Openbox: A generalized black-box optimization service". In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 2021, pp. 3209–3219.

[102] David G Luenberger et al. "Penalty and barrier methods". In: *Linear and Nonlinear Programming* (2016), pp. 397–428.

[103] Scott M Lundberg, Gabriel G Erion, and Su-In Lee. "Consistent individualized feature attribution for tree ensembles". In: *arXiv preprint arXiv:1802.03888* (2018).

[104] Scott M Lundberg and Su-In Lee. "A unified approach to interpreting model predictions". In: *Advances in neural information processing systems* 30 (2017).

[105] Gabriel Luque and Enrique Alba. *Parallel genetic algorithms: Theory and real world applications*. Vol. 367. Springer, 2011.

[106] Irv Lustig et al. "The analytics journey". In: *Analytics Magazine* 3.6 (2010), pp. 11–13.

[107] Sedigheh Mahdavi, Mohammad Ebrahim Shiri, and Shahryar Rahnamayan. "Cooperative co-evolution with a new decomposition method for large-scale optimization". In: *2014 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2014, pp. 1285–1292.

[108] Fahad Maqbool et al. "Large Scale Distributed Optimization using Apache Spark: Distributed Scalable Shade-Bat (DistSSB)". In: *2021 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2021, pp. 2559–2566.

[109] Fahad Maqbool et al. "Scalable distributed genetic algorithm using apache spark (s-ga)". In: *International conference on intelligent computing*. Springer. 2019, pp. 424–435.

[110] Alexandra Meliou. "The Power of How-To Queries". In: ().

[111] Alexandra Meliou, Wolfgang Gatterbauer, and Dan Suciu. "Reverse data management". In: *Proceedings of the VLDB Endowment* 4.12 (2011), pp. 1490–1493.

[112] Alexandra Meliou and Dan Suciu. "Tiresias: the database oracle for how-to queries". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012, pp. 337–348.

[113] *MISO docs*. `https://optimization.lbl.gov/downloads#h.p_BjSaeAORU9gm`. Accessed: 2023-06-12.

[114] *mlrMBO GitHub*. `https://github.com/mlr-org/mlrMBO`. Accessed: 2023-06-12.

[115] Martin Moesmann. *Data-Driven Prescriptive Analytics by Data-Intensive Black-Box Optimization with Apache Spark*. Aalborg University, 2023.

[116] Marcin Molga and Czesław Smutnicki. "Test functions for optimization needs". In: *Test functions for optimization needs* 101 (2005), p. 48.

[117] Christoph Molnar. *Interpretable machine learning*. Lulu. com, 2020.

[118] *Monitoring and Instrumentation*. `https://spark.apache.org/docs/3.3.1/monitoring.html#metrics`. Accessed: 2023-01-06.

[119] Maliki Moustapha and Bruno Sudret. "Surrogate-assisted reliability-based design optimization: a survey and a unified modular framework". In: *Structural and Multidisciplinary Optimization* 60 (2019), pp. 2157–2176.

[120] Juliane Müller. "MISO: mixed-integer surrogate optimization framework". In: *Optimization and Engineering* 17 (2016), pp. 177–203.

[121] *mystic: constrained nonlinear optimization for scientific machine learning, UQ, and AI*. `https://mystic.readthedocs.io/en/latest/`. Accessed: 2023-05-06.

[122] *Nevergrad - A gradient-free optimization platform*. `https://facebookresearch.github.io/nevergrad/`. Accessed: 2023-06-10.

[123] *Nevergrad - A gradient-free optimization platform*. `https://facebookresearch.github.io/nevergrad/optimizers_ref.html#nevergrad.optimizers.base.Optimizer.tell`. Accessed: 2023-06-10.

[124] *NOMAD*. `https://github.com/bbopt/nomad`. Accessed: 2023-01-06.

[125] M Omidvar, Xiaodong Li, and Xin Yao. "A review of population-based metaheuristics for large-scale black-box global optimization: Part A". In: *IEEE Transactions on Evolutionary Computation* (2021).

[126] M Omidvar, Xiaodong Li, and Xin Yao. "A review of population-based metaheuristics for large-scale black-box global optimization: Part B". In: *IEEE Transactions on Evolutionary Computation* (2021).

[127] *OpenBox: Generalized and Efficient Blackbox Optimization System*. `https://open-box.readthedocs.io/en/latest/`. Accessed: 2023-06-10.

[128] *optim GitHub*. `https://github.com/kthohr/optim`. Accessed: 2023-06-12.

[129] *Optimization and root finding (scipy.optimize)*. `https://docs.scipy.org/doc/scipy/reference/optimize.html`. Accessed: 2023-05-06.

[130] *Optimization.jl GitHub*. `https://github.com/SciML/Optimization.jl`. Accessed: 2023-06-12.

[131] *Optim.jl GitHub*. `https://github.com/JuliaNLSolvers/Optim.jl`. Accessed: 2023-06-12.

[132] *org.apache.spark.sql.catalog.Catalog*. `https://spark.apache.org/docs/latest/api/scala/org/apache/spark/sql/catalog/Catalog.html`. Accessed: 2023-05-07.

[133] *P-N-Suganthan/2020-RW-Constrained-Optimisation*. https://github.com/P-N-Suganthan/2020-RW-Constrained-Optimisation. Accessed: 2023-06-10.

[134] Ciprian Paduraru, Marius-Constantin Melemciuc, and Alin Stefanescu. "A distributed implementation using apache spark of a genetic algorithm applied to test data generation". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2017, pp. 1857–1863.

[135] Jayesh Patel. "The democratization of machine learning features". In: *2020 IEEE 21st International Conference on Information Reuse and Integration for Data Science (IRI)*. IEEE. 2020, pp. 136–141.

[136] Nikolaos Ploskas and Nikolaos V Sahinidis. "Review and comparison of algorithms and software for mixed-integer derivative-free optimization". In: *Journal of Global Optimization* (2021), pp. 1–30.

[137] Tony Pourmohamad and Herbert KH Lee. "Bayesian optimization via barrier functions". In: *Journal of Computational and Graphical Statistics* 31.1 (2022), pp. 74–83.

[138] Tony Pourmohamad and Herbert KH Lee. "The statistical filter approach to constrained optimization". In: *Technometrics* 62.3 (2020), pp. 303–312.

[139] William H Press et al. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[140] *PyBrain docs*. http://pybrain.org/docs/tutorial/optimization.html. Accessed: 2023-06-12.

[141] *pySOT: Surrogate Optimization Toolbox for Python*. https://github.com/dme65/pySOT. Accessed: 2023-05-06.

[142] Nestor V Queipo et al. "Surrogate-based analysis and optimization". In: *Progress in aerospace sciences* 41.1 (2005), pp. 1–28.

[143] Gomathy Ramaswami, Teo Susnjak, and Anuradha Mathrani. "Supporting Students' Academic Performance Using Explainable Machine Learning with Automated Prescriptive Analytics". In: *Big Data and Cognitive Computing* 6.4 (2022), p. 105.

[144] *RBFOpt GitHub*. https://github.com/coin-or/rbfopt. Accessed: 2023-06-12.

[145] Rommel G Regis. "A survey of surrogate approaches for expensive constrained black-box optimization". In: *Optimization of Complex Systems: Theory, Models, Algorithms and Applications*. Springer. 2020, pp. 37–47.

[146] Rommel G Regis. "Stochastic radial basis function algorithms for large-scale optimization involving expensive black-box objective and constraint functions". In: *Computers & Operations Research* 38.5 (2011), pp. 837–853.

[147] Rommel G Regis and Christine A Shoemaker. "A stochastic radial basis function method for the global optimization of expensive functions". In: *INFORMS Journal on Computing* 19.4 (2007), pp. 497–509.

[148] Rommel G Regis and Christine A Shoemaker. "Combining radial basis function surrogates and dynamic coordinate search in high-dimensional expensive black-box optimization". In: *Engineering Optimization* 45.5 (2013), pp. 529–555.

[149] Rommel G Regis and Christine A Shoemaker. "Parallel stochastic global optimization using radial basis functions". In: *INFORMS Journal on Computing* 21.3 (2009), pp. 411–426.

[150] Luis Miguel Rios and Nikolaos V Sahinidis. "Derivative-free optimization: a review of algorithms and comparison of software implementations". In: *Journal of Global Optimization* 56 (2013), pp. 1247–1293.

[151] *RoBo docs*. `https://github.com/automl/RoBO`. Accessed: 2023-06-12.

[152] *scalanlp/breeze*. `https://github.com/scalanlp/breeze`. Accessed: 2023-05-07.

[153] *scikit-optimize docs*. `https://scikit-optimize.github.io/stable/`. Accessed: 2023-06-12.

[154] *scikit-optimize: Sequential model-based optimization in Python*. `https://scikit-optimize.github.io/stable/`. Accessed: 2023-05-06.

[155] Songqing Shan and G Gary Wang. "Metamodeling for high dimensional simulation-based design problems". In: (2010).

[156] *SHAP: A game theoretic approach to explain the output of any machine learning model.* `https://github.com/slundberg/shap`. Accessed: 2023-05-06.

[157] Renhe Shi et al. "Filter-based adaptive Kriging method for black-box optimization problems with expensive objective and constraints". In: *Computer Methods in Applied Mechanics and Engineering* 347 (2019), pp. 782–805.

[158] Laurynas Siksnys and Torben Bach Pedersen. "Prescriptive analytics". In: *Encyclopedia of database systems*. Springer, 2018.

[159] Laurynas Šikšnys and Torben Bach Pedersen. "Solvedb: Integrating optimization problem solvers into sql databases". In: *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*. 2016, pp. 1–12.

[160] Laurynas Siksnys et al. "SolveDB+: SQL-Based Prescriptive Analytics." In: *EDBT*. 2021, pp. 133–144.

[161] Michael Sipser. "Introduction to the Theory of Computation". In: *ACM Sigact News* 27.1 (1996), pp. 27–29.

[162] *smile.regression.gpr*. https://haifengl.github.io/api/scala/smile/regression/package\protect\T1\textdollar\protect\T1\textdollargpr\protect\T1\textdollar.html. Accessed: 2023-05-07.

[163] *SNOBFIT docs*. https://arnold-neumaier.at/software/snobfit/index.html. Accessed: 2023-06-12.

[164] *SolveDB: Integrating Optimization Problem Solvers and Prescriptive Analytics Into SQL Databases*. https://www.daisy.aau.dk/projects/solvedb-integrating-optimization-problem-solvers-into-sql-databases//. Accessed: 2022-18-10.

[165] *[SPARK-28702] Display useful error message (instead of NPE) for invalid Dataset operations (e.g. calling actions inside of transformations)*. https://issues.apache.org/jira/browse/SPARK-28702. Accessed: 2023-01-06.

[166] *Spark Docs - Cluster Mode Overview*. https://spark.apache.org/docs/3.3.1/cluster-overview.html. Accessed: 2023-01-06.

[167] *Spark Docs - Machine Learning Library (MLlib) Guide*. https://spark.apache.org/docs/3.3.1/ml-guide.html. Accessed: 2023-01-06.

[168] *Spark Docs - ML Pipelines*. https://spark.apache.org/docs/3.3.1/ml-pipeline.html. Accessed: 2023-01-06.

[169] *Spark Docs - Overview*. https://spark.apache.org/docs/3.3.1/. Accessed: 2023-01-06.

[170] *Spark Docs - RDD Programming Guide*. https://spark.apache.org/docs/3.3.1/rdd-programming-guide.html. Accessed: 2023-01-06.

[171] *Spark Docs - Scalar User Defined Functions (UDFs)*. https://spark.apache.org/docs/3.3.1/sql-ref-functions-udf-scalar.html. Accessed: 2023-01-06.

[172] *Spark Docs - Spark SQL, DataFrames and Datasets Guide*. https://spark.apache.org/docs/3.3.1/sql-programming-guide.html. Accessed: 2023-01-06.

[173] *Spark Project ML Library*. https://mvnrepository.com/artifact/org.apache.spark/spark-mllib. Accessed: 2023-05-07.

[174] *Spark Scala API - SQLTransformer*. https://spark.apache.org/docs/3.3.1/api/scala/org/apache/spark/ml/feature/SQLTransformer.html. Accessed: 2023-01-06.

[175] *Spark Scala API - Transformer*. https://spark.apache.org/docs/3.3.1/api/scala/org/apache/spark/ml/Transformer.html. Accessed: 2023-01-06.

[176] *spark-sql-perf*. https://github.com/databricks/spark-sql-perf. Accessed: 2023-01-06.

[177] *Spark Standalone Mode*. https://spark.apache.org/docs/3.3.1/spark-standalone.html. Accessed: 2023-01-06.

[178] *SRBFStrategy*. https://pysot.readthedocs.io/en/latest/options.html#srbfstrategy. Accessed: 2023-05-07.

[179] Sharan Srinivas and A Ravi Ravindran. "Optimizing outpatient appointment system using machine learning algorithms and scheduling rules: a prescriptive analytics framework". In: *Expert Systems with Applications* 102 (2018), pp. 245–261.

[180] *Stack Overflow Developer Survey 2022*. https://survey.stackoverflow.co/2022/. Accessed: 2023-01-06.

[181] *Stopping Monte Carlo simulation once certain convergence level is reached*. https://quant.stackexchange.com/questions/21764/stopping-monte-carlo-simulation-once-certain-convergence-level-is-reached/21769#21769. Accessed: 2023-05-07.

[182] Erik Strumbelj and Igor Kononenko. "An efficient explanation of individual classifications using game theory". In: *The Journal of Machine Learning Research* 11 (2010), pp. 1–18.

[183] Erik Štrumbelj and Igor Kononenko. "Explaining prediction models and individual predictions with feature contributions". In: *Knowledge and information systems* 41 (2014), pp. 647–665.

[184] *Surrogates.jl: Surrogate models and optimization for scientific machine learning*. https://docs.sciml.ai/Surrogates/stable/. Accessed: 2023-05-07.

[185] Teo Susnjak. "A Prescriptive Learning Analytics Framework: Beyond Predictive Modelling and onto Explainable AI with Prescriptive Analytics". In: *arXiv preprint arXiv:2208.14582* (2022).

[186] Ke Tang et al. "Benchmark functions for the CEC'2008 special session and competition on large scale global optimization". In: *Nature inspired computation and applications laboratory, USTC, China* 24 (2007), pp. 1–18.

[187] *TOMLAB docs*. https://tomopt.com/tomlab/optimization/costly.php. Accessed: 2023-06-12.

[188] *TOMLAB: For fast and robust large-scale optimization in MATLAB*. https://tomopt.com/. Accessed: 2023-06-10.

[189] *TPC BENCHMARK™ DS Standard Specification*. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v3.2.0.pdf. Accessed: 2023-01-06.

[190] *TPC-DS Version 2 and Version 3*. https://www.tpc.org/tpcds/default5.asp. Accessed: 2023-01-06.

[191] Diane Villanueva et al. "Locating multiple candidate designs with surrogate-based optimization". In: *10th World Congress on Structural and Multidisciplinary Optimization, Orlando, FL, USA*. 2013.

[192] Johannes Kunze Von Bischhoffshausen et al. "An information system for sales team assignments utilizing predictive and prescriptive analytics". In: *2015 IEEE 17th Conference on Business Informatics*. Vol. 1. IEEE. 2015, pp. 68–76.

[193] Walter D Wallis and John C George. *Introduction to combinatorics*. CRC press, 2016.

[194] G Gary Wang, Zuomin Dong, Peter Aitchison, et al. "Adaptive response surface method-a global optimization scheme for approximation-based design problems". In: *Engineering Optimization* 33.6 (2001), pp. 707–734.

[195] Stefan M Wild, Rommel G Regis, and Christine A Shoemaker. "ORBIT: Optimization by radial basis function interpolation in trust-regions". In: *SIAM Journal on Scientific Computing* 30.6 (2008), pp. 3197–3219.

[196] Eyal Winter. "The shapley value". In: *Handbook of game theory with economic applications* 3 (2002), pp. 2025–2054.

[197] David H Wolpert and William G Macready. "No free lunch theorems for optimization". In: *IEEE transactions on evolutionary computation* 1.1 (1997), pp. 67–82.

[198] David H Wolpert, William G Macready, et al. *No free lunch theorems for search*. Tech. rep. Citeseer, 1995.

[199] Congmin Yang et al. "Parallel Particle Swarm Optimization Based on Spark for Academic Paper Co-Authorship Prediction". In: *Information* 12.12 (2021), p. 530.

[200] Pengcheng Ye. "A review on surrogate-based global optimization methods for computationally expensive functions". In: *Softw. Eng* 7 (2019), pp. 68–84.

[201] Wei Zhao et al. "Shap values for explaining cnn-based text classification models". In: *arXiv preprint arXiv:2008.11825* (2020).

Appendix A

Appendix

A.1 Experiments

This section contains all raw experimental data, insofar that this can be sensibly presented.

A.1.1 Experiment 1

Modified TPC-DS Q48

The mildly modified version of Q48 from the TPC-DS benchmark suite [189]. *__THIS__* (alias *sol*) on line 2 denotes the input table, being the large *store_sales* fact table with the instantiated decision variables projected unto it. Decision variables (capitalized here) are named like their substitution parameters in the original query. Only one original parameter, the *d_year* on line 5 was was provided a fixed value (originally done for story purposes [115]).

```
1  select *
2  from store, customer_demographics, customer_address, date_dim,
       __THIS__ sol
3  where s_store_sk = ss_store_sk
4    and ss_sold_date_sk = d_date_sk
5    and d_year = 2000
6    and ((cd_demo_sk = ss_cdemo_sk
7      and cd_marital_status = sol.MS1
8      and cd_education_status = sol.ES1
9      and ss_sales_price between 100.00 and 150.00)
10     or
11     (cd_demo_sk = ss_cdemo_sk
12     and cd_marital_status = sol.MS2
13     and cd_education_status = sol.ES2
14     and ss_sales_price between 50.00 and 100.00)
```

```
    or
    (cd_demo_sk = ss_cdemo_sk
    and cd_marital_status = sol.MS3
    and cd_education_status = sol.ES3
    and ss_sales_price between 150.00 and 200.00))
  and ((ss_addr_sk = ca_address_sk
    and ca_country = 'United States'
    and ca_state = sol.STATE1
    and ss_net_profit between 0 and 2000)
    or
    (ss_addr_sk = ca_address_sk
    and ca_country = 'United States'
    and ca_state = sol.STATE2
    and ss_net_profit between 150 and 3000)
    or
    (ss_addr_sk = ca_address_sk
    and ca_country = 'United States'
    and ca_state = sol.STATE3
    and ss_net_profit between 50 and 25000));
```

Run data

Results from all runs. To save space here, so to speak, only the mean and standard deviation of all timesteps are provided for driver memory usage.

Strategy	Scale	Executors	Run	Runtime	Memory Mean	Memory Std
Sequential	1	3	1	1247	1.015	0.2653
Sequential	1	3	2	1253	1.0272	0.2704
Sequential	1	3	3	1215	0.8406	0.2479
Sequential	1	3	4	1284	0.9736	0.2633
Sequential	1	3	5	1256	1.1085	0.3018
Sequential	1	6	1	1246	0.8438	0.2432
Sequential	1	6	2	1232	1.236	0.5616
Sequential	1	6	3	1183	0.9111	0.2559
Sequential	1	6	4	1194	0.915	0.2823
Sequential	1	6	5	1244	0.8921	0.2485
Sequential	1	9	1	1156	0.99	0.2743
Sequential	1	9	2	1272	0.8951	0.2479
Sequential	1	9	3	1233	0.9014	0.245
Sequential	1	9	4	1320	0.9031	0.2454
Sequential	1	9	5	1202	0.9241	0.2636
Sequential	1	12	1	1217	0.931	0.2798
Sequential	1	12	2	1215	0.936	0.2881

Sequential	1	12	3	1184	0.9997	0.2585
Sequential	1	12	4	1222	0.9085	0.2617
Sequential	1	12	5	1132	0.976	0.2691
Sequential	1	15	1	1101	0.9052	0.2674
Sequential	1	15	2	1530	0.8849	0.2235
Sequential	1	15	3	1642	0.833	0.2431
Sequential	1	15	4	1589	0.8897	0.2262
Sequential	1	15	5	1649	0.9991	0.2261
Sequential	1	18	1	1643	0.9924	0.229
Sequential	1	18	2	1643	1.1053	0.3922
Sequential	1	18	3	1632	0.9032	0.2306
Sequential	1	18	4	1667	0.8982	0.2246
Sequential	1	18	5	1618	0.7528	0.1712
Sequential	2	3	1	1430	0.8015	0.1919
Sequential	2	3	2	1420	0.8254	0.2277
Sequential	2	3	3	1296	0.9291	0.2649
Sequential	2	3	4	1307	0.9569	0.2457
Sequential	2	3	5	1298	0.9995	0.3219
Sequential	2	6	1	1188	0.9076	0.2637
Sequential	2	6	2	1296	0.8967	0.2475
Sequential	2	6	3	1678	0.8956	0.2241
Sequential	2	6	4	1207	1.0455	0.3043
Sequential	2	6	5	1270	0.8947	0.2487
Sequential	2	9	1	1259	0.9693	0.2757
Sequential	2	9	2	1677	0.9591	0.2404
Sequential	2	9	3	1264	0.8419	0.2533
Sequential	2	9	4	1290	0.8945	0.2919
Sequential	2	9	5	1148	0.9096	0.2635
Sequential	2	12	1	1262	0.9386	0.2633
Sequential	2	12	2	1228	0.9611	0.2851
Sequential	2	12	3	1163	0.9445	0.3082
Sequential	2	12	4	1264	1.1915	0.5193
Sequential	2	12	5	1251	0.8973	0.2588
Sequential	2	15	1	1175	1.3991	0.458
Sequential	2	15	2	1254	0.9288	0.29
Sequential	2	15	3	1214	1.0237	0.3039
Sequential	2	15	4	1685	0.8912	0.2255
Sequential	2	15	5	1645	0.8978	0.2254
Sequential	2	18	1	1668	0.9733	0.2333

Sequential	2	18	2	1679	0.9981	0.2249
Sequential	2	18	3	1141	0.958	0.3271
Sequential	2	18	4	1212	0.9082	0.2664
Sequential	2	18	5	1224	0.9024	0.2614
Sequential	4	3	1	1409	0.9347	0.2726
Sequential	4	3	2	1380	1.0308	0.2733
Sequential	4	3	3	1370	1.0482	0.4726
Sequential	4	3	4	1393	1.0048	0.2773
Sequential	4	3	5	1417	0.9894	0.2729
Sequential	4	6	1	1304	1.0592	0.2825
Sequential	4	6	2	1246	0.9797	0.2956
Sequential	4	6	3	1372	0.9164	0.2534
Sequential	4	6	4	1317	1.0003	0.2776
Sequential	4	6	5	1724	0.9706	0.2369
Sequential	4	9	1	1251	0.9501	0.2895
Sequential	4	9	2	1243	1.0253	0.2801
Sequential	4	9	3	1325	0.8963	0.2492
Sequential	4	9	4	1317	0.9134	0.2546
Sequential	4	9	5	1384	1.2488	0.363
Sequential	4	12	1	1680	0.8932	0.226
Sequential	4	12	2	1277	1.0615	0.3422
Sequential	4	12	3	1262	0.8161	0.2663
Sequential	4	12	4	1674	1.1845	0.4284
Sequential	4	12	5	1285	0.9927	0.2743
Sequential	4	15	1	1315	1.0453	0.2707
Sequential	4	15	2	1233	0.9355	0.2926
Sequential	4	15	3	1690	1.3127	0.4074
Sequential	4	15	4	1242	1.2939	0.5092
Sequential	4	15	5	1230	0.9392	0.2785
Sequential	4	18	1	1784	0.9016	0.2232
Sequential	4	18	2	1708	0.9018	0.2283
Sequential	4	18	3	1280	1.0182	0.2788
Sequential	4	18	4	1300	0.924	0.2627
Sequential	4	18	5	1649	1.0733	0.2999
Sequential	8	3	1	1582	0.9797	0.2574
Sequential	8	3	2	1560	1.101	0.2976
Sequential	8	3	3	1552	1.034	0.284
Sequential	8	3	4	1533	0.9888	0.2895
Sequential	8	3	5	1607	0.7832	0.2438

Sequential	8	6	1	1828	1.114	0.2961
Sequential	8	6	2	1413	1.1313	0.4541
Sequential	8	6	3	1398	1.0407	0.3094
Sequential	8	6	4	1403	0.9097	0.2552
Sequential	8	6	5	1361	1.0282	0.4154
Sequential	8	9	1	1768	0.8161	0.1843
Sequential	8	9	2	1304	1.0212	0.3022
Sequential	8	9	3	1382	0.9709	0.2729
Sequential	8	9	4	1703	0.9278	0.2615
Sequential	8	9	5	1332	0.9142	0.2634
Sequential	8	12	1	1755	0.8836	0.2267
Sequential	8	12	2	1347	0.9536	0.2734
Sequential	8	12	3	1343	1.0295	0.2811
Sequential	8	12	4	1799	0.8876	0.23
Sequential	8	12	5	1306	1.0665	0.3181
Sequential	8	15	1	1746	0.9886	0.2277
Sequential	8	15	2	1381	0.9259	0.3366
Sequential	8	15	3	1738	0.9048	0.2269
Sequential	8	15	4	1329	1.0421	0.3039
Sequential	8	15	5	1333	0.9683	0.2981
Sequential	8	18	1	1705	0.89	0.2252
Sequential	8	18	2	1334	0.8877	0.2661
Sequential	8	18	3	1717	0.9678	0.2198
Sequential	8	18	4	1749	1.0419	0.2154
Sequential	8	18	5	1762	0.9016	0.2255
Sequential	16	3	1	2040	0.9809	0.3813
Sequential	16	3	2	1797	1.0053	0.2816
Sequential	16	3	3	1695	1.0251	0.2605
Sequential	16	3	4	1761	1.0871	0.3051
Sequential	16	3	5	1844	0.908	0.2492
Sequential	16	6	1	1514	0.9211	0.2634
Sequential	16	6	2	1618	1.1031	0.2929
Sequential	16	6	3	1561	0.9623	0.3022
Sequential	16	6	4	1522	1.2777	0.468
Sequential	16	6	5	1662	0.8902	0.2239
Sequential	16	9	1	1496	0.9521	0.2811
Sequential	16	9	2	1499	1.0377	0.3077
Sequential	16	9	3	1428	1.0155	0.3553
Sequential	16	9	4	1450	1.0058	0.2983

Sequential	16	9	5	1505	1.4256	0.3571
Sequential	16	12	1	1511	0.9021	0.2285
Sequential	16	12	2	1437	1.0335	0.2674
Sequential	16	12	3	1455	1.2088	0.4251
Sequential	16	12	4	1488	0.9218	0.2608
Sequential	16	12	5	1468	0.9264	0.2651
Sequential	16	15	1	1926	0.7276	0.1628
Sequential	16	15	2	1845	0.9629	0.2402
Sequential	16	15	3	1927	0.874	0.2251
Sequential	16	15	4	1870	0.9381	0.2299
Sequential	16	15	5	1956	0.9188	0.2286
Sequential	16	18	1	1373	1.4102	0.3661
Sequential	16	18	2	1846	0.8993	0.2246
Sequential	16	18	3	1835	0.9485	0.2275
Sequential	16	18	4	1381	0.9379	0.2824
Sequential	16	18	5	1798	0.984	0.2188
Sequential	32	3	1	2202	1.0034	0.287
Sequential	32	3	2	2324	1.0171	0.2707
Sequential	32	3	3	2276	1.0518	0.3089
Sequential	32	3	4	2340	1.0859	0.4622
Sequential	32	3	5	2258	0.9852	0.2671
Sequential	32	6	1	1822	0.9305	0.2607
Sequential	32	6	2	1868	0.9667	0.249
Sequential	32	6	3	1779	0.9971	0.2807
Sequential	32	6	4	1805	0.9495	0.3192
Sequential	32	6	5	1698	0.9207	0.2563
Sequential	32	9	1	1555	1.0718	0.3097
Sequential	32	9	2	1689	0.93	0.2611
Sequential	32	9	3	1663	1.0359	0.3208
Sequential	32	9	4	1680	0.9398	0.267
Sequential	32	9	5	2073	0.86	0.2294
Sequential	32	12	1	1621	0.9242	0.2604
Sequential	32	12	2	1677	0.9721	0.2527
Sequential	32	12	3	1570	1.1364	0.4562
Sequential	32	12	4	2013	0.8621	0.2313
Sequential	32	12	5	1605	0.9742	0.2648
Sequential	32	15	1	2006	0.8709	0.2335
Sequential	32	15	2	1579	0.9164	0.263
Sequential	32	15	3	1546	0.9286	0.2623

Sequential	32	15	4	1616	0.925	0.26
Sequential	32	15	5	1625	0.9655	0.2751
Sequential	32	18	1	1625	1.0196	0.2641
Sequential	32	18	2	2015	0.9156	0.2532
Sequential	32	18	3	2003	1.3103	0.3036
Sequential	32	18	4	1641	1.0615	0.2676
Sequential	32	18	5	1969	0.9609	0.2465
Sequential	64	3	1	3203	1.4675	0.658
Sequential	64	3	2	3251	1.1758	0.4774
Sequential	64	3	3	3559	1.318	0.6194
Sequential	64	3	4	3316	0.9539	0.2527
Sequential	64	3	5	3374	1.071	0.3281
Sequential	64	6	1	2715	0.9615	0.3837
Sequential	64	6	2	2338	0.9234	0.2568
Sequential	64	6	3	2394	0.9756	0.2807
Sequential	64	6	4	2685	1.1033	0.6291
Sequential	64	6	5	2278	1.2387	0.5607
Sequential	64	9	1	1950	1.173	0.3706
Sequential	64	9	2	1986	1.015	0.2918
Sequential	64	9	3	2455	0.9156	0.2203
Sequential	64	9	4	1946	1.2186	0.3604
Sequential	64	9	5	2514	0.9752	0.232
Sequential	64	12	1	2037	1.749	0.4596
Sequential	64	12	2	1946	0.9739	0.2606
Sequential	64	12	3	1973	1.0234	0.3093
Sequential	64	12	4	2018	0.9633	0.2557
Sequential	64	12	5	1933	0.9603	0.2867
Sequential	64	15	1	2157	1.055	0.2853
Sequential	64	15	2	2017	1.4069	0.6212
Sequential	64	15	3	2353	0.9248	0.2249
Sequential	64	15	4	1853	0.972	0.2951
Sequential	64	15	5	2244	0.8649	0.2168
Sequential	64	18	1	1851	1.0251	0.3165
Sequential	64	18	2	1862	1.0216	0.3159
Sequential	64	18	3	1942	0.9818	0.2645
Sequential	64	18	4	2291	0.9138	0.2127
Sequential	64	18	5	1918	1.986	0.456
Sequential	128	3	1	5779	1.0039	0.4062
Sequential	128	3	2	5617	0.8925	0.36

Sequential	128	3	3	5579	0.8315	0.3168
Sequential	128	3	4	5749	0.9594	0.4543
Sequential	128	3	5	5756	1.0393	0.4545
Sequential	128	6	1	3961	1.2152	0.5202
Sequential	128	6	2	3369	0.9715	0.2669
Sequential	128	6	3	3633	1.0821	0.2813
Sequential	128	6	4	3483	0.9946	0.28
Sequential	128	6	5	3551	1.1047	0.4324
Sequential	128	9	1	2877	0.9999	0.2993
Sequential	128	9	2	2865	1.2533	0.4206
Sequential	128	9	3	2801	1.2279	0.552
Sequential	128	9	4	2853	1.3153	0.5244
Sequential	128	9	5	2896	1.0691	0.4775
Sequential	128	12	1	2858	0.9184	0.2206
Sequential	128	12	2	2596	1.0851	0.2924
Sequential	128	12	3	2736	1.2419	0.4891
Sequential	128	12	4	2690	1.2591	0.4795
Sequential	128	12	5	3190	0.9611	0.2428
Sequential	128	15	1	3066	0.7594	0.2301
Sequential	128	15	2	2636	1.2729	0.3386
Sequential	128	15	3	2605	1.0768	0.2698
Sequential	128	15	4	2969	0.7946	0.2838
Sequential	128	15	5	2606	0.9309	0.2494
Sequential	128	18	1	2925	0.8514	0.2401
Sequential	128	18	2	2417	0.9958	0.2826
Sequential	128	18	3	2513	1.1157	0.3114
Sequential	128	18	4	2954	0.8326	0.2215
Sequential	128	18	5	2923	0.8773	0.236
Static	1	3	1	176	4.277	2.2522
Static	1	3	2	178	3.8016	1.908
Static	1	3	3	166	7.26	3.6346
Static	1	3	4	162	5.7373	3.1688
Static	1	3	5	178	4.5526	2.7025
Static	1	6	1	119	7.3357	4.4162
Static	1	6	2	139	7.2797	4.2785
Static	1	6	3	141	7.5949	4.1791
Static	1	6	4	141	8.2275	4.8849
Static	1	6	5	141	6.8821	3.7604
Static	1	9	1	122	10.1796	5.4713

Static	1	9	2	124	7.0203	3.9628
Static	1	9	3	105	8.8234	5.8375
Static	1	9	4	104	7.3499	4.6024
Static	1	9	5	105	8.8579	5.5441
Static	1	12	1	105	9.5006	6.0909
Static	1	12	2	106	8.8646	5.4528
Static	1	12	3	99	9.7406	6.2896
Static	1	12	4	100	10.8186	7.4078
Static	1	12	5	102	9.8543	6.4217
Static	1	15	1	110	10.9109	7.0031
Static	1	15	2	101	10.0585	6.671
Static	1	15	3	109	10.4318	6.5653
Static	1	15	4	105	10.2897	6.3068
Static	1	15	5	105	9.8412	6.2439
Static	1	18	1	107	11.7506	8.1022
Static	1	18	2	107	11.6673	7.78
Static	1	18	3	106	11.1489	7.5273
Static	1	18	4	107	11.3898	7.6119
Static	1	18	5	110	11.0326	6.7164
Static	2	3	1	217	4.1859	2.295
Static	2	3	2	205	8.6138	4.7556
Static	2	3	3	207	4.1903	2.1441
Static	2	3	4	204	6.7372	3.0479
Static	2	3	5	205	4.6036	2.6795
Static	2	6	1	145	6.7019	3.999
Static	2	6	2	145	6.9718	4.181
Static	2	6	3	145	7.4541	4.6292
Static	2	6	4	139	6.7176	3.6928
Static	2	6	5	158	7.1688	4.1182
Static	2	9	1	123	8.8749	5.267
Static	2	9	2	117	7.8834	4.6705
Static	2	9	3	122	8.6534	5.0515
Static	2	9	4	121	8.2086	5.1495
Static	2	9	5	123	9.6633	5.6071
Static	2	12	1	112	9.2284	5.6087
Static	2	12	2	113	9.9938	6.3938
Static	2	12	3	112	9.2989	5.8487
Static	2	12	4	113	9.2204	5.7385
Static	2	12	5	114	9.6131	5.5465

Static	2	15	1	115	9.8011	6.0647
Static	2	15	2	116	10.2068	6.7071
Static	2	15	3	114	9.0854	5.5395
Static	2	15	4	112	8.7677	5.3739
Static	2	15	5	112	9.6401	5.7888
Static	2	18	1	122	9.692	5.886
Static	2	18	2	124	10.3984	6.4449
Static	2	18	3	120	10.2474	6.3423
Static	2	18	4	115	10.0705	6.1565
Static	2	18	5	121	10.298	6.4279
Static	4	3	1	280	4.7559	2.3157
Static	4	3	2	299	3.4238	1.6561
Static	4	3	3	285	3.5402	1.8116
Static	4	3	4	277	6.9598	3.956
Static	4	3	5	282	3.6271	1.7183
Static	4	6	1	183	6.6045	3.8651
Static	4	6	2	182	6.4624	3.2478
Static	4	6	3	184	6.0846	3.3867
Static	4	6	4	186	5.5796	3.0165
Static	4	6	5	183	7.0902	4.2632
Static	4	9	1	149	7.9735	4.7305
Static	4	9	2	160	7.7341	4.2878
Static	4	9	3	148	8.5509	4.825
Static	4	9	4	149	7.8268	4.5022
Static	4	9	5	149	8.801	4.9618
Static	4	12	1	138	9.4463	5.2563
Static	4	12	2	137	9.6229	6.2116
Static	4	12	3	137	8.0526	4.6526
Static	4	12	4	138	8.1768	5.0249
Static	4	12	5	169	8.2576	4.3609
Static	4	15	1	167	11.3059	6.1202
Static	4	15	2	165	8.7384	5.1027
Static	4	15	3	167	11.3054	6.5305
Static	4	15	4	175	10.1117	5.8204
Static	4	15	5	156	9.4279	5.3709
Static	4	18	1	147	9.8065	5.5307
Static	4	18	2	149	10.1493	5.7017
Static	4	18	3	137	10.5898	6.4733
Static	4	18	4	145	9.8266	5.4684

Static	4	18	5	149	10.492	6.0131
Static	8	3	1	430	3.3253	1.4335
Static	8	3	2	423	3.3019	1.5931
Static	8	3	3	460	3.8133	1.9018
Static	8	3	4	431	3.4794	1.6202
Static	8	3	5	429	3.5101	1.671
Static	8	6	1	257	6.3629	3.5117
Static	8	6	2	258	6.2436	3.4013
Static	8	6	3	269	6.2888	3.4064
Static	8	6	4	247	6.0042	2.9214
Static	8	6	5	262	6.8539	3.5027
Static	8	9	1	203	8.1311	4.5972
Static	8	9	2	262	7.6486	4.3576
Static	8	9	3	259	7.6394	4.0148
Static	8	9	4	272	8.2463	4.3316
Static	8	9	5	276	9.0021	4.6318
Static	8	12	1	190	9.8185	5.872
Static	8	12	2	188	8.7259	4.6221
Static	8	12	3	190	10.0539	5.5571
Static	8	12	4	198	9.3873	4.9491
Static	8	12	5	189	9.4017	5.5411
Static	8	15	1	185	10.647	5.9641
Static	8	15	2	190	9.7736	5.4864
Static	8	15	3	182	10.1546	5.8067
Static	8	15	4	191	11.6276	6.1673
Static	8	15	5	185	11.6463	6.7053
Static	8	18	1	184	11.0377	5.6421
Static	8	18	2	193	9.3957	5.1455
Static	8	18	3	187	9.2678	4.7837
Static	8	18	4	189	10.4224	5.7656
Static	8	18	5	188	9.1355	5.0267
Static	16	3	1	695	7.4621	3.7781
Static	16	3	2	695	7.8658	3.3183
Static	16	3	3	715	5.7984	2.8305
Static	16	3	4	693	2.9365	1.2506
Static	16	3	5	806	3.2014	1.3338
Static	16	6	1	398	5.7493	2.9226
Static	16	6	2	402	6.7006	3.2858
Static	16	6	3	393	4.8869	2.2023

Static	16	6	4	391	5.9346	3.3037
Static	16	6	5	396	6.2897	3.103
Static	16	9	1	301	6.9923	3.733
Static	16	9	2	309	7.3458	3.7865
Static	16	9	3	301	6.6409	3.3432
Static	16	9	4	312	7.797	3.8217
Static	16	9	5	298	7.7138	4.138
Static	16	12	1	278	7.8898	4.0085
Static	16	12	2	280	8.443	4.6226
Static	16	12	3	280	8.3394	4.2426
Static	16	12	4	293	8.756	4.5225
Static	16	12	5	279	9.1101	4.8271
Static	16	15	1	277	10.5746	5.3703
Static	16	15	2	269	11.2022	5.6832
Static	16	15	3	266	9.5645	4.9201
Static	16	15	4	269	10.2846	5.8248
Static	16	15	5	270	11.1802	6.1268
Static	16	18	1	270	7.8787	3.5307
Static	16	18	2	271	11.6107	6.2913
Static	16	18	3	270	11.0519	6.0468
Static	16	18	4	269	12.2556	6.9014
Static	16	18	5	272	10.5428	5.63
Static	32	3	1	1296	2.7631	1.1538
Static	32	3	2	1289	2.4794	0.9908
Static	32	3	3	1239	2.8153	1.1396
Static	32	3	4	1259	2.7923	1.1393
Static	32	3	5	1262	3.1289	1.3621
Static	32	6	1	679	4.9155	2.2907
Static	32	6	2	701	8.9236	3.5004
Static	32	6	3	690	5.0561	2.4803
Static	32	6	4	717	4.8358	2.2402
Static	32	6	5	982	5.1892	2.2829
Static	32	9	1	537	7.4663	3.6506
Static	32	9	2	516	6.7806	2.931
Static	32	9	3	523	6.9916	3.2933
Static	32	9	4	521	6.4617	2.8184
Static	32	9	5	513	7.2101	3.9538
Static	32	12	1	470	8.7465	4.3998
Static	32	12	2	678	8.6834	4.2325

Static	32	12	3	684	8.8157	3.9572
Static	32	12	4	492	7.8727	3.5789
Static	32	12	5	472	8.3066	4.2268
Static	32	15	1	467	10.5208	4.8502
Static	32	15	2	451	10.103	4.8803
Static	32	15	3	456	8.8159	4.2956
Static	32	15	4	452	10.7994	5.0182
Static	32	15	5	450	9.7491	4.5693
Static	32	18	1	436	11.2003	5.8205
Static	32	18	2	442	11.5302	5.9987
Static	32	18	3	448	8.8834	4.3978
Static	32	18	4	432	9.6567	3.9026
Static	32	18	5	436	8.2011	3.8422
Static	64	3	1	2310	4.2285	2.3677
Static	64	3	2	2408	2.2024	0.8495
Static	64	3	3	2368	2.2259	0.676
Static	64	3	4	2494	2.2816	0.7894
Static	64	3	5	2585	2.3531	0.7217
Static	64	6	1	1294	4.2226	1.8299
Static	64	6	2	1291	4.1206	1.4673
Static	64	6	3	1302	4.0352	1.518
Static	64	6	4	1305	4.4899	1.9334
Static	64	6	5	1301	4.3641	1.7917
Static	64	9	1	997	6.7233	2.7304
Static	64	9	2	947	5.8836	2.6274
Static	64	9	3	1038	5.7935	2.2244
Static	64	9	4	960	6.2288	2.6452
Static	64	9	5	961	6.4176	2.7217
Static	64	12	1	848	8.3535	3.3934
Static	64	12	2	926	8.2148	2.9684
Static	64	12	3	847	7.9106	3.3688
Static	64	12	4	872	8.3259	3.7157
Static	64	12	5	815	7.9831	3.355
Static	64	15	1	807	8.6271	3.6375
Static	64	15	2	787	9.4069	4.4332
Static	64	15	3	790	9.7021	3.8907
Static	64	15	4	786	9.408	4.5302
Static	64	15	5	777	9.7065	4.7939
Static	64	18	1	761	9.0951	4.0811

Static	64	18	2	748	10.5217	4.6078
Static	64	18	3	768	11.6307	4.5005
Static	64	18	4	755	9.8579	4.6548
Static	64	18	5	741	10.7863	4.5788
Static	128	3	1	4456	1.9903	0.606
Static	128	3	2	4535	2.0365	0.6038
Static	128	3	3	4529	1.921	0.569
Static	128	3	4	4492	2.0162	0.5676
Static	128	3	5	4661	2.0145	0.6009
Static	128	6	1	2601	3.8955	1.8863
Static	128	6	2	2365	3.7366	1.2548
Static	128	6	3	2488	3.7365	1.3249
Static	128	6	4	2390	4.4707	2.2855
Static	128	6	5	2457	3.6093	1.252
Static	128	9	1	1784	5.3289	1.9871
Static	128	9	2	1712	5.3748	1.9642
Static	128	9	3	1825	5.3827	2.0859
Static	128	9	4	1773	5.2409	1.8934
Static	128	9	5	1687	5.6552	2.1031
Static	128	12	1	1596	6.9074	2.6507
Static	128	12	2	1551	6.762	2.6304
Static	128	12	3	1666	6.9093	2.4824
Static	128	12	4	2201	6.7847	2.8127
Static	128	12	5	1528	6.7639	2.3831
Static	128	15	1	1426	8.5697	3.4328
Static	128	15	2	1491	8.324	2.8775
Static	128	15	3	1456	8.9322	3.1099
Static	128	15	4	1409	8.3806	2.7965
Static	128	15	5	1450	7.9076	3.2804
Static	128	18	1	1556	9.1739	3.6686
Static	128	18	2	1395	9.311	3.3709
Static	128	18	3	1426	9.0616	3.5086
Static	128	18	4	1525	9.785	4.0257
Static	128	18	5	1474	8.7316	3.2454
Dynamic	1	3	1	240	3.0357	1.6693
Dynamic	1	3	2	212	5.6878	3.3379
Dynamic	1	3	3	228	5.0442	2.9838
Dynamic	1	3	4	214	3.9521	2.3365
Dynamic	1	3	5	236	3.1991	1.8123

Dynamic	1	6	1	145	6.899	4.7548
Dynamic	1	6	2	185	5.6546	3.5581
Dynamic	1	6	3	146	6.5646	4.4221
Dynamic	1	6	4	173	4.1573	2.4653
Dynamic	1	6	5	196	3.2582	2.0972
Dynamic	1	9	1	180	3.6073	2.1657
Dynamic	1	9	2	159	5.3386	3.363
Dynamic	1	9	3	186	3.3943	2.0813
Dynamic	1	9	4	196	5.9958	3.7789
Dynamic	1	9	5	176	5.1034	3.4945
Dynamic	1	12	1	124	7.3121	5.5985
Dynamic	1	12	2	117	7.1267	5.3421
Dynamic	1	12	3	116	8.1988	6.0996
Dynamic	1	12	4	140	6.5916	4.3902
Dynamic	1	12	5	200	3.8249	2.3766
Dynamic	1	15	1	123	6.6863	4.7753
Dynamic	1	15	2	178	4.8602	3.2272
Dynamic	1	15	3	123	7.1968	5.1905
Dynamic	1	15	4	146	5.5586	3.8643
Dynamic	1	15	5	140	4.704	3.4878
Dynamic	1	18	1	134	7.8564	5.8796
Dynamic	1	18	2	132	8.2489	5.9809
Dynamic	1	18	3	125	8.0501	5.8829
Dynamic	1	18	4	165	4.3901	2.9284
Dynamic	1	18	5	142	7.1916	5.1544
Dynamic	2	3	1	245	4.0906	2.3383
Dynamic	2	3	2	278	4.0466	2.5525
Dynamic	2	3	3	263	3.8611	2.2548
Dynamic	2	3	4	269	5.0552	2.701
Dynamic	2	3	5	254	6.8656	4.0783
Dynamic	2	6	1	220	3.8229	2.3029
Dynamic	2	6	2	240	3.3234	1.7797
Dynamic	2	6	3	216	3.7196	2.4253
Dynamic	2	6	4	236	4.0302	2.4148
Dynamic	2	6	5	202	6.4818	4.2341
Dynamic	2	9	1	187	8.7021	5.1886
Dynamic	2	9	2	218	3.6543	2.648
Dynamic	2	9	3	219	3.0856	1.9384
Dynamic	2	9	4	192	4.0967	2.7156

Dynamic	2	9	5	182	4.8014	3.192
Dynamic	2	12	1	147	5.7167	3.9949
Dynamic	2	12	2	217	3.5055	2.088
Dynamic	2	12	3	190	4.3192	2.9212
Dynamic	2	12	4	141	6.5541	4.8809
Dynamic	2	12	5	145	7.4768	5.6406
Dynamic	2	15	1	162	6.9888	4.9795
Dynamic	2	15	2	131	7.7566	5.9133
Dynamic	2	15	3	141	6.5402	4.6183
Dynamic	2	15	4	140	5.4643	3.84
Dynamic	2	15	5	154	5.0757	3.3959
Dynamic	2	18	1	131	7.7609	5.7209
Dynamic	2	18	2	138	8.787	6.7274
Dynamic	2	18	3	166	6.2561	4.3573
Dynamic	2	18	4	158	6.6089	4.6829
Dynamic	2	18	5	162	6.5299	4.9828
Dynamic	4	3	1	353	3.1648	1.7492
Dynamic	4	3	2	353	3.461	1.8287
Dynamic	4	3	3	346	7.6009	4.0658
Dynamic	4	3	4	397	2.8386	1.5211
Dynamic	4	3	5	366	4.0947	2.5139
Dynamic	4	6	1	223	6.5006	3.9708
Dynamic	4	6	2	290	4.9301	2.7486
Dynamic	4	6	3	234	4.5627	2.639
Dynamic	4	6	4	307	3.3665	1.791
Dynamic	4	6	5	297	3.6597	1.9645
Dynamic	4	9	1	235	3.8032	2.3313
Dynamic	4	9	2	218	4.4488	2.7828
Dynamic	4	9	3	234	3.8031	2.5283
Dynamic	4	9	4	207	8.3933	5.5131
Dynamic	4	9	5	221	4.5517	2.741
Dynamic	4	12	1	168	7.2517	5.2125
Dynamic	4	12	2	169	6.0371	4.1327
Dynamic	4	12	3	206	4.8142	3.2701
Dynamic	4	12	4	197	5.6157	3.9471
Dynamic	4	12	5	198	4.2389	2.4252
Dynamic	4	15	1	161	7.2662	5.0893
Dynamic	4	15	2	214	4.2052	2.7379
Dynamic	4	15	3	158	6.4625	4.6703

Dynamic	4	15	4	204	3.0548	1.9246
Dynamic	4	15	5	203	3.9913	2.7617
Dynamic	4	18	1	190	5.6361	3.9203
Dynamic	4	18	2	159	8.1929	5.6241
Dynamic	4	18	3	174	6.6709	4.8577
Dynamic	4	18	4	168	5.6486	3.8329
Dynamic	4	18	5	160	8.23	5.6198
Dynamic	8	3	1	543	3.3194	1.5955
Dynamic	8	3	2	650	2.8556	1.355
Dynamic	8	3	3	651	3.2479	1.4106
Dynamic	8	3	4	654	3.539	1.8392
Dynamic	8	3	5	587	3.6039	1.5619
Dynamic	8	6	1	329	6.0095	3.338
Dynamic	8	6	2	300	4.8254	2.6228
Dynamic	8	6	3	325	3.922	2.1412
Dynamic	8	6	4	305	5.5439	3.19
Dynamic	8	6	5	349	4.0689	2.1097
Dynamic	8	9	1	270	4.6685	2.8098
Dynamic	8	9	2	265	5.7623	3.5185
Dynamic	8	9	3	272	5.5823	3.1356
Dynamic	8	9	4	290	7.1664	4.3461
Dynamic	8	9	5	259	5.0481	3.1073
Dynamic	8	12	1	262	6.0432	3.662
Dynamic	8	12	2	256	4.4271	2.6151
Dynamic	8	12	3	260	4.8277	2.7761
Dynamic	8	12	4	333	5.0865	3.7378
Dynamic	8	12	5	281	5.5592	3.6246
Dynamic	8	15	1	212	7.0426	4.7266
Dynamic	8	15	2	207	5.4313	3.5507
Dynamic	8	15	3	218	5.1944	3.3979
Dynamic	8	15	4	298	4.5792	2.8771
Dynamic	8	15	5	322	4.1204	2.3509
Dynamic	8	18	1	339	3.6715	2.0889
Dynamic	8	18	2	268	4.3129	3.0193
Dynamic	8	18	3	202	6.2562	4.0151
Dynamic	8	18	4	294	3.6875	2.3454
Dynamic	8	18	5	199	7.7452	5.3775
Dynamic	16	3	1	716	5.3526	2.5045
Dynamic	16	3	2	713	3.7008	1.7237

Dynamic	16	3	3	762	3.3718	1.5817
Dynamic	16	3	4	767	7.9657	3.9991
Dynamic	16	3	5	698	4.3661	2.2198
Dynamic	16	6	1	480	3.7289	1.8509
Dynamic	16	6	2	492	3.6333	2.0474
Dynamic	16	6	3	471	4.359	2.0679
Dynamic	16	6	4	497	4.2715	1.9901
Dynamic	16	6	5	482	2.4973	1.0905
Dynamic	16	9	1	378	5.1791	2.7059
Dynamic	16	9	2	384	4.1032	2.2125
Dynamic	16	9	3	412	3.0125	1.4937
Dynamic	16	9	4	411	5.2032	2.915
Dynamic	16	9	5	470	3.0376	1.6194
Dynamic	16	12	1	357	3.697	1.8876
Dynamic	16	12	2	330	4.8192	2.7307
Dynamic	16	12	3	338	4.7137	2.5201
Dynamic	16	12	4	318	6.7475	3.7536
Dynamic	16	12	5	371	3.8989	2.2069
Dynamic	16	15	1	307	5.2742	3.2521
Dynamic	16	15	2	294	4.1432	2.4718
Dynamic	16	15	3	285	5.2103	3.1315
Dynamic	16	15	4	375	3.4584	2.0399
Dynamic	16	15	5	312	4.1722	2.537
Dynamic	16	18	1	316	6.5399	4.1752
Dynamic	16	18	2	303	5.6251	3.6299
Dynamic	16	18	3	314	5.2818	3.0774
Dynamic	16	18	4	344	3.716	1.9829
Dynamic	16	18	5	282	6.6714	3.9839
Dynamic	32	3	1	1317	3.7743	1.7392
Dynamic	32	3	2	1217	2.6378	1.2637
Dynamic	32	3	3	1668	2.5961	1.0389
Dynamic	32	3	4	1580	2.8668	1.0756
Dynamic	32	3	5	1305	2.5423	0.9634
Dynamic	32	6	1	695	4.2583	2.1384
Dynamic	32	6	2	673	3.7155	1.9117
Dynamic	32	6	3	720	4.4338	2.2206
Dynamic	32	6	4	747	3.154	1.4399
Dynamic	32	6	5	758	3.148	1.4753
Dynamic	32	9	1	551	4.171	2.0818

Dynamic	32	9	2	535	3.6446	2.0685
Dynamic	32	9	3	554	3.2735	1.6075
Dynamic	32	9	4	715	3.3176	1.5115
Dynamic	32	9	5	1330	3.1817	1.2362
Dynamic	32	12	1	516	5.4717	3.0493
Dynamic	32	12	2	595	2.9006	1.4265
Dynamic	32	12	3	587	3.2492	1.8979
Dynamic	32	12	4	599	3.4515	1.8803
Dynamic	32	12	5	698	3.1669	1.5065
Dynamic	32	15	1	703	4.4271	2.2447
Dynamic	32	15	2	466	3.9672	2.2635
Dynamic	32	15	3	456	6.2828	3.0236
Dynamic	32	15	4	466	4.1221	2.2356
Dynamic	32	15	5	495	4.3589	2.2123
Dynamic	32	18	1	461	4.5096	2.6849
Dynamic	32	18	2	470	4.6003	2.3945
Dynamic	32	18	3	646	3.9727	2.1761
Dynamic	32	18	4	698	3.7924	1.9711
Dynamic	32	18	5	458	5.6844	3.5126
Dynamic	64	3	1	2193	2.6271	1.2388
Dynamic	64	3	2	2281	2.5051	0.9255
Dynamic	64	3	3	2137	2.521	0.8819
Dynamic	64	3	4	2277	2.6624	1.1105
Dynamic	64	3	5	2309	4.8693	2.961
Dynamic	64	6	1	1308	2.917	1.2432
Dynamic	64	6	2	1287	4.7068	2.3597
Dynamic	64	6	3	1359	2.7858	1.158
Dynamic	64	6	4	1339	2.9118	1.1467
Dynamic	64	6	5	1226	5.7061	2.447
Dynamic	64	9	1	971	3.6544	1.6967
Dynamic	64	9	2	874	3.6467	1.5703
Dynamic	64	9	3	933	3.8798	1.6338
Dynamic	64	9	4	974	3.0806	1.41
Dynamic	64	9	5	1412	2.8549	1.1485
Dynamic	64	12	1	864	3.889	1.7374
Dynamic	64	12	2	817	3.0241	1.3304
Dynamic	64	12	3	828	3.076	1.4101
Dynamic	64	12	4	927	3.1562	1.9237
Dynamic	64	12	5	911	3.5089	1.4835

Dynamic	64	15	1	764	3.7229	1.8406
Dynamic	64	15	2	765	3.9189	1.9534
Dynamic	64	15	3	890	3.1664	1.5445
Dynamic	64	15	4	866	3.683	1.6753
Dynamic	64	15	5	782	3.1885	1.4075
Dynamic	64	18	1	820	3.5145	1.6504
Dynamic	64	18	2	755	4.2954	2.1722
Dynamic	64	18	3	1139	3.1404	1.3262
Dynamic	64	18	4	852	3.4751	1.4729
Dynamic	64	18	5	823	4.0588	2.0601
Dynamic	128	3	1	4518	2.4054	1.0055
Dynamic	128	3	2	5273	2.2125	0.8482
Dynamic	128	3	3	4414	3.0125	1.3061
Dynamic	128	3	4	4522	2.5708	0.946
Dynamic	128	3	5	4688	2.2662	0.7686
Dynamic	128	6	1	2548	2.4461	0.8968
Dynamic	128	6	2	2476	3.7591	1.5915
Dynamic	128	6	3	2368	3.2665	1.2241
Dynamic	128	6	4	2413	3.0497	1.261
Dynamic	128	6	5	2448	3.3333	1.3128
Dynamic	128	9	1	1695	3.8697	1.5437
Dynamic	128	9	2	1756	3.5007	1.2812
Dynamic	128	9	3	1705	3.6135	1.3317
Dynamic	128	9	4	1903	2.8351	1.0668
Dynamic	128	9	5	1667	3.7691	1.3978
Dynamic	128	12	1	1554	3.4445	1.2444
Dynamic	128	12	2	1584	3.0211	1.1198
Dynamic	128	12	3	1686	2.7101	0.9627
Dynamic	128	12	4	1735	2.825	1.1093
Dynamic	128	12	5	1493	3.6365	1.3311
Dynamic	128	15	1	1543	2.5713	0.9092
Dynamic	128	15	2	1658	3.7464	1.3969
Dynamic	128	15	3	1522	2.7512	0.9978
Dynamic	128	15	4	1523	3.274	1.3418
Dynamic	128	15	5	1461	3.0235	1.1913
Dynamic	128	18	1	1398	3.033	1.2104
Dynamic	128	18	2	1621	2.8473	1.056
Dynamic	128	18	3	1542	3.0466	1.3075
Dynamic	128	18	4	1572	2.8697	0.9749

Dynamic	128	18	5	1419	3.6545	1.3801

Table A.1: This took a while to run.

A.1.2 Experiment 2

Evaluation is the trial number at which the returned solution was found.

I chose to leave out runtimes, since they were only good for measuring HRM update times, when applicable. The total time taken to do HRM updates was $\mu = 10.45$ seconds ($\sigma = 1.08, n = 50$) across all applicable runs.

Run	Optimizer	Strategy	Best	Violation	Evaluation
1	LHS	None	34301.8547	0.0	140
1	MADS	Historical	80805.2973	0.0	925
1	MADS	Penalty	132253.2771	9.7082	936
1	SRBF	Historical	44966.4879	0.0	874
1	SRBF	Penalty	7005.6023	7.3993	935
2	LHS	None	34563.6731	0.0	87
2	MADS	Historical	57958.4793	0.0	924
2	MADS	Penalty	105898.8038	4.392	945
2	SRBF	Historical	46446.167	0.0	939
2	SRBF	Penalty	6547.7654	8.4304	803
3	LHS	None	46560.3052	0.0	158
3	MADS	Historical	75842.7342	0.0	786
3	MADS	Penalty	112668.0277	5.6555	952
3	SRBF	Historical	64663.1835	0.0	873
3	SRBF	Penalty	73757.1302	1.8947	988
4	LHS	None	12853.1212	0.0	83
4	MADS	Historical	68919.2857	0.0	570
4	MADS	Penalty	69847.1394	10.1338	988
4	SRBF	Historical	21170.9085	0.0	873
4	SRBF	Penalty	106770.2268	6.1178	984
5	LHS	None	32949.0307	0.0	133
5	MADS	Historical	80795.7111	0.0	928
5	MADS	Penalty	117729.6765	6.6049	953
5	SRBF	Historical	36062.8096	0.0	879
5	SRBF	Penalty	96437.1755	5.5442	993
6	LHS	None	36385.1083	0.0	70
6	MADS	Historical	52199.8693	0.0	793
6	MADS	Penalty	136827.8404	9.0325	985

6	SRBF	Historical	47570.7726	0.0	938
6	SRBF	Penalty	11533.9982	7.3904	401
7	LHS	None	49245.9613	0.0	123
7	MADS	Historical	69577.5874	0.0	985
7	MADS	Penalty	120340.5955	6.7449	936
7	SRBF	Historical	55554.8337	0.0	872
7	SRBF	Penalty	7846.5133	6.4888	803
8	LHS	None	36947.5992	0.0	147
8	MADS	Historical	69184.8043	0.0	451
8	MADS	Penalty	125828.6315	7.3619	983
8	SRBF	Historical	66019.6245	0.0	985
8	SRBF	Penalty	111529.1367	6.0758	993
9	LHS	None	38540.1428	0.0	109
9	MADS	Historical	68946.5551	0.0	992
9	MADS	Penalty	86499.9037	15.3869	997
9	SRBF	Historical	51200.8841	0.0	937
9	SRBF	Penalty	99413.8581	5.8198	995
10	LHS	None	29634.5761	0.0	117
10	MADS	Historical	74379.6416	0.0	722
10	MADS	Penalty	103675.6701	4.7309	891
10	SRBF	Historical	50676.7498	0.0	541
10	SRBF	Penalty	108515.5233	5.7326	981
11	LHS	None	21488.1797	0.0	81
11	MADS	Historical	68855.0237	0.0	568
11	MADS	Penalty	110887.4738	5.4338	946
11	SRBF	Historical	30526.991	0.0	940
11	SRBF	Penalty	9786.238	9.5745	468
12	LHS	None	36896.5486	0.0	80
12	MADS	Historical	52365.029	0.0	791
12	MADS	Penalty	119855.5191	7.4233	943
12	SRBF	Historical	51189.8678	0.0	944
12	SRBF	Penalty	8668.8394	6.8279	869
13	LHS	None	35873.31	0.0	129
13	MADS	Historical	70162.6388	0.0	513
13	MADS	Penalty	136680.3863	8.9862	951
13	SRBF	Historical	54051.6961	0.0	943
13	SRBF	Penalty	6945.1937	7.8286	869
14	LHS	None	32387.8566	0.0	102
14	MADS	Historical	68869.789	0.0	853

14	MADS	Penalty	97541.4108	8.6461	869
14	SRBF	Historical	38285.6849	0.0	967
14	SRBF	Penalty	91870.2868	6.2955	988
15	LHS	None	28883.9287	0.0	115
15	MADS	Historical	69194.3829	0.0	992
15	MADS	Penalty	56477.8487	3.2242	982
15	SRBF	Historical	32160.3087	0.0	942
15	SRBF	Penalty	100591.9858	5.1904	996
16	LHS	None	33913.6169	0.0	149
16	MADS	Historical	80804.8841	0.0	921
16	MADS	Penalty	136945.8335	9.0695	967
16	SRBF	Historical	61935.1659	0.0	873
16	SRBF	Penalty	7322.4935	10.6574	869
17	LHS	None	40085.0558	0.0	115
17	MADS	Historical	62260.5604	0.0	454
17	MADS	Penalty	110147.7006	4.9412	953
17	SRBF	Historical	59793.5689	0.0	942
17	SRBF	Penalty	5698.9814	7.8336	334
18	LHS	None	16601.0594	0.0	98
18	MADS	Historical	68943.3086	0.0	786
18	MADS	Penalty	139225.6286	9.1215	996
18	SRBF	Historical	31749.3688	0.0	938
18	SRBF	Penalty	7667.0751	8.2171	935
19	LHS	None	38997.2375	0.0	159
19	MADS	Historical	80955.7665	0.0	929
19	MADS	Penalty	107042.7133	4.5413	956
19	SRBF	Historical	70471.8534	0.0	937
19	SRBF	Penalty	8009.6676	8.652	401
20	LHS	None	42264.272	0.0	124
20	MADS	Historical	69551.6104	0.0	783
20	MADS	Penalty	103043.0219	4.6911	954
20	SRBF	Historical	56674.4179	0.0	807
20	SRBF	Penalty	5747.2821	6.4524	869
21	LHS	None	32186.5934	0.0	102
21	MADS	Historical	71284.2872	0.0	658
21	MADS	Penalty	73740.824	12.9169	985
21	SRBF	Historical	45058.3244	0.0	939
21	SRBF	Penalty	6262.26	10.2482	869
22	LHS	None	54295.9689	0.0	141

22	MADS	Historical	72072.2424	0.0	771
22	MADS	Penalty	135134.1865	9.3592	1000
22	SRBF	Historical	64320.0823	0.0	809
22	SRBF	Penalty	9450.5968	8.4324	869
23	LHS	None	35979.3059	0.0	113
23	MADS	Historical	68933.6504	0.0	772
23	MADS	Penalty	85900.5562	6.1981	871
23	SRBF	Historical	48998.8261	0.0	936
23	SRBF	Penalty	106951.049	5.2297	976
24	LHS	None	42804.2102	0.0	118
24	MADS	Historical	80771.2513	0.0	799
24	MADS	Penalty	66857.7876	8.6555	992
24	SRBF	Historical	48582.7416	0.0	939
24	SRBF	Penalty	6947.9086	11.9715	401
25	LHS	None	43207.8914	0.0	140
25	MADS	Historical	81091.3997	0.0	907
25	MADS	Penalty	116088.7006	12.0755	963
25	SRBF	Historical	58803.216	0.0	871
25	SRBF	Penalty	6684.3203	8.7863	869

Table A.2: No Penalty hits as hard as a History lesson.

A.1.3 Experiment 3

Evaluation is the trial number at which the returned solution was found.
The shift vector utilized for eq. (5.2) was z = [36.41, 29.32, -30.60, -35.67, 26.49, 64.54, -20.10, -35.89, -5.82, 45.26, 67.09, -10.16, 39.98, -18.15, -51.62, 15.10, -46.44, 52.15, -52.45, 13.99]
I chose to leave out runtimes, since there were only good for measuring the time taken to estimate Shapley values, when applicable. The total time to estimate Shapley values was $\mu = 19.2373$ seconds ($\sigma = 6.2861, n = 200$) across all applicable runs. Runs in which some SHAP values were overestimated are denoted by one or two asterisks (*/**) under the Strategy column, denoting that dimension $\log(Split) + 1$ or $\log(Split) + 2$ was suboptimally split instead of dimension $\log(Split)$, respectively.

Run	**Optimizer**	**Strategy**	**Split**	**Best**	**Evaluation**
1	LHS	None	1	104938.5174	37
1	MADS	Best	2	8358.1619	1994
1	MADS	Best	4	5607.293	1997

1	MADS	Best	8	8709.6542	1998
1	MADS	Best	16	41486.3927	1390
1	MADS	Average	2	8358.1619	1994
1	MADS	Average	4	9062.6739	1998
1	MADS	Average	8	24931.7641	2000
1	MADS	Average	16	50850.0475	1754
1	MADS	Worst	2	6505.6585	1998
1	MADS	Worst	4	13350.6445	1999
1	MADS	Worst	8	24864.4677	1267
1	MADS	Worst	16	50535.1256	657
1	DIRECT	Best	2	9735.7598	1929
1	DIRECT	Best	4	10749.1504	1974
1	DIRECT	Best	8	15529.879	1686
1	DIRECT	Best	16	11853.7632	1607
1	DIRECT	Average	2	10817.2496	989
1	DIRECT	Average	4	9061.4397	1996
1	DIRECT	Average	8	11797.9532	1753
1	DIRECT	Average	16	20899.982	1558
1	DIRECT	Worst	2	10123.2826	1955
1	DIRECT	Worst	4	13892.6043	502
1	DIRECT	Worst	8	13965.2474	1442
1	DIRECT	Worst	16	14768.9843	636
1	MADS	None	1	50535.1256	657
1	DIRECT	None	1	14768.9843	636
2	LHS	None	1	72146.4728	27
2	MADS	Best	2	9285.7043	1998
2	MADS	Best	4	21383.6984	1999
2	MADS	Best	8	38020.3518	777
2	MADS	Best	16	50342.772	1756
2	MADS	Average	2	5398.1747	1994
2	MADS	Average	4	4320.1206	2000
2	MADS	Average	8	33582.6285	2000
2	MADS	Average	16	29489.2217	1389
2	MADS	Worst	2	9285.7043	1998
2	MADS	Worst	4	21365.6464	1022
2	MADS	Worst	8	38212.1022	1756
2	MADS	Worst	16	53419.8123	1756
2	DIRECT	Best	2	9735.7598	1929
2	DIRECT	Best	4	10749.1504	1974

2	DIRECT	Best	8	15529.879	1686
2	DIRECT	Best	16	11853.7632	1607
2	DIRECT	Average	2	10817.2496	989
2	DIRECT	Average	4	9061.4397	1996
2	DIRECT	Average	8	11797.9532	1753
2	DIRECT	Average	16	20899.982	1558
2	DIRECT	Worst	2	10123.2826	1955
2	DIRECT	Worst	4	13892.6043	502
2	DIRECT	Worst	8	13965.2474	1442
2	DIRECT	Worst	16	14768.9843	636
2	MADS	None	1	53419.8123	1756
2	DIRECT	None	1	14768.9843	636
3	LHS	None	1	58730.4268	37
3	MADS	Best	2	3451.6678	2000
3	MADS	Best	4	10599.5119	1999
3	MADS	Best	8	20582.3992	1756
3	MADS	Best	16	31958.9542	1755
3	MADS	Average	2	1322.7695	1999
3	MADS	Average	4	7363.9228	2000
3	MADS	Average	8	9079.2299	2000
3	MADS	Average	16	23501.2033	2000
3	MADS	Worst	2	3451.6678	1021
3	MADS	Worst	4	10599.5119	1510
3	MADS	Worst	8	20582.3992	1756
3	MADS	Worst	16	31958.9542	901
3	DIRECT	Best	2	9735.7598	1929
3	DIRECT	Best	4	10749.1504	1974
3	DIRECT	Best	8	15529.879	1686
3	DIRECT	Best	16	11853.7632	1607
3	DIRECT	Average	2	10817.2496	989
3	DIRECT	Average	4	9061.4397	1996
3	DIRECT	Average	8	11797.9532	1753
3	DIRECT	Average	16	20899.982	1558
3	DIRECT	Worst	2	10123.2826	1955
3	DIRECT	Worst	4	13892.6043	502
3	DIRECT	Worst	8	13965.2474	1442
3	DIRECT	Worst	16	14768.9843	636
3	MADS	None	1	31958.9542	901
3	DIRECT	None	1	14768.9843	636

4	LHS	None	1	65800.0464	29
4	MADS	Best	2	6973.5733	1998
4	MADS	Best	4	25467.0845	1999
4	MADS	Best	8	39219.6136	1512
4	MADS	Best	16	48566.3145	1512
4	MADS	Average	2	2628.0379	1997
4	MADS	Average	4	4849.398	1995
4	MADS	Average	8	29515.1324	1997
4	MADS	Average	16	47399.8484	2000
4	MADS	Worst	2	14234.5443	1021
4	MADS	Worst	4	25467.0845	1999
4	MADS	Worst	8	39219.6136	777
4	MADS	Worst	16	48566.3145	410
4	DIRECT	Best	2	9735.7598	1929
4	DIRECT	Best	4	10749.1504	1974
4	DIRECT	Best	8	15529.879	1686
4	DIRECT	Best	16	11853.7632	1607
4	DIRECT	Average	2	10817.2496	989
4	DIRECT	Average	4	9061.4397	1996
4	DIRECT	Average	8	11797.9532	1753
4	DIRECT	Average	16	20899.982	1558
4	DIRECT	Worst	2	10123.2826	1955
4	DIRECT	Worst	4	13892.6043	502
4	DIRECT	Worst	8	13965.2474	1442
4	DIRECT	Worst	16	14768.9843	636
4	MADS	None	1	48566.3145	410
4	DIRECT	None	1	14768.9843	636
5	LHS	None	1	86890.3059	25
5	MADS	Best	2	3812.6576	1998
5	MADS	Best	4	12583.2944	2000
5	MADS	Best	8	20310.1461	1999
5	MADS	Best	16	30073.7201	1878
5	MADS	Average	2	5817.5924	1997
5	MADS	Average	4	5079.6804	1999
5	MADS	Average	8	13256.1193	1753
5	MADS	Average	16	30073.7201	1390
5	MADS	Worst	2	5817.5924	1018
5	MADS	Worst	4	12502.6967	532
5	MADS	Worst	8	20255.0181	1267

5	MADS	Worst	16	29837.1225	657
5	DIRECT	Best	2	9735.7598	1929
5	DIRECT	Best	4	10749.1504	1974
5	DIRECT	Best	8	15529.879	1686
5	DIRECT	Best	16	11853.7632	1607
5	DIRECT	Average	2	10817.2496	989
5	DIRECT	Average	4	9061.4397	1996
5	DIRECT	Average	8	11797.9532	1753
5	DIRECT	Average	16	20899.982	1558
5	DIRECT	Worst	2	10123.2826	1955
5	DIRECT	Worst	4	13892.6043	502
5	DIRECT	Worst	8	13965.2474	1442
5	DIRECT	Worst	16	14768.9843	636
5	MADS	None	1	29837.1225	657
5	DIRECT	None	1	14768.9843	636
6	LHS	None	1	84983.1772	40
6	MADS	Best	2	6119.6717	1998
6	MADS	Best	4	14531.5866	2000
6	MADS	Best	8	23975.181	1756
6	MADS	Best	16	38699.2749	1634
6	MADS	Average	2	7795.1452	1020
6	MADS	Average	4	7946.4777	1022
6	MADS	Average	8	15136.9733	1999
6	MADS	Average	16	38699.2749	1878
6	MADS	Worst	2	7859.0932	1021
6	MADS	Worst	4	14393.0325	532
6	MADS	Worst	8	23920.2705	1267
6	MADS	Worst	16	38699.2749	1634
6	DIRECT	Best	2	9735.7598	1929
6	DIRECT	Best	4	10749.1504	1974
6	DIRECT	Best	8	15529.879	1686
6	DIRECT	Best	16	11853.7632	1607
6	DIRECT	Average	2	10817.2496	989
6	DIRECT	Average	4	9061.4397	1996
6	DIRECT	Average	8	11797.9532	1753
6	DIRECT	Average	16	20899.982	1558
6	DIRECT	Worst	2	10123.2826	1955
6	DIRECT	Worst	4	13892.6043	502
6	DIRECT	Worst	8	13965.2474	1442

6	DIRECT	Worst	16	14768.9843	636
6	MADS	None	1	38699.2749	1634
6	DIRECT	None	1	14768.9843	636
7	LHS	None	1	128113.1448	28
7	MADS	Best	2	12880.8902	1998
7	MADS	Best	4	31068.2096	1510
7	MADS	Best	8	53565.9208	776
7	MADS	Best*	16	69629.6459	1877
7	MADS	Average	2	4586.4388	1994
7	MADS	Average	4	14771.2898	2000
7	MADS	Average	8	53565.9208	1266
7	MADS	Average	16	47168.8829	656
7	MADS	Worst	2	10962.8374	1993
7	MADS	Worst	4	13454.1648	1019
7	MADS	Worst	8	35917.7108	287
7	MADS	Worst	16	14698.5798	656
7	DIRECT	Best	2	9735.7598	1929
7	DIRECT	Best	4	10749.1504	1974
7	DIRECT	Best	8	15529.879	1686
7	DIRECT	Best	16	14408.4017	1607
7	DIRECT	Average	2	10817.2496	989
7	DIRECT	Average	4	9061.4397	1996
7	DIRECT	Average	8	11797.9532	1753
7	DIRECT	Average	16	20899.982	1558
7	DIRECT	Worst	2	10123.2826	1955
7	DIRECT	Worst	4	13892.6043	502
7	DIRECT	Worst	8	13965.2474	1442
7	DIRECT	Worst	16	14768.9843	636
7	MADS	None	1	14698.5798	656
7	DIRECT	None	1	14768.9843	636
8	LHS	None	1	114019.0481	7
8	MADS	Best	2	11226.5238	1021
8	MADS	Best	4	16790.9924	1022
8	MADS	Best**	8	35710.1382	1511
8	MADS	Best*	16	22803.9641	1267
8	MADS	Average	2	5048.8643	2000
8	MADS	Average	4	20627.8337	1998
8	MADS	Average	8	40420.98	1756
8	MADS	Average	16	58416.4031	1878

8	MADS	Worst	2	11226.5238	1021
8	MADS	Worst	4	22003.091	1511
8	MADS	Worst	8	41498.8584	1756
8	MADS	Worst	16	59599.4774	902
8	DIRECT	Best	2	9735.7598	1929
8	DIRECT	Best	4	10749.1504	1974
8	DIRECT	Best	8	15072.2732	1510
8	DIRECT	Best	16	14408.4017	1607
8	DIRECT	Average	2	10817.2496	989
8	DIRECT	Average	4	9061.4397	1996
8	DIRECT	Average	8	11797.9532	1753
8	DIRECT	Average	16	20899.982	1558
8	DIRECT	Worst	2	10123.2826	1955
8	DIRECT	Worst	4	13892.6043	502
8	DIRECT	Worst	8	13965.2474	1442
8	DIRECT	Worst	16	14768.9843	636
8	MADS	None	1	59599.4774	902
8	DIRECT	None	1	14768.9843	636
9	LHS	None	1	128827.0857	28
9	MADS	Best	2	12367.9602	1020
9	MADS	Best	4	16077.5224	1997
9	MADS	Best	8	38018.0494	1754
9	MADS	Best*	16	51199.0009	1024
9	MADS	Average	2	8804.1045	1999
9	MADS	Average	4	17078.766	1021
9	MADS	Average	8	28625.5359	1019
9	MADS	Average	16	51199.0009	902
9	MADS	Worst	2	13963.3686	2000
9	MADS	Worst	4	18361.1552	1511
9	MADS	Worst	8	43365.432	1511
9	MADS	Worst	16	51199.0009	1268
9	DIRECT	Best	2	9735.7598	1929
9	DIRECT	Best	4	10749.1504	1974
9	DIRECT	Best	8	15529.879	1686
9	DIRECT	Best	16	14408.4017	1607
9	DIRECT	Average	2	10817.2496	989
9	DIRECT	Average	4	9061.4397	1996
9	DIRECT	Average	8	11797.9532	1753
9	DIRECT	Average	16	20899.982	1558

9	DIRECT	Worst	2	10123.2826	1955
9	DIRECT	Worst	4	13892.6043	502
9	DIRECT	Worst	8	13965.2474	1442
9	DIRECT	Worst	16	14768.9843	636
9	MADS	None	1	51199.0009	1268
9	DIRECT	None	1	14768.9843	636
10	LHS	None	1	98451.6178	37
10	MADS	Best	2	2027.6514	1999
10	MADS	Best	4	18822.1714	2000
10	MADS	Best	8	50258.2507	1266
10	MADS	Best	16	66483.7929	1389
10	MADS	Average	2	6111.4945	1999
10	MADS	Average	4	13104.1444	1999
10	MADS	Average	8	21958.1719	1753
10	MADS	Average	16	32384.7059	1633
10	MADS	Worst	2	5026.9945	1996
10	MADS	Worst	4	13821.7428	1021
10	MADS	Worst	8	12453.3345	1512
10	MADS	Worst	16	21570.6742	779
10	DIRECT	Best	2	9735.7598	1929
10	DIRECT	Best	4	10749.1504	1974
10	DIRECT	Best	8	15529.879	1686
10	DIRECT	Best	16	11853.7632	1607
10	DIRECT	Average	2	10817.2496	989
10	DIRECT	Average	4	9061.4397	1996
10	DIRECT	Average	8	11797.9532	1753
10	DIRECT	Average	16	20899.982	1558
10	DIRECT	Worst	2	10123.2826	1955
10	DIRECT	Worst	4	13892.6043	502
10	DIRECT	Worst	8	13965.2474	1442
10	DIRECT	Worst	16	14768.9843	636
10	MADS	None	1	21570.6742	779
10	DIRECT	None	1	14768.9843	636
11	LHS	None	1	77020.0895	19
11	MADS	Best	2	4049.6688	2000
11	MADS	Best	4	27037.931	1999
11	MADS	Best	8	22182.9182	1265
11	MADS	Best	16	35820.8084	1268
11	MADS	Average	2	14385.2613	1998

11	MADS	Average	4	13752.6031	2000
11	MADS	Average	8	28733.0937	1998
11	MADS	Average	16	35820.8084	1024
11	MADS	Worst	2	3091.5434	2000
11	MADS	Worst	4	12487.4675	1021
11	MADS	Worst	8	22415.8226	1754
11	MADS	Worst	16	35820.8084	902
11	DIRECT	Best	2	9735.7598	1929
11	DIRECT	Best	4	10749.1504	1974
11	DIRECT	Best	8	15529.879	1686
11	DIRECT	Best	16	11853.7632	1607
11	DIRECT	Average	2	10817.2496	989
11	DIRECT	Average	4	9061.4397	1996
11	DIRECT	Average	8	11797.9532	1753
11	DIRECT	Average	16	20899.982	1558
11	DIRECT	Worst	2	10123.2826	1955
11	DIRECT	Worst	4	13892.6043	502
11	DIRECT	Worst	8	13965.2474	1442
11	DIRECT	Worst	16	14768.9843	636
11	MADS	None	1	35820.8084	902
11	DIRECT	None	1	14768.9843	636
12	LHS	None	1	98735.4481	8
12	MADS	Best	2	12560.4999	1020
12	MADS	Best	4	30925.1091	2000
12	MADS	Best	8	30276.4304	1755
12	MADS	Best*	16	59206.85	780
12	MADS	Average	2	12560.4999	1999
12	MADS	Average	4	18357.3289	1997
12	MADS	Average	8	35096.1112	2000
12	MADS	Average	16	64202.4352	780
12	MADS	Worst	2	3135.497	2000
12	MADS	Worst	4	17278.8681	2000
12	MADS	Worst	8	20259.2705	777
12	MADS	Worst	16	39841.5436	534
12	DIRECT	Best	2	9735.7598	1929
12	DIRECT	Best	4	10749.1504	1974
12	DIRECT	Best	8	15529.879	1686
12	DIRECT	Best	16	14408.4017	1607
12	DIRECT	Average	2	10817.2496	989

12	DIRECT	Average	4	9061.4397	1996
12	DIRECT	Average	8	11797.9532	1753
12	DIRECT	Average	16	20899.982	1558
12	DIRECT	Worst	2	10123.2826	1955
12	DIRECT	Worst	4	13892.6043	502
12	DIRECT	Worst	8	13965.2474	1442
12	DIRECT	Worst	16	14768.9843	636
12	MADS	None	1	39841.5436	534
12	DIRECT	None	1	14768.9843	636
13	LHS	None	1	82846.8922	25
13	MADS	Best	2	3897.5187	2000
13	MADS	Best	4	8882.9256	1999
13	MADS	Best	8	22881.6055	1755
13	MADS	Best	16	34937.1119	1756
13	MADS	Average	2	10435.8045	2000
13	MADS	Average	4	22892.3467	2000
13	MADS	Average	8	35831.7571	2000
13	MADS	Average	16	44837.9881	1999
13	MADS	Worst	2	10435.8045	1021
13	MADS	Worst	4	23966.9375	1511
13	MADS	Worst	8	35831.7571	1756
13	MADS	Worst	16	29908.4202	1268
13	DIRECT	Best	2	9735.7598	1929
13	DIRECT	Best	4	10749.1504	1974
13	DIRECT	Best	8	15529.879	1686
13	DIRECT	Best	16	11853.7632	1607
13	DIRECT	Average	2	10817.2496	989
13	DIRECT	Average	4	9061.4397	1996
13	DIRECT	Average	8	11797.9532	1753
13	DIRECT	Average	16	20899.982	1558
13	DIRECT	Worst	2	10123.2826	1955
13	DIRECT	Worst	4	13892.6043	502
13	DIRECT	Worst	8	13965.2474	1442
13	DIRECT	Worst	16	14768.9843	636
13	MADS	None	1	29908.4202	1268
13	DIRECT	None	1	14768.9843	636
14	LHS	None	1	52035.7585	39
14	MADS	Best	2	3423.1298	1998
14	MADS	Best	4	16719.1401	1508

14	MADS	Best	8	19270.6778	1511
14	MADS	Best*	16	12847.1789	1268
14	MADS	Average	2	5835.8203	1999
14	MADS	Average	4	12736.8877	1999
14	MADS	Average	8	9912.9516	2000
14	MADS	Average	16	12846.6039	1756
14	MADS	Worst	2	4159.4841	1995
14	MADS	Worst	4	6810.7923	1022
14	MADS	Worst	8	9912.9516	1511
14	MADS	Worst	16	12846.6039	779
14	DIRECT	Best	2	9735.7598	1929
14	DIRECT	Best	4	10749.1504	1974
14	DIRECT	Best	8	15529.879	1686
14	DIRECT	Best	16	14408.4017	1607
14	DIRECT	Average	2	10817.2496	989
14	DIRECT	Average	4	9061.4397	1996
14	DIRECT	Average	8	11797.9532	1753
14	DIRECT	Average	16	20899.982	1558
14	DIRECT	Worst	2	10123.2826	1955
14	DIRECT	Worst	4	13892.6043	502
14	DIRECT	Worst	8	13965.2474	1442
14	DIRECT	Worst	16	14768.9843	636
14	MADS	None	1	12846.6039	779
14	DIRECT	None	1	14768.9843	636
15	LHS	None	1	96241.8448	25
15	MADS	Best	2	14221.5542	1020
15	MADS	Best	4	32690.3856	531
15	MADS	Best	8	12406.8857	1754
15	MADS	Best	16	35127.3139	1756
15	MADS	Average	2	5615.08	2000
15	MADS	Average	4	32690.3856	2000
15	MADS	Average	8	43318.7781	1756
15	MADS	Average	16	58143.2423	778
15	MADS	Worst	2	8411.4292	2000
15	MADS	Worst	4	23572.4047	1996
15	MADS	Worst	8	24564.8465	1997
15	MADS	Worst	16	58143.2423	1389
15	DIRECT	Best	2	9735.7598	1929
15	DIRECT	Best	4	10749.1504	1974

15	DIRECT	Best	8	15529.879	1686
15	DIRECT	Best	16	11853.7632	1607
15	DIRECT	Average	2	10817.2496	989
15	DIRECT	Average	4	9061.4397	1996
15	DIRECT	Average	8	11797.9532	1753
15	DIRECT	Average	16	20899.982	1558
15	DIRECT	Worst	2	10123.2826	1955
15	DIRECT	Worst	4	13892.6043	502
15	DIRECT	Worst	8	13965.2474	1442
15	DIRECT	Worst	16	14768.9843	636
15	MADS	None	1	58143.2423	1389
15	DIRECT	None	1	14768.9843	636
16	LHS	None	1	76142.4611	14
16	MADS	Best	2	3778.9461	2000
16	MADS	Best	4	16821.7138	1999
16	MADS	Best	8	30590.112	1509
16	MADS	Best*	16	40070.7244	1755
16	MADS	Average	2	2692.4158	1998
16	MADS	Average	4	4994.0721	1999
16	MADS	Average	8	17202.1027	1756
16	MADS	Average	16	40070.7244	1023
16	MADS	Worst	2	2979.2549	1996
16	MADS	Worst	4	18947.1022	531
16	MADS	Worst	8	30590.112	1264
16	MADS	Worst	16	40070.7244	1267
16	DIRECT	Best	2	9735.7598	1929
16	DIRECT	Best	4	10749.1504	1974
16	DIRECT	Best	8	15529.879	1686
16	DIRECT	Best	16	14408.4017	1607
16	DIRECT	Average	2	10817.2496	989
16	DIRECT	Average	4	9061.4397	1996
16	DIRECT	Average	8	11797.9532	1753
16	DIRECT	Average	16	20899.982	1558
16	DIRECT	Worst	2	10123.2826	1955
16	DIRECT	Worst	4	13892.6043	502
16	DIRECT	Worst	8	13965.2474	1442
16	DIRECT	Worst	16	14768.9843	636
16	MADS	None	1	40070.7244	1267
16	DIRECT	None	1	14768.9843	636

17	LHS	None	1	68990.1922	21
17	MADS	Best	2	2462.111	1999
17	MADS	Best	4	12359.3161	1021
17	MADS	Best	8	18763.9307	1018
17	MADS	Best*	16	31441.7229	901
17	MADS	Average	2	5943.3015	2000
17	MADS	Average	4	12359.3161	2000
17	MADS	Average	8	18763.9307	1753
17	MADS	Average	16	31441.7229	1877
17	MADS	Worst	2	5943.3015	2000
17	MADS	Worst	4	12359.3161	1021
17	MADS	Worst	8	18763.9307	1508
17	MADS	Worst	16	31072.0056	780
17	DIRECT	Best	2	9735.7598	1929
17	DIRECT	Best	4	10749.1504	1974
17	DIRECT	Best	8	15529.879	1686
17	DIRECT	Best	16	14408.4017	1607
17	DIRECT	Average	2	10817.2496	989
17	DIRECT	Average	4	9061.4397	1996
17	DIRECT	Average	8	11797.9532	1753
17	DIRECT	Average	16	20899.982	1558
17	DIRECT	Worst	2	10123.2826	1955
17	DIRECT	Worst	4	13892.6043	502
17	DIRECT	Worst	8	13965.2474	1442
17	DIRECT	Worst	16	14768.9843	636
17	MADS	None	1	31072.0056	780
17	DIRECT	None	1	14768.9843	636
18	LHS	None	1	125081.2018	20
18	MADS	Best	2	3901.9693	1999
18	MADS	Best	4	9873.3994	1506
18	MADS	Best	8	23255.7223	1022
18	MADS	Best**	16	63598.4628	2000
18	MADS	Average	2	4912.4692	1997
18	MADS	Average	4	12776.7253	1999
18	MADS	Average	8	48705.0571	1756
18	MADS	Average	16	63598.4628	1878
18	MADS	Worst	2	9296.5736	1999
18	MADS	Worst	4	20653.2876	1507
18	MADS	Worst	8	37387.5861	1756

18	MADS	Worst	16	72468.8334	779
18	DIRECT	Best	2	9735.7598	1929
18	DIRECT	Best	4	10749.1504	1974
18	DIRECT	Best	8	15529.879	1686
18	DIRECT	Best	16	20664.3989	1756
18	DIRECT	Average	2	10817.2496	989
18	DIRECT	Average	4	9061.4397	1996
18	DIRECT	Average	8	11797.9532	1753
18	DIRECT	Average	16	20899.982	1558
18	DIRECT	Worst	2	10123.2826	1955
18	DIRECT	Worst	4	13892.6043	502
18	DIRECT	Worst	8	13965.2474	1442
18	DIRECT	Worst	16	14768.9843	636
18	MADS	None	1	72468.8334	779
18	DIRECT	None	1	14768.9843	636
19	LHS	None	1	66149.0056	15
19	MADS	Best	2	6501.52	1998
19	MADS	Best	4	13876.1777	1511
19	MADS	Best	8	33647.8449	1267
19	MADS	Best*	16	45348.0696	780
19	MADS	Average	2	2137.663	1991
19	MADS	Average	4	7693.0882	2000
19	MADS	Average	8	33036.5408	1020
19	MADS	Average	16	44124.7653	780
19	MADS	Worst	2	3819.6181	1999
19	MADS	Worst	4	13500.2916	1018
19	MADS	Worst	8	25795.3438	532
19	MADS	Worst	16	44124.7653	534
19	DIRECT	Best	2	9735.7598	1929
19	DIRECT	Best	4	10749.1504	1974
19	DIRECT	Best	8	15529.879	1686
19	DIRECT	Best	16	14408.4017	1607
19	DIRECT	Average	2	10817.2496	989
19	DIRECT	Average	4	9061.4397	1996
19	DIRECT	Average	8	11797.9532	1753
19	DIRECT	Average	16	20899.982	1558
19	DIRECT	Worst	2	10123.2826	1955
19	DIRECT	Worst	4	13892.6043	502
19	DIRECT	Worst	8	13965.2474	1442

19	DIRECT	Worst	16	14768.9843	636
19	MADS	None	1	44124.7653	534
19	DIRECT	None	1	14768.9843	636
20	LHS	None	1	75793.7674	14
20	MADS	Best	2	5704.5172	1997
20	MADS	Best	4	23448.8777	1510
20	MADS	Best	8	38606.2656	777
20	MADS	Best*	16	50342.4926	656
20	MADS	Average	2	4182.6282	1993
20	MADS	Average	4	12300.7596	1998
20	MADS	Average	8	31383.1818	1022
20	MADS	Average	16	34419.5881	1994
20	MADS	Worst	2	4165.0149	1996
20	MADS	Worst	4	21505.4105	1511
20	MADS	Worst	8	38606.2656	1512
20	MADS	Worst	16	50342.4926	1756
20	DIRECT	Best	2	9735.7598	1929
20	DIRECT	Best	4	10749.1504	1974
20	DIRECT	Best	8	15529.879	1686
20	DIRECT	Best	16	14408.4017	1607
20	DIRECT	Average	2	10817.2496	989
20	DIRECT	Average	4	9061.4397	1996
20	DIRECT	Average	8	11797.9532	1753
20	DIRECT	Average	16	20899.982	1558
20	DIRECT	Worst	2	10123.2826	1955
20	DIRECT	Worst	4	13892.6043	502
20	DIRECT	Worst	8	13965.2474	1442
20	DIRECT	Worst	16	14768.9843	636
20	MADS	None	1	50342.4926	1756
20	DIRECT	None	1	14768.9843	636
21	LHS	None	1	112143.6876	34
21	MADS	Best	2	4891.9085	1997
21	MADS	Best	4	22000.7344	1511
21	MADS	Best	8	34341.1172	1512
21	MADS	Best*	16	53920.5863	1998
21	MADS	Average	2	11393.683	1999
21	MADS	Average	4	12133.5762	1999
21	MADS	Average	8	24925.1365	1754
21	MADS	Average	16	51367.89	1876

21	MADS	Worst	2	2954.1201	1996
21	MADS	Worst	4	20794.4235	1022
21	MADS	Worst	8	34633.9393	1512
21	MADS	Worst	16	51367.89	1876
21	DIRECT	Best	2	9735.7598	1929
21	DIRECT	Best	4	10749.1504	1974
21	DIRECT	Best	8	15529.879	1686
21	DIRECT	Best	16	14408.4017	1607
21	DIRECT	Average	2	10817.2496	989
21	DIRECT	Average	4	9061.4397	1996
21	DIRECT	Average	8	11797.9532	1753
21	DIRECT	Average	16	20899.982	1558
21	DIRECT	Worst	2	10123.2826	1955
21	DIRECT	Worst	4	13892.6043	502
21	DIRECT	Worst	8	13965.2474	1442
21	DIRECT	Worst	16	14768.9843	636
21	MADS	None	1	51367.89	1876
21	DIRECT	None	1	14768.9843	636
22	LHS	None	1	67745.2463	38
22	MADS	Best	2	10680.4214	2000
22	MADS	Best	4	21208.6959	2000
22	MADS	Best	8	32580.5738	1267
22	MADS	Best*	16	44382.8266	1755
22	MADS	Average	2	10680.4214	1021
22	MADS	Average	4	11067.4429	1998
22	MADS	Average	8	33312.1644	1756
22	MADS	Average	16	44262.5686	780
22	MADS	Worst	2	6501.1593	1999
22	MADS	Worst	4	20664.8945	1021
22	MADS	Worst	8	33957.7666	1511
22	MADS	Worst	16	44262.5686	780
22	DIRECT	Best	2	9735.7598	1929
22	DIRECT	Best	4	10749.1504	1974
22	DIRECT	Best	8	15529.879	1686
22	DIRECT	Best	16	14408.4017	1607
22	DIRECT	Average	2	10817.2496	989
22	DIRECT	Average	4	9061.4397	1996
22	DIRECT	Average	8	11797.9532	1753
22	DIRECT	Average	16	20899.982	1558

22	DIRECT	Worst	2	10123.2826	1955
22	DIRECT	Worst	4	13892.6043	502
22	DIRECT	Worst	8	13965.2474	1442
22	DIRECT	Worst	16	14768.9843	636
22	MADS	None	1	44262.5686	780
22	DIRECT	None	1	14768.9843	636
23	LHS	None	1	64318.9423	36
23	MADS	Best	2	3686.3102	2000
23	MADS	Best	4	17569.5357	1507
23	MADS	Best	8	30091.2277	1509
23	MADS	Best*	16	38594.7822	1509
23	MADS	Average	2	9648.74	1021
23	MADS	Average	4	13246.219	1509
23	MADS	Average	8	17716.9043	1022
23	MADS	Average	16	38594.7822	1875
23	MADS	Worst	2	10145.05	1999
23	MADS	Worst	4	9539.8268	1022
23	MADS	Worst	8	21470.3378	1022
23	MADS	Worst	16	38594.7822	1509
23	DIRECT	Best	2	9735.7598	1929
23	DIRECT	Best	4	10749.1504	1974
23	DIRECT	Best	8	15529.879	1686
23	DIRECT	Best	16	14408.4017	1607
23	DIRECT	Average	2	10817.2496	989
23	DIRECT	Average	4	9061.4397	1996
23	DIRECT	Average	8	11797.9532	1753
23	DIRECT	Average	16	20899.982	1558
23	DIRECT	Worst	2	10123.2826	1955
23	DIRECT	Worst	4	13892.6043	502
23	DIRECT	Worst	8	13965.2474	1442
23	DIRECT	Worst	16	14768.9843	636
23	MADS	None	1	38594.7822	1509
23	DIRECT	None	1	14768.9843	636
24	LHS	None	1	79397.5792	31
24	MADS	Best	2	12028.1625	1999
24	MADS	Best	4	24221.0147	1997
24	MADS	Best	8	28217.9016	1756
24	MADS	Best*	16	42056.4008	1756
24	MADS	Average	2	12028.1625	1999

24	MADS	Average	4	22740.6981	1021
24	MADS	Average	8	29958.4449	1020
24	MADS	Average	16	42056.4008	1634
24	MADS	Worst	2	9347.61	1999
24	MADS	Worst	4	20943.3593	1022
24	MADS	Worst	8	26270.6149	1511
24	MADS	Worst	16	41362.2181	780
24	DIRECT	Best	2	9735.7598	1929
24	DIRECT	Best	4	10749.1504	1974
24	DIRECT	Best	8	15529.879	1686
24	DIRECT	Best	16	14408.4017	1607
24	DIRECT	Average	2	10817.2496	989
24	DIRECT	Average	4	9061.4397	1996
24	DIRECT	Average	8	11797.9532	1753
24	DIRECT	Average	16	20899.982	1558
24	DIRECT	Worst	2	10123.2826	1955
24	DIRECT	Worst	4	13892.6043	502
24	DIRECT	Worst	8	13965.2474	1442
24	DIRECT	Worst	16	14768.9843	636
24	MADS	None	1	41362.2181	780
24	DIRECT	None	1	14768.9843	636
25	LHS	None	1	122591.8946	19
25	MADS	Best	2	2146.6965	1998
25	MADS	Best	4	14157.918	1022
25	MADS	Best	8	20702.1772	1022
25	MADS	Best*	16	20025.018	1755
25	MADS	Average	2	2585.7674	1993
25	MADS	Average	4	8479.2254	1995
25	MADS	Average	8	20727.1273	1754
25	MADS	Average	16	40258.4746	1877
25	MADS	Worst	2	1955.4537	1984
25	MADS	Worst	4	5372.4197	1020
25	MADS	Worst	8	10568.0493	1510
25	MADS	Worst	16	21365.7257	779
25	DIRECT	Best	2	9735.7598	1929
25	DIRECT	Best	4	10749.1504	1974
25	DIRECT	Best	8	15529.879	1686
25	DIRECT	Best	16	14408.4017	1607
25	DIRECT	Average	2	10817.2496	989

25	DIRECT	Average	4	9061.4397	1996
25	DIRECT	Average	8	11797.9532	1753
25	DIRECT	Average	16	20899.982	1558
25	DIRECT	Worst	2	10123.2826	1955
25	DIRECT	Worst	4	13892.6043	502
25	DIRECT	Worst	8	13965.2474	1442
25	DIRECT	Worst	16	14768.9843	636
25	MADS	None	1	21365.7257	779
25	DIRECT	None	1	14768.9843	636

Table A.3: Sometimes, knowing whether an idea is good is kind of like a game.