AALBORG UNIVERSITY

STUDENT REPORT

Learn Smarter, Not Harder: Improving Uppaal Stratego through Preprocessing

Project Group:
Kasper Kohsel Terndrup
Simon Vandel Sillesen

Supervisors:
Kim Guldstrand Larsen
Peter Gjøl Jensen
Andreas Berre Eriksen

Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg Ø
http://www.cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Learn Smarter, Not Harder: Improving Uppaal Stratego through Preprocessing

Project period:
Spring semester 2018

Project group:
deis107f18

Participants:
Kasper Kohsel Terndrup
Simon Vandel Sillesen

Supervisors:
Kim Guldstrand Larsen
Peter Gjøl Jensen
Andreas Berre Eriksen

Pages: 62

Date of completion:
June 1, 2018

Abstract:
The UPPAAL STRATEGO tool can synthesize near-optimal strategies for Priced Timed Markov Decision Processes. However, model elements that are irrelevant or redundant for the optimal strategy, can mislead the synthesis by needlessly increasing the state space. In this thesis, we propose a preprocessing addition to the UPPAAL STRATEGO algorithm, that can provide relief for redundancy and irrelevance in the synthesis. The addition enables the application of Principal Component Analysis or Fast Correlation Based Filter with the intention of reducing or removing irrelevant and redundant elements from models. We conduct a series of experiments, and show that preprocessing can improve strategy synthesis, in terms of better strategy performance and reduced size of the produced strategies. The results provide a basis for the inclusion of preprocessing capabilities, in the future development of UPPAAL STRATEGO.

Preface

This is the thesis of Kasper Kohsel Terndrup and Simon Vandel Sillesen, produced as the conclusive project of our Master's degree in Software Engineering at the Department of Computer Science at Aalborg University.

We would like to extend our gratitude to our supervisors Kim Guldstrand Larsen, Peter Gjøl Jensen, and Andreas Berre Eriksen, for their guidance and support during the project.

Citations throughout the thesis follows the IEEE style, and the bibliography is placed near the end of the report, before the appendices.

Summary

Here follows a summary of the thesis in accordance with the submission requirements at Aalborg University.

The thesis starts off by introducing UPPAAL STRATEGO. It presents a Timed Automaton of a wind turbine, which is used throughout the thesis for examples and experiments. The syntactical and semantic elements of the model are defined and explained. The model is then transformed into a Timed Game and the concept of modelling a game with an environment is introduced. Then the model is again developed, this time into a Priced Timed Game and shortly after into a Priced Timed Markov Decision Process, in order to introduce the concept of optimizing strategies towards a qualitative goal.

After this model development, the UPPAAL STRATEGO algorithm is described and different elements of the tool is explained. This includes the idea of learning sub-strategies and the unpublished manual state transformation feature.

The focus of the report is then motivated through an example, which demonstrates that the synthesis can be improved through an analysis and an associated manual state transformation. This is developed into the stated problem of preprocessing the model information prior to learning, in order to enable the learning methods to reach a better result, in terms of strategy performance and size. Then the thesis is delimited from considering memory and speed during the synthesis.

When the problem has been introduced, the thesis justifies the choice of the two issues of irrelevance and redundancy, which are defined and explained with examples. The field of preprocessing is then outlined, followed by a consideration of related work and a clarification of the novelty of the thesis.

Subsequently, the techniques that will be used to resolve the issues of irrelevance and redundancy are put forth, and their workings explained. This includes considerations with the implementation of the preprocessing elements into the existing UPPAAL STRATEGO tool.

Once the chosen preprocessing approach is delineated, experiments are presented and the results of them displayed. This is followed by a discussion of the implications of those results, for the thesis and preprocessing in UPPAAL STRATEGO. At the end, the thesis is concluded, and future work is suggested.

Contents

1 Introduction

The creation of models is a common and useful approach for solving problems in the real world. It allows people to focus their perspectives on an issue, as well as create abstractions and generalizations in an attempt to resolve it. UPPAAL STRATEGO [1] is a part of the UPPAAL tool suite [2], that can automatically create a strategy for some problem within a given system. An example of such a system, could be a wind turbine where the owners would like to know how to control it such that it generates as much electricity as possible without risking destruction. UPPAAL STRATEGO creates strategies for such systems, by combining model checking elements from the UPPAAL tool suite with machine learning techniques.

However, while UPPAAL STRATEGO will in theory find near-optimal strategies, it is not always able to achieve this under constrained resources. We will in this thesis present model traits that makes synthesis of strategies problematic, and then propose a preprocessing addition to UPPAAL STRATEGO that alleviates this problem.

We will first examine the general problem which UPPAAL STRATEGO addresses, and then give an introduction to how UPPAAL STRATEGO solves it. Afterwards, in Chapter 3, we will describe the exact problem that is the focus of this thesis, and then present our proposed solution, which we evaluate through experiments.

2 Uppaal Stratego

In this chapter we examine UPPAAL STRATEGO in order to understand the domain of the thesis and its contributions. First we introduce the problem which UPPAAL STRATEGO solves, i.e. synthesis of strategies, and then we describe the algorithm that it applies to do so. The observations and definitions found in this entire chapter is adapted from previous work done [3], [4].

An example of a system where the application of UPPAAL STRATEGO can be helpful, is the control of a wind turbine. We will in the following sections develop such an example, and explain why we use UPPAAL STRATEGO for solving this problem. The example will be incrementally expanded as we go, and used throughout the thesis.

2.1 Timed Automata

The UPPAAL tool suite, which UPPAAL STRATEGO is part of, can be used to analyze a class of models called Timed Automata (TAs). A simple TA model of a wind turbine could look like Figure 2.1.

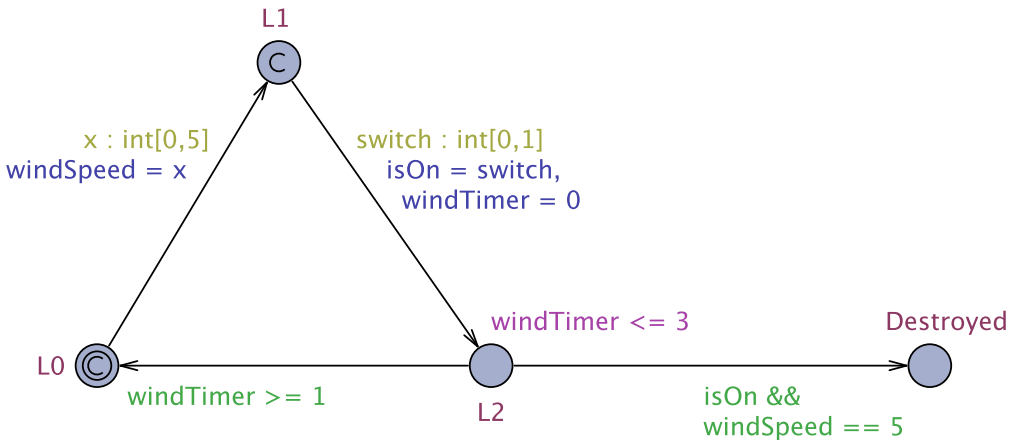

Figure 2.1: Example TA of a wind turbine. The variable declarations of the model can be seen in Listing 2.1.

```
1 int windSpeed = 0;
2 int isOn = 0;
3 clock time, windTimer;
```

Listing 2.1: Declarations for the TA wind turbine model.

The model follows a loop starting from its initial location, which is the circle marked with an inner circle and the name `L0` to the left of it. The first edge in the model is the assignment of the `windSpeed` variable. The syntax `x: int[0,5]` denotes that `x` is non-deterministically assigned an integer value from 0 to 5. It is equivalent to 6 edges where each edge assigns the respective integer to `x`. This edge models a sensor that reads the wind speed near the wind turbine. The second edge in the model happens from location `L1` to location `L2`. With this edge the wind turbine is either switched on or off. The locations `L0` and `L1`, from which the first two steps of the loop occur, are marked with a C. This means that time does not progress while we are in these "committed" locations, illustrating in this case, that the process of reading the wind speed and turning the wind turbine on or off, is so much faster than the rest of the model, that it can be regarded as instantaneous. On entering the location `L2`, we set our `windTimer` to zero. `windTimer` is a clock, which models the passing of time. In this case, `windTimer` is used to model the time between a possible change in the wind speed. We do this by declaring the "invariant" for the location to be `windTimer <= 3`, meaning that the system can only be in this state, if it adheres to this predicate. This illustrates that the wind speed is not static, and will change at least once every third time unit, whatever unit this might be. The last part of the loop models the wind turbine interacting with the wind, which has two possible outcomes. Either the wind turbine spins normally until the wind speed changes, or the wind turbine spins too fast and is destroyed. The possibilities of these outcomes are limited by the associated "guards". A guard is a predicate for an edge, which must be true before the edge can be followed. The leftmost guard, `windTimer >= 1`, denotes that the wind turbine will spin for at least one time unit before another windreading is taken. The rightmost guard, `isOn && windSpeed == 5`, denotes that the wind turbine risks destruction if the wind speed is 5 and the wind turbine is turned on. If the wind turbine is not destroyed, it will return to the initial location after 1 to 3 time units of spinning, at which point the process repeats itself. If the wind turbine is instead destroyed, it will go to the fourth location called `Destroyed`, from which no further action can be taken.

We use the formal definition of a TA given in [2][1]:

Definition 1. *A timed automaton is a tuple* $(L, \ell_0, X, \Sigma, E, I)$*, where* L *is a set of locations,* $\ell_0 \in L$ *is the initial location,* X *is the set of clocks,* Σ *is a set of actions* $E \subseteq L \times \Sigma \times B(X) \times 2^X \times L$ *is a set of edges between locations with an action, a guard, and a set of clocks to be reset, and* $I : L \to B(X)$ *assigns invariants to locations. Where* $B(X)$ *is the set of conjunctions over simple conditions of the form* $x \bowtie n \mid x - y \bowtie n$*, where* $x, y \in X$*,* $n \in \mathbb{N}$ *and* $\bowtie \in \{\leq, <, =, >, \geq\}$

The set of states of a TA is denoted as $Q \subseteq L \times \mathbb{R}^X_{\geq 0}$, where each state $q = (\ell, v) \in Q$ is a combination of a location $\ell \in L$ and a clock valuation function $v : X \to \mathbb{R}_{\geq 0}$, where $\mathbb{R}^X_{\geq 0}$ is the set of clock valuations. We use the notation $v \in \mathbb{R}^X_{\geq 0}$ to denote that v is a clock valuation for all clocks in X.

UPPAAL also allows an extension of TAs, where variables can be declared and used in guards and invariants, as well as mutated when edges in the model are followed. These

[1] We adapt naming to fit this thesis

variables, together with ℓ, make up the discrete part of a state, e.g. $q = (\{\ell, v1, v2\}, v)$. We use the notation $q.$<variable> as a shorthand for the value of that variable, e.g. $q.v1$ gives the valuation of $v1$ in state q.

The semantics of a TA are given in [2]:

Definition 2. *Let $(L, \ell_0, X, \Sigma, E, I)$ be a timed automaton. The semantics is defined as a labelled transition system $\langle Q, q_0, \rightarrow \rangle$, where $Q \subseteq L \times \mathbb{R}^X$ is the set of states, $q_0 = (\ell_0, v_0)$ is the initial state, and $\rightarrow \subseteq Q \times (\mathbb{R}_{\geq 0} \cup \Sigma) \times Q$ is the transition relation such that:*

- $(\ell, v) \xrightarrow{d} (\ell, v + d)$ *if* $\forall d' : 0 \leq d' \leq d \implies v + d' \in I(\ell)$, *and*
- $(\ell, v) \xrightarrow{a} (\ell', v')$ *if there exists* $e = (\ell, a, g, r, \ell') \in E$ *s.t.* $v \in g$, $v' = [r \mapsto 0]v$, *and* $v' \in I(\ell')$

where for $d \in \mathbb{R}_{\geq 0}$, $v + d$ maps each clock x in X to the value $v(x) + d$, and $[r \mapsto 0]v$ denotes the clock valuation which maps each clock in r to 0 and agrees with v over $X \setminus r$.

We can use UPPAAL to check if certain things are possible within this TA. We could for example ask the query:

> A[] not(WindTurbine.Destroyed)

which translates to "For all paths through the model, does it hold that we do not reach the location called destroyed?", or even simpler: "Is it always the case that the wind turbine is not destroyed?". In this case the answer is no, and UPPAAL can show us how it can occur.

Knowing that the wind turbine risks destruction is a start, but we would like to be able to prevent it all together. We could do this by analyzing the model and implementing a function $f(x)$ to control the on/off switch, which takes `windSpeed` as input and turns the wind turbine off when the speed is too high. However, imagine a more complex model of the wind turbine, with multiple factors, noisy sensors, and other problems that are present in the real world. In that case, the creation of f, might not be trivial and even if we create such a function, it might not be very good. So instead of doing that, we change our model from a TA to a Timed Game (TG).

2.2 Timed Games

The change from a TA to a TG is done by partitioning the actions Σ into two sets Σ_c and Σ_u. With this partition we differentiate between controllable and uncontrollable actions, which for our wind turbine example means, that the action that switches the wind turbine on or off is controllable, and the action that sets the `windSpeed` is uncontrollable. The actions that define the risk of destruction and the change in `windSpeed` are also uncontrollable. The graphical representation of this TG is shown in Figure 2.2, where the uncontrollable edges are dashed.

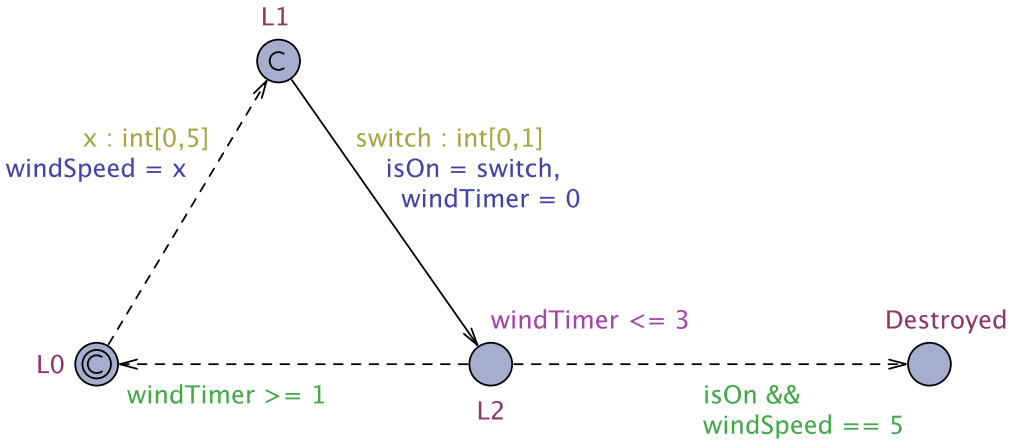

Figure 2.2: Example TG of a wind turbine. The variable declarations can be seen in Listing 2.2.

```
1 int windSpeed = 0;
2 int isOn = 0;
3 clock time, windTimer;
```

Listing 2.2: Declarations for the TG wind turbine model.

With this change to our model, we have changed our perspective on the system, such that we now consider it a game. In this game we are playing as the controller of the wind turbine against the environment, with the goal of not destroying the wind turbine. In order to learn how to play this game, we can apply the tool called UPPAAL TIGA [4] which solves TGs. We do this by supplying UPPAAL TIGA with the query:

> strategy safe = control: A[] not(WindTurbine.Destroyed)

This query translates to "Make a strategy called safe, that plays the controller such that any reachable state must not destroy the wind turbine". UPPAAL TIGA can give us a strategy for this wind turbine example. Such a strategy is a function $\sigma^c : Q \to 2^{\Sigma_c \cup \{\lambda\}}$, i.e. a function that given a state in our TG outputs a controllable action or λ meaning that no action should be taken at this point in time. In fact, UPPAAL TIGA gives us the most-permissive strategy, i.e. the largest strategy suggesting only actions that ensures safety. The most-permissive strategy for this example is:

$$\sigma^c(q) = \begin{cases} \text{switch} \in \{0,1\} & \text{if } q.\text{windSpeed} \neq 5 \\ \text{switch} \in \{0\} & \text{if } q.\text{windSpeed} = 5 \end{cases}$$

Here, $\sigma^c(q)$ defines the set of permitted actions for all $q \in Q$, where $q = (\ell, v)$. switch $\in \{0,1\}$ denotes the action where setting switch to 0 or 1 is permitted. We can

now make sure that the wind turbine is not destroyed during operation, if we follow the strategy given to us by UPPAAL TIGA. While this is definitely a positive result, we are not yet done. So far we have only been playing the game with the goal of not destroying the wind turbine. However, this can be easily achieved by simply never turning the wind turbine on, i.e. $\sigma^c(q) = \text{switch} \in \{0\}$, for all $q \in Q$. Obviously this is not a satisfying strategy, as we have a desire to generate power from the wind turbine, which is only done when the wind turbine is turned on. We include this qualitative goal into our wind turbine model, by changing the TG to a Priced Timed Game (PTG).

2.3 Priced Timed Games

We add the aspect of power generation to our TG in Figure 2.2 as a cost called `power`, which turns it into the PTG shown in Figure 2.3. The cost is a clock that can have different rates for different states. Changing this rate is denoted as `power' == <rate>`, where `<rate>` is an expression evaluating to a non-negative integer. We use this to measure the time spent generating power, i.e. when the wind turbine is turned on. Unfortunately, the general problem of solving PTGs is undecidable [5], which means that UPPAAL TIGA can not be used to solve it. However, because UPPAAL STRATEGO takes an approximation approach to PTGs, we can use it for this problem.

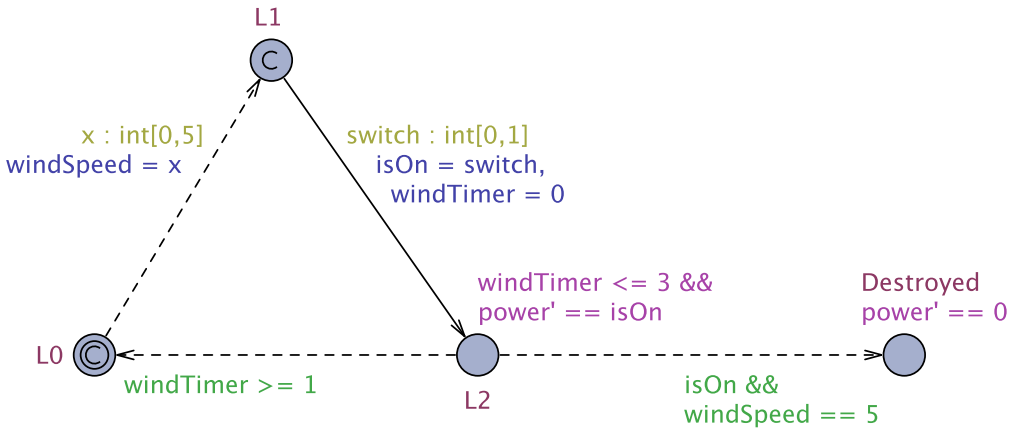

Figure 2.3: Example PTG of a wind turbine. The variable declarations can be seen in Listing 2.3.

```
1 int windSpeed = 0;
2 int isOn = 0;
3 clock time, windTimer;
4 hybrid clock power;
```

Listing 2.3: Declarations for the PTG wind turbine model.

Before we explain exactly how UPPAAL STRATEGO approaches the problem of PTGs, we must consider one more thing regarding our example model. The PTG wind turbine does not define how the uncontrollable actions are chosen. Because we want to model an actual environment, we use a stochastic strategy μ^u to model the real world behavior of said environment. In the case of real world wind speeds, they generally follow the Weibull distribution [6], however for the sake of simplicity we will assume a uniform distribution for all uncontrollable actions in our example. The combination of a PTG and the strategy μ^u, is a Priced Timed Markov Decision Process (PTMDP), which is defined based on the definitions given in [3] as:

Definition 3. *A pair* $M = \langle G, \mu^u \rangle$*, where*
$G = (L, \ell_0, X, \Sigma_c, \Sigma_u, E, P, I)$ *is a PTG where*
L *is a finite set of locations,* $\ell_0 \in L$ *is the initial location,*
X *is a finite set of non-negative real-valued clocks,*
$\Sigma_c \cup \Sigma_u$ *is a finite set of actions where*
Σ_c *is the controllable actions and* Σ_u *is the uncontrollable actions,*
$E \subseteq L \times B(X) \times 2^X \times L$ *is a finite set of edges,*
$P : L \to \mathbb{N}$ *assigns a price-rate to each location, and*
$I : L \to B(X)$ *sets an invariant for each location.*
The stochastic strategy μ^u *is a family of density-functions,* $\{\mu^u_q : \exists\, \ell\, \exists\, v.q = (\ell, v)\}$ *where* $\ell \in L$ *and* $v \in \mathbb{R}^X_{\geq 0}$*, with* $\mu^u_q(d,u) \in \mathbb{R}_{\geq 0}$ *assigning the density of the environment aiming at taking the uncontrollable action* $u \in \Sigma_u$ *after a delay of* d *from the state* q*.*

We have now reached the model type, for which UPPAAL STRATEGO can synthesize strategies. We do this with the following query:

```
strategy safe = control: A[] not(WindTurbine.Destroyed)
strategy opt = maxE(power) [<=100] : <> time > 99 under safe
```

This query can be read as “make a strategy called opt, that tries to maximize the expected value of power. Train the strategy from runs of length 100 or less, where time is greater than 99, while adhering to the safe strategy”. First of, we give an expression that should be optimized for, which in this case is to maximize `power`. Then we tell UPPAAL STRATEGO how to generate runs that can be used to learn the strategy. Here we denote that runs should last at most 100 time units, and that every path should satisfy the condition “time > 99”. The result for this query is that all runs used to learn, are exactly 100 time units long. Lastly we use the optional syntax “under safe”, to tell the strategy that it can only make choices within the bounds of our TG strategy safe, such that we still avoid the destruction of the wind turbine. In the next section we will describe how UPPAAL STRATEGO synthesizes this strategy from the description of the runs.

2.4 Strategy Synthesis

Now that the problem of synthesizing strategies has been outlined, we introduce the algorithm employed by UPPAAL STRATEGO, to generate the controller strategy. The algorithm is an iterative loop, which in each iteration performs a sequence of steps. First we outline the entire algorithm, and then we consider the central part in further detail. A visual illustration of the algorithm can be seen in Figure 2.4.

Simulation The first step is to simulate a number of runs through the model we are trying to learn a strategy for. For the first iteration we simulate with a uniform strategy, but later iterations will use the stochastic strategy from the previous iteration. A uniform strategy is a stochastic strategy that is equally likely to choose any action.

Filtering These runs are then filtered such that only runs exhibiting good performance, while satisfying the goal and time-bound requirements, are retained. We call these good runs.

Learning The good runs are used as input to a learning method, that learns a stochastic strategy from them.

Determinization The stochastic strategy is then determinized which results in a deterministic strategy.

Evaluation The stochastic strategy is then used for the simulation of runs in the first step of the next iteration, but only if the deterministic strategy exhibits better performance than the previous strategy. Performance is defined as the expected cost, which is the mean cost observed across the evaluation runs. After sufficiently many iterations, we exit the loop and determinize the best strategy μ_b^c that was found. The actual amount that qualifies as "sufficiently many", is determined by user input or default parameters

In each iteration of the algorithm, an, hopefully, improved strategy is learned from the sampled runs. The algorithm converges towards a strategy that is in a (local) minima w.r.t. expected cost. This method of learning a strategy based on a set of runs and then evaluating its overall performance in the system, makes the algorithm fall within the category of machine learning techniques called reinforcement learning.

Learning Strategies The central part of the algorithm is the learning method. When given a set of runs $\Pi = \{\pi_0, \pi_1 ... \pi_{n-1}\}$ it learns a stochastic strategy for the controller μ^c. A single run through a PTMDP is formally defined in [3] as:

Definition 4. *An alternating sequence of priced action and delay transitions of its priced transition system* $S : \pi = q_0 \xrightarrow{d_0}_{p_0} q_0' \xrightarrow{a_0}_0 q_1 \xrightarrow{d_1}_{p_1} q_1' \xrightarrow{a_1}_0 \cdots \xrightarrow{d_{n-1}}_{p_{n-1}} q_{n-1}' \xrightarrow{a_{n-1}}_0 q_n \cdots$, *where* $a_i \in \Sigma_c \cup \Sigma_u$, $d_i \in \mathbb{R}_{\geq 0}$ *is a delay,* $p_i \in \mathbb{R}_{\geq 0}$ *is a cost, and* q_i *is a state in the system.* $\sum_i p_i$ *is the total cost for the run.*

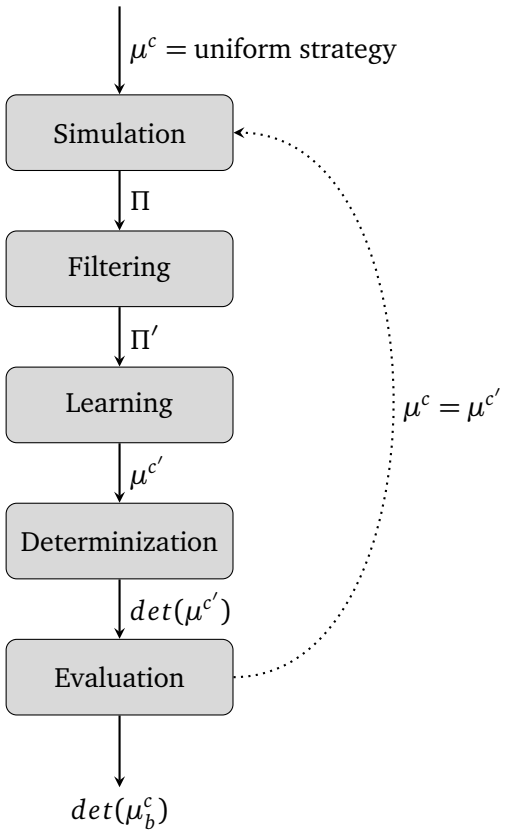

Figure 2.4: Depiction of general algorithm for learning a strategy. Adapted from [3]

This means that for every step in a run there is a delay d with a cost of p followed by an action a.

μ^c is defined in [3] as:

Definition 5. *A stochastic strategy μ^c for a PTMDP $M = \langle G, \mu^u \rangle$ is a family of density-functions, $\mu^c_q : \exists \ell \exists v.q = (\ell, v)$, with $\mu^c_q(d, c) \in \mathbb{R}_{\geq 0}$ assigning the density of the controller aiming at taking the controllable action $c \in \Sigma_c$ after a delay d from state q.*

Given a PTMDP one can define an expected-cost function for a controller strategy [3]. We denote this function $\mathbb{E}(\mu^c)_{\langle G,\mu^u \rangle} \in \mathbb{R}$ and omit specifying the PTMDP when it is obvious from the context. In [3] it is shown that a controller strategy μ^c can be found by learning sub-strategies μ^c_q for any state $q \in Q$. This is possible because μ^c can be a memoryless strategy, i.e. a strategy that only requires the latest state in a run in order to choose an optimal action, as opposed to requiring information about the previous states. UPPAAL STRATEGO currently considers sub-strategies for discrete states $\ell \in L$. We denote these sub-strategies as μ^c_ℓ. A sub-strategy μ^c_ℓ is learned from the information found in the set In_ℓ, which is defined as:

Definition 6. $In_\ell = \{ (s_n, v) \in (\Sigma_c \cup \mathbb{R}) \times \mathbb{R}^X_{\geq 0} \mid (q_0 \xrightarrow{s_0}_{p_0} \dots \xrightarrow{s_{n-1}}_{p_{n-1}} (\ell, v) \xrightarrow{s_n}_{p_n} \dots) \in \Pi \}$, *where (s_n, v) is an action-valuation pair, s_n is either a delay or an action, and $\mathbb{R}^X_{\geq 0}$ is the state space of the valuation v.*

This tells us, that the information relevant for learning a sub-strategy for a given discrete state ℓ is In_ℓ, which consists of all the valuations v that each represents the continuous part of a state (ℓ, v), paired with the action s_n that was taken from that particular state.

Lazy Strategies A current delimitation of UPPAAL STRATEGO is the lack of lazy strategies. A controller strategy $\mu^c_q(d, c) \in \mathbb{R}_{\geq 0}$, will assign the density of taking the controllable action $c \in \Sigma_c$ after a delay of d from state q. A non-lazy strategy will have the density $\mu^c_q(d, c) = 0$ for $d > 0$, meaning that the strategy will either suggest an urgent action or to yield in favor of the environment, possible delaying indefinitely which is denoted as action λ. The determinization of a stochastic controller strategy $\mu^c_q(d, c) \in \mathbb{R}_{\geq 0}$, assigns the density of an action with the highest probability a value of 1, and all other a value of 0, such that $det(\mu^c_q) : (\Sigma_c \cup \{\lambda\}) \to \{0, 1\}$.

Learning Methods UPPAAL STRATEGO currently has five different learning methods implemented. The first three original learning methods are described in [3], while the two newest learning methods are unpublished methods based on Q-learning [7] and model-based learning [8] respectively. We will in this report only concern ourselves with these two learning methods, referring to them as Q-learning and model-learning, as they seem more promising. Both algorithms can be used to iteratively learn strategies for Markov Decision Process (MDP).

The contributions are made to be independent from the learning methods, such that they can be can be compatible with future developments.

Manual State Transformation Another, unpublished, functionality in UPPAAL STRATEGO is manual state transformation. This allows a strategy query to specify which elements of the model are observable by the learning method, and hence will be taken into account of the learned strategy. An example of such a query for the wind turbine example in Figure 2.3, could be:

> strategy safe = control: A[] not(WindTurbine.Destroyed)
> strategy opt = maxE(power) [<=100] { WindTurbine.location } -> { windSpeed } :
> <> time > 99 under safe

A state $q \in Q$ was previously defined as a pair (ℓ, ν), with $\ell \in L$ being a location, and $\nu \in \mathbb{R}^X$ being a clock valuation, where $q.$<variable> is used to access variables in the state. Manual state transformation can be seen as a function $t : Q \to Q'$ that allows us to transform q and the accessible variables within it, into a new pair $t(q) = (k, u)$.

Here, $k : K \to \mathbb{R}$ denotes a mapping from the set of expressions K in the curly brackets left of the arrow, ->. $u : U \to \mathbb{R}$ denotes a mapping from the set of expressions U in the curly brackets to the right of the arrow. We use the notation $k \in \mathbb{R}^K$ and $u \in \mathbb{R}^U$, to denote valuations of the expressions. We will refer to manually partitioned sub-strategies as μ_k^c, and to instances of u feature vectors. By changing the expression that make up K, we can control how sub-strategies are partitioned, and through U we can define the values used for learning the sub-strategies.

In the query above, we have specified that a sub-strategy should exist for every different location in our model, and that all these sub-strategies should be trained solely by the value of the variable `windSpeed`. In this particular case, since the controller only takes actions from a single location, only one sub-strategy will be made, which could also have been achieved by leaving the first set of curly brackets empty. We will use manual state transformation to control what the learning method can learn on, which is useful for problems where certain variables are not observable in the real world, but required to simulate the model.

We will define In_k similarly to In_ℓ, but with t applied to (l, ν):

Definition 7. $In_k = \{(s_n, u) \in (\Sigma_c \cup \mathbb{R}) \times \mathbb{R}^U \mid (q_0 \xrightarrow{s_0}_{p_0} \dots \xrightarrow{s_{n-1}}_{p_{n-1}} (\ell, \nu) \xrightarrow{s_n}_{p_n} \dots) \in \Pi, (k, u) = t((l, \nu))\}$, *where* (s_n, u) *is an action and feature-vector pair, and* s_n *is either a delay or an action.*

In the rest of the thesis we will denote $q = (k, u)$ as a shorthand notation for $t(q) = (k, u)$.

User Parameters UPPAAL STRATEGO can be tuned through input parameters exposed the user, which affects how the learning algorithm behaves. In this thesis we will be using these to set the stopping criteria for when to exit the loop. We do this through the parameters: **good runs**, **total runs**, and **eval runs**.

Good runs controls how many runs can be used by the learning method. Increasing this will also increase processing time, but might also increase the performance of the learned strategy.

Total runs limits the amount of attempts UPPAAL STRATEGO can make towards generating the requested amount of **good runs**. Since not all runs will qualify as a **good run**, limiting the amount of **total runs**, will make sure that a strategy with a low chance of generating **good runs**, will not inflate the run-time too much, while trying to gather the requested amount.

Eval runs specifies how many runs has to be simulated in the evaluation step of the algorithm. A higher number increases the confidence of the evaluated expected cost.

Any future mentioning of setting runs to a specific value entails setting all these parameters to that specific value.

This concludes our summary of UPPAAL STRATEGO and the problem which it can be used to solve. The most important points to consider for the remainder of the thesis, is that UPPAAL STRATEGO finds strategies for PTMDPs through iterative reinforcement learning based on feature vectors.

3 Problem Statement

UPPAAL STRATEGO is a powerful tool, but not without its limits. We will in this chapter, through an example, introduce and motivate our focus on preprocessing as an approach to achieving better strategy synthesis.

Wind Turbine with Service The model in Figure 3.1 is an extension of the previously introduced wind turbine models. In this version, the wind turbine should still try to achieve the goal of generating power while avoiding destruction, but we have also added the concept of service. If the turbine is not serviced before the service timer exceeds 25, it can no longer be turned on, and therefore not generate power. However in order to service the wind turbine, it must be turned off, which means that no power can be generated for 1 time unit. The optimal safe strategy for this model is still to never have the turbine running while `windSpeed` is at a value of 5, but service should also be performed before the service timer exceeds 25. If the model is simulated for 100 time units, the optimal strategy will average an output of $\frac{5}{6}(100-3) = 80.8333$, where $\frac{5}{6}$ is the fraction of time `windSpeed` is not 5 and $100-3$ is the total time minus the time spent servicing.

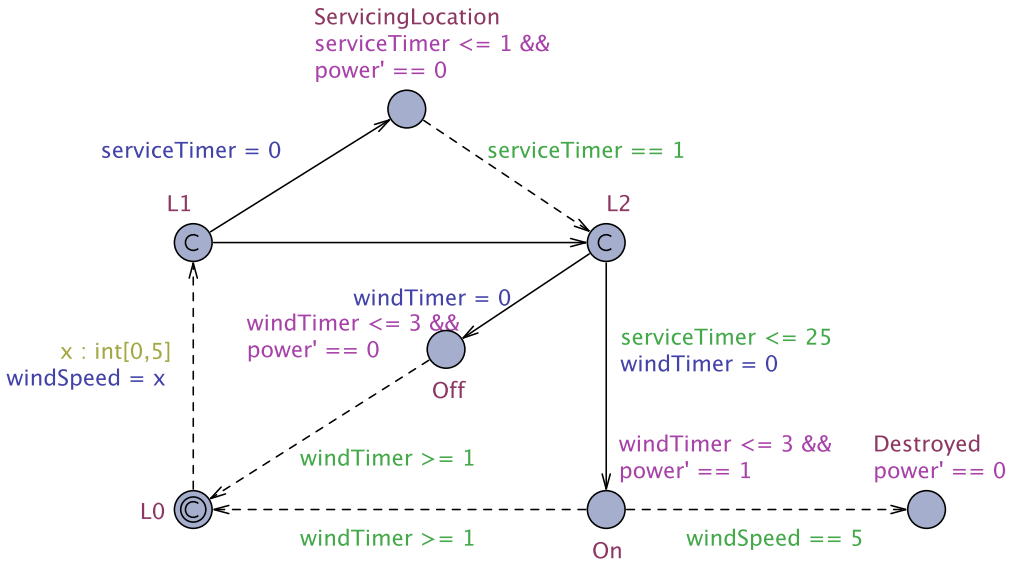

Figure 3.1: Wind turbine example with required periodic service. The variable declarations for this model can be seen in Listing 3.1.

In order to reach the optimal performance, the strategy must consider two choices. In L1 the strategy must decide if service is needed, and in L2 the strategy must decide whether to turn the wind turbine on or off. We can use UPPAAL STRATEGO to synthesize a

```
1 int windSpeed = 0;
2 clock windTimer, serviceTimer, time;
3 hybrid clock power;
```

Listing 3.1: Declarations for the wind turbine model with service.

strategy for this model, by using model-learning and runs set to 50, and then giving this query:

> strategy safe = control: A[] not(WindTurbine.Destroyed)
> strategy opt = maxE(power) [<=100] { WindTurbine.location } -> { windSpeed, serviceTimer, time, power, windTimer } : <> time > 99 under safe

Here we have partitioned the sub-strategies based on locations, and added the remaining variables to the feature vector. We repeated this query with 50 different seeds, and plotted the average power generated from the resulting strategies as the orange density in Figure 3.2. If we compare this to the optimal power generation, they can hardly be considered near-optimal. This could suggest that we should raise the amount of runs, in order to give UPPAAL STRATEGO enough information to find a near-optimal strategy. However, in the same figure we also see the result of running UPPAAL STRATEGO with the same parameters, except that the query now specifies a different manual state transform:

> strategy safe = control: A[] not(WindTurbine.Destroyed)
> strategy opt = maxE(power) [<=100] { WindTurbine.location } -> { windSpeed, serviceTimer } : <> time > 99 under safe

The results from this query, in the blue density, are quite a bit closer to an optimal strategy, despite not increasing the number of runs. In fact, the learning method had fewer variables to learn from. This result reflects the notion that not all data, available in a model, is beneficial for learning a near-optimal strategy. In this case, we examined the model, and saw that the two choices depended only on `serviceTimer` and `windSpeed` respectively.

Through this model analysis we were able to improve the synthesis, without having to increase the amount of runs. But if we imagine a more complex model with many choices depending on different combinations of variables, such an analysis becomes increasingly difficult, and it is clear that manual state transformation is not a perfect solution. Instead, we need some way of deciding which features to use for the learning method.

The Focus of the Thesis We believe that certain underlying issues are likely to be shared between models, and hypothesize that if these issues are resolved prior to applying the learning method, the synthesized strategies will be improved. We intend to test this hypothesis, by applying preprocessing techniques to UPPAAL STRATEGO in order to alleviate underlying model issues, and demonstrate an improved strategy synthesis.

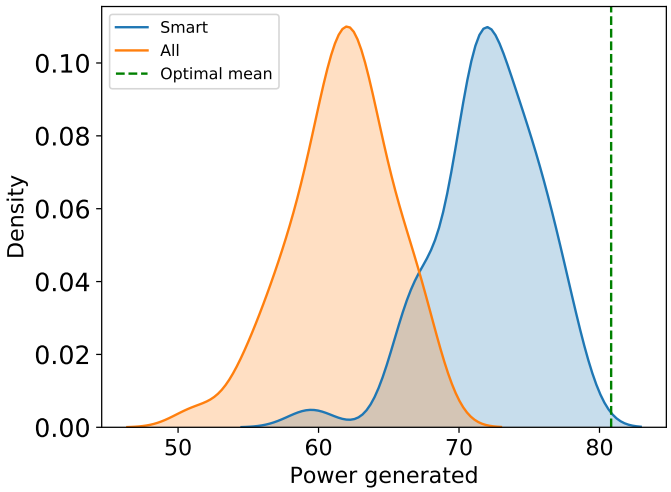

Figure 3.2: Plot showing average power generated for strategies with manually selected features and stratgies with all features

We choose to focus on preprocessing techniques, as they have been shown to lower storage requirements, improve speed of the learning algorithm, and increased predictive ability in learned models [9], and because it is an orthogonal approach to ongoing development of improved learning methods. We have formalized this intention into the following problem statement:

> How can preprocessing be used to alleviate model issues and thereby improve the synthesis of strategies for Priced Timed Markov Decision Processes by UPPAAL STRATEGO?

We will delimit the thesis from improvement of the run time of UPPAAL STRATEGO, as well as the memory required for the synthesis, and instead focus on the quality of the synthesized strategies. This means that we will not consider run time and memory in the conclusion.

In the coming chapters we will attempt to answer this problem statement by examining potential issues within UPPAAL models, and exploring known preprocessing techniques that may be used to combat these issues. We will then conduct experiments, in order to investigate how the potential issues affects the synthesis of strategies, and to which degree they can be alleviated by applying the preprocessing techniques. We then discuss the results of those experiments, and how well they show the merits of preprocessing, before finally concluding on the benefits of preprocessing in UPPAAL STRATEGO.

4 Data Issues

The previous chapter motivated the problem by showing that UPPAAL STRATEGO can have difficulties finding an optimal strategy for a model with data issues. There are many data issues that could potentially affect strategy synthesis. We will restrict the scope of this thesis to only consider the issues of irrelevance and redundancy. The following sections will define each issue and give examples, to achieve an understanding and concretisation of the problems.

4.1 Feature Irrelevance

There are many ways to define feature irrelevance [10]. Before we give our definition of irrelevance, let us introduce the shorthand notation for feature omission from a vector as follows. Let $u \in R^U$ be a sample, then we denote by $u' = u \setminus \{f\}$ the vector $u' \in R^{U \setminus \{f\}}$ s.t. for all $f' \in U \setminus \{f\}$ it holds that $u'(f') = u(f')$. Feature irrelevance is then defined as follows:

Definition 8. *A feature $f \in U$ is irrelevant if $\mathbb{E}(\mu^c_q) = \mathbb{E}(\mu^c_{q'}) + \epsilon$, where*
μ^c_q is an optimal stochastic strategy with $q = (k, u)$,
$\mu^c_{q'}$ is an optimal stochastic strategy with $q' = (k, u \setminus \{f\})$,
$k \in \mathbb{R}^K$, $u \in \mathbb{R}^U$, and $\epsilon \in \mathbb{R}$ is a small threshold value

Recall the motivating example from the problem statement, which is repeated in Figure 4.1. In this model, the only relevant features are `windSpeed` and `serviceTimer`. However their relevance is not equal across all sub-strategies; `serviceTimer` is relevant in location `L1`, but irrelevant in location `L2`. The opposite is true for `windSpeed`.

```
1 int windSpeed = 0;
2 clock windTimer, serviceTimer, time;
3 hybrid clock power;
```

Listing 4.1: Declarations for the wind turbine model with service.

With the precise description of irrelevance, we would say that `windTimer` is irrelevant to the optimal action c in `L2`, if $\mu^c_q = \mu^c_{q'} + \epsilon$, for all instances of $q = (k, u)$ and $q' = (k, u \setminus \{\texttt{windTimer}\})$ where `L2` $\in k$. In other words, knowing `windTimer` does not give any information about the optimal choice in `L2`.

The main motivation of eliminating irrelevant features is to reduce the state space the learning method has to learn from, which we expect will improve the performance of the learned strategy both in terms of accuracy and memory requirements.

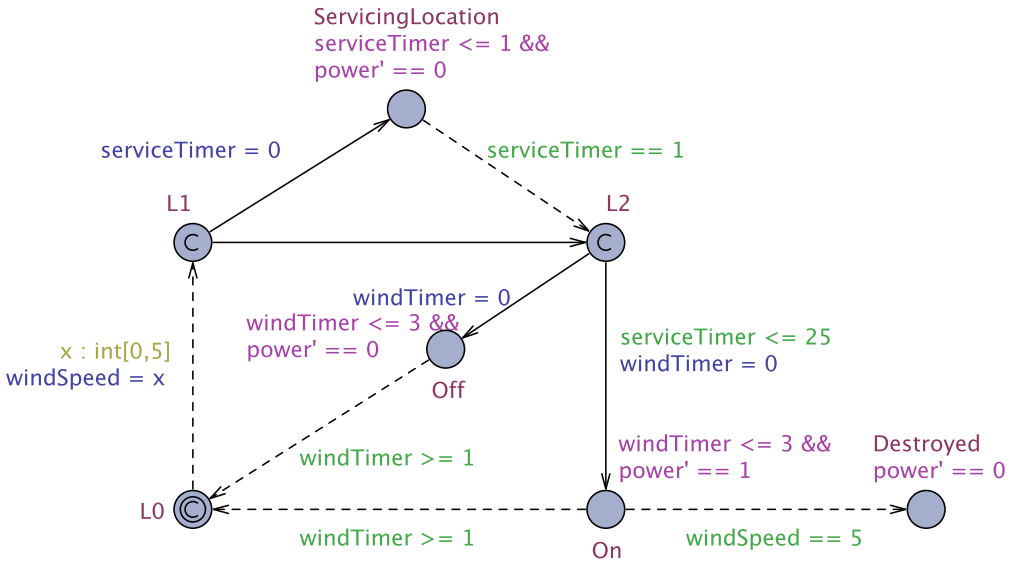

Figure 4.1: Wind turbine example with required periodic service, illustrating feature irrelevance. The variable declarations can be seen in Listing 4.1.

A secondary motivation, is that irrelevant features might appear relevant, which can lead to a sub-optimal strategy. For the example in Figure 4.1, a strategy for location `L1` that services the wind turbine when `windSpeed` is at a value of 3, will cause the controller to service the wind turbine once every sixth reading on average. Given that the time between readings is uniformly distributed between 1 and 3 and that service lasts 1 time unit, this strategy will service the turbine on average once every $1 + \frac{(1+3)/2}{1/6} = 13$ time unit. This means that with runs lasting 100 time units, service will be done on average $\lfloor \frac{100}{13} \rfloor = 7$ times, as opposed to the optimal strategy servicing only 3 times. This means, that the sub-optimal strategy will on average generate $\frac{5}{6}(100-7) = 77.5$ power, while the optimal strategy would average power generation at $\frac{5}{6}(100-3) = 80.8333$.

In other words, the sub-optimal strategy that on average suggests service almost twice as often as the optimal strategy will only see a performance decrease of $\frac{80.8333-77.5}{80.8333}100 = 4.123\%$. Because reinforcement learning values strategies based on outcomes, it is not unreasonable to fear that the strategy will deem `windSpeed` to be a relevant feature for deciding when to service the turbine.

4.2 Feature Redundancy

Recall that a feature f is irrelevant if it does not contribute information to the optimal stochastic strategy. Feature redundancy can be seen as a special case of feature irrelevance. A feature f_1 is redundant with a feature f_2 if f_1 is irrelevant for an optimal stochastic strategy when $f_2 \in U$, but relevant when $f_2 \notin U$. In other words, the optimal stochastic strategy does not not gain any information from f_1 if that information is already expressed

in f_2. A formal definition is given here:

Definition 9. *A feature f_1 is redundant to a feature f_2 if $\mathbb{E}(\mu^c_q) = \mathbb{E}(\mu^c_{q'}) + \epsilon \neq \mathbb{E}(\mu^c_{q''})$, where*
μ^c_q is an optimal stochastic strategy with $q = (k, u \setminus \{f_1\})$,
$\mu^c_{q'}$ is an optimal stochastic strategy with $q' = (k, u \setminus \{f_2\})$,
$\mu^c_{q''}$ is an optimal stochastic strategy with $q'' = (k, u \setminus \{f_1, f_2\})$,
$k \in \mathbb{R}^K$, $u \in \mathbb{R}^U$ and $\{f_1, f_2\} \subseteq U$, and $\epsilon \in \mathbb{R}$ is a small threshold value

Consider Figure 4.2 as an example of a UPPAAL model with redundant features. This wind turbine model is similar to the previously presented wind turbine models, but has access to several anemometers[1] instead of just one. The wind speed measurements can however be noisy. This noise for the example is generated by the *measureWind* function, based on the true wind speed reading x. The implementation is shown in Listing 4.3.

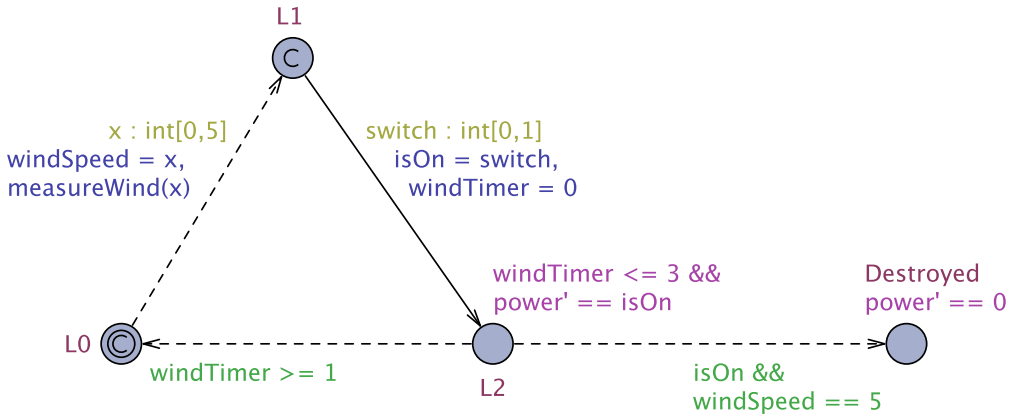

Figure 4.2: Wind Turbine model with redundant wind measurements. The variable declarations of the model can be seen in Listing 4.2.

```
1 const int num_anemometer = 30;
2 const double epsilon = 0.0;
3 double anemometers[num_anemometer];
4 int windSpeed;
5 int isOn = 0;
6 clock time, windTimer;
7 hybrid clock power;
```

Listing 4.2: Declarations for wind turbine model with redundant features.

The function simulates a number of wind readings by adding some amount of noise to the actual `windSpeed`, where the noise is randomly chosen between *-epsilon* and *epsilon*.

[1] A device used for measuring wind speed

```
1  void measureWind(int x) {
2      int i = 0;
3      double x_noisy = 0.0;
4      for (i = 0; i < num_anemometer; i++) {
5          double noise = random(2*epsilon);
6          // we want the noise to be between -epsilon to epsilon
7          noise = noise - epsilon;
8          x_noisy = x + noise;
9          anemometers[i] = x_noisy;
10     }
11 }
```

Listing 4.3: Implementation of measureWind(x)

The relationship between the true wind speed x and the noisy reading x_noisy is a linear dependency: $x\_noisy = x + \epsilon$. This relation is a specific kind of redundancy called collinearity. Two features are collinear if their relation can be described as $f_1 = a \cdot f_2 + b + \epsilon$, where a and b are constants and ϵ is a small amount of noise. In the current example $a = 1$ and $b = 0$.

Our motivation for considering collinearity as a data issue, is due to the fact that the clocks used in PTMDPs can easily be collinear for individual locations. An example of how this can occur can be seen in Figure 4.3. In this example the only edge going to L1 has a guard denoting that $x = 1$ and $y = 4$. This means that regardless of how long the environment waits in L1, the relation between the two clocks for either L2 or L3 is always $x = y + 3$.

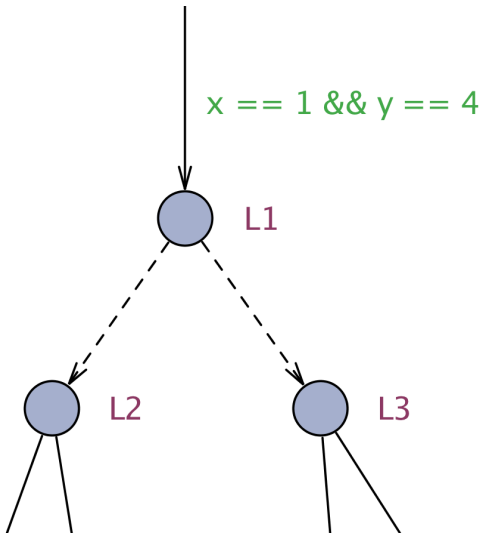

Figure 4.3: Example with collinear clocks

5 Preprocessing

Preprocessing is a central topic in machine learning. The overall concept is to induce some transformation in the data, that supports the learning process [11]. This process can take on many forms and provide different benefits. In this chapter we will give a short overview of preprocessing and present related work, before explaining the approaches we take in this thesis.

5.1 Categories of Preprocessing

Preprocessing is a broad term that can be attributed to a host of methods such as data cleaning [12], where information is reorganized and badly formed entries are taken care of. The kind of preprocessing we consider in this thesis are those which transform a feature vector $u \in \mathbb{R}^U$, into a feature vector $u' \in \mathbb{R}^M$, where $\mathbb{R}^M$ is a transformed state space with dimensionality M. We denote this transformation by the function $T : \mathbb{R}^U \to \mathbb{R}^M$. Such a function should attempt to remove irrelevant features or resolve redundancy, but preferably both. However, even though we intend to resolve the same underlying issues in different models, we are not able to tailor the preprocessing to specific cases, as different models can produce diverse versions of the issues.

As preprocessing is a common practice within machine learning and other fields, a multitude of different approaches have been proposed. Algorithms in the category called feature selection, attempt to reduce the dimensionality of the data by selecting or ranking the best features [13]. These algorithms can have significantly different optics on what constitutes the best features, but they all retain the features in their original form. The idea behind this approach is to keep features understandable, such that the selection itself can provide insight into the value of features. This possibility is not directly relevant for the purposes of this thesis, but it could be a useful tool in other uses of UPPAAL STRATEGO.

Another category of preprocessing algorithms, called feature extraction, does not consider such restrictions and freely modifies the features. This allows the algorithms to tackle problems such as noise reduction or discovery of latent features [14]. A particularly interesting extraction method is the autoencoder [15]. This technique uses a neural network to learn an identity function, with the caveat that an internal layer of the network exists has a lower dimensionality than the input. The intuition is that a neural network that can reduce the representation to the dimensionality of this internal layer, and then reproduce the original feature vector, must have found some intrinsic representation. Once the identity function has been learned, the computations from the input to the intrinsic internal layer is used as the feature extraction algorithm. Unfortunately, autoencoders generally require a considerable amount of hyperparameter tuning, which makes it less suitable for application across varying PTMDPs. Therefore, we choose to delimit us from further considering them in this thesis.

5.2 Related Work

We will in this section outline how related work have applied preprocessing techniques to reduce the dimensionality of MDPs. A brief description of their work will be presented, followed by a comparison to our problem.

A method of clustering states, such that the number of states are reduced has been proposed [16]. Initially, only one state is formed. New states are then gradually added when better clusters can be made. The clustering technique is then applied in conjunction with Q-learning.

Another proposed technique uses Neighbourhood Components Analysis (NCA) [17] as a linear projection to transform a high-dimensional state-space onto a low-dimensional space [18]. The low-dimensional space can then be used in a reinforcement learning setting to solve an MDP.

It has also been shown, that an application of Exponential family Principal Component Analysis (E-PCA) [19] can be used to reduce the dimensionality of an MDP [20]. It demonstrated improved learning performance in experiments.

All of the presented related works applies a preprocessing technique that reduces the number of states in an MDP. The novelty of this thesis is in the application of preprocessing to PTMDPs in the context of UPPAAL STRATEGO, with the goal of alleviating the data problems described in Chapter 4.

5.3 Preprocessing for Sub-Strategies

Recall that the synthesized strategies, created by UPPAAL STRATEGO, are made up of sub-strategies, and that these sub-strategies are learned from relevant information In_k, which has been previously defined as:

Definition 7. $In_k = \{(s_n, u) \in (\Sigma_c \cup \mathbb{R}) \times \mathbb{R}^U \mid (q_0 \xrightarrow{s_0}_{p_0} \dots \xrightarrow{s_{n-1}}_{p_{n-1}} (\ell, v) \xrightarrow{s_n}_{p_n} \dots) \in \Pi, (k, u) = t((l, v))\}$*, where* (s_n, u) *is an action and feature-vector pair, and* s_n *is either a delay or an action.*

Separating the runs into these sets, enables us to make certain observations that can be useful for preprocessing. While the feature vector u consists of the same variables across all sub-strategies, the distribution of a given variable can change between sub-strategies, as may the relations between the variables. Because of this, it is possible that the preprocessing opportunities are different for each sub-strategy, which is why our approach is to use sub-strategy specific transformations $T_k : \mathbb{R}^U \to \mathbb{R}^M$. T_k is learned based on the information contained in In_k. We will describe how a preprocessing function is learned later in this chapter. With this approach, a data problem that is only present in In_k for some k, can be addressed directly.

Given that the preprocessing is intended to improve sub-strategies μ_k^c, by transformation In_k through application of T_k, it occurs after the filtering step has gathered the runs Π' used to create In_k, and before the learning step. This is illustrated in Figure 5.1.

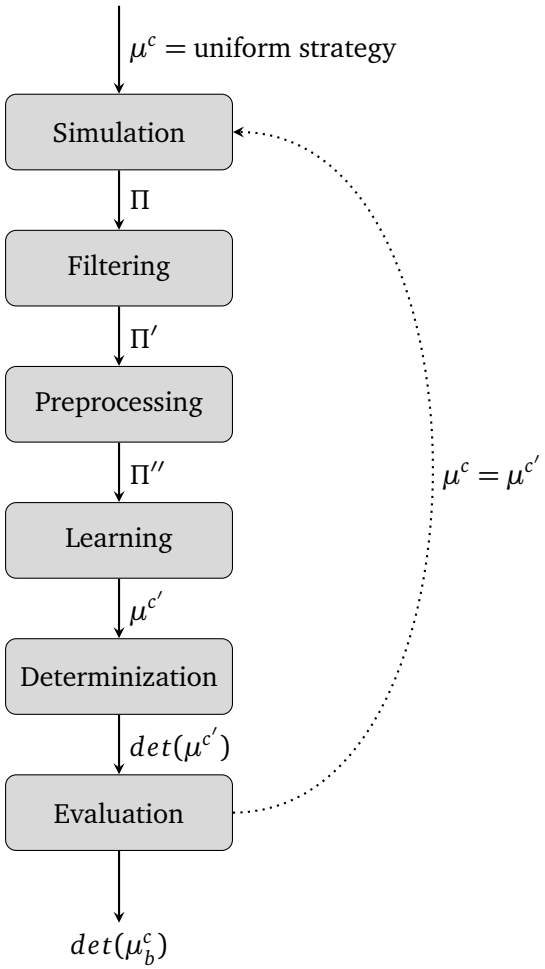

Figure 5.1: The UPPAAL STRATEGO learning algorithm with preprocessing introduced

An important thing to note, is that the application of preprocessing causes the strategies to no longer suggest actions based on a state $q = (k, u)$ in the model, but rather a transformed state $q' = (k, T(u))$. Therefore, the preprocessing function T_k must be kept with the strategy, such that q' can be obtained. This dictates that any performance benefits in μ_k^c that was gained by the application of T_k, should exceed the cost of representing or applying T_k.

5.4 Preprocessing Techniques

In this section we will discuss the algorithms that we use to combat the data issues. After introducing them, we will discuss some concerns regarding the integration of these techniques into the UPPAAL STRATEGO algorithm.

5.4.1 Principal Component Analysis

Principal Component Analysis (PCA) [21] is a feature extraction algorithm useful for reducing the dimensionality of a data space. The complexity of the implementation we use is $O(\max(N,U)^2 \cdot \min(N,U))$ [22], where N is the number of samples and U is the number of features.

We will first explain how PCA works and then consider how it can be used to eliminate redundancy in the form of collinearity in data, before giving some final considerations as to the application of PCA in UPPAAL STRATEGO.

PCA finds a projection of a matrix $D \in \mathbb{R}^{N \times U}$, where row i $d_{i,*} \in \mathbb{R}^U$ is a feature vector and N is the number of samples, to another matrix $\mathbb{R}^{N \times M}$, such that the variance is maximized along the axes in the projected space.

The variance along axis $d_{*,m} \in \mathbb{R}^N$ is given by

$$Var(d_{*,m}) = \frac{1}{N} \sum_{n \in N} (d_{n,m} - \overline{d_m})^2$$

Where $\overline{d_m}$ is the mean of $d_{*,m}$. The idea of maximizing variance is to capture as much information of the data as possible. Consider an axis with no variance, i.e. with all values being equal. No information is gained from this axis, and it can be removed without loss of information. When using PCA we assume that data values with a high variance contain a high degree of information, as they are more spread out from the mean.

Principal Components

The projection, which maximizes variance, is given by a set of principal component vectors that form an orthogonal basis. The first axis, i.e. the first principal component, expresses the greatest variance of the data. The second axis then expresses the second greatest variance, and so forth. We can project data set X onto this basis by matrix multiplication: $M = XB$.

It can be shown that the principal components are equal to the eigenvectors of the co-variance matrix of X [11]. The ordering of the principal components is found by the eigenvalues of each eigenvector, since a larger value means it captures more of the variance. If the original data set X contains redundant features, some of the the principal component will have a low corresponding eigenvalue. We can reduce the dimensionality of X by removing the principal components with eigenvalues below our desired threshold. One method for doing this would be to select principal components until a desired amount of variance is retained in the projected space, and then disregard the remaining principal components. The remaining principal components can then be used to project X to a new subspace M that is of lower dimensionality, while still preserving much of the variance.

When we use PCA for preprocessing, we find the principal components of In_k, and keep enough components to retain 90% of the variance, and use the transformation given by the basis to create $T_k(u) = uB_k$, where u is the feature vector and B_k is the principal components basis.

Removing Collinear Features

With this knowledge of PCA, we can now explain why it works well in resolving collinearity issues. Consider a data set D of 2 collinear features, f_1 and f_2 with the collinear relationship $f_2 = 2 \cdot f_1 + \epsilon$, where ϵ is a noise variable. Figure 5.2 gives a visual explanation of the relationship between f_1 and f_2, as well as displaying the principal components $PC1$ and $PC2$, which help in the following explanation.

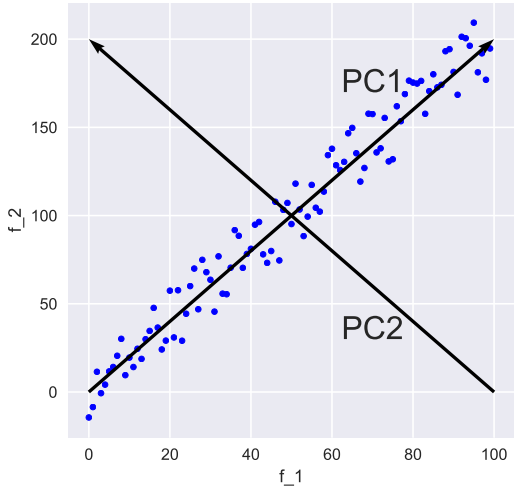

Figure 5.2: Principal components on collinear features

If we view both features as vectors u_1 and u_2, we could similarly write $u_2 = 2 \cdot u_1 + \epsilon$, which is a vector scaling. The magnitudes of the vectors are different, but they share the same general direction. This is a critical observation; Recall that PCA finds a projection such that the principal components are orthogonal to each other, while maximizing variance along its axis. Seeing as u_1 and u_2 have approximately the same direction, the first principal component would be approximately equal to u_2. The second principal component, orthogonal to the first, has a small eigenvalue, meaning it does not describe much variance. We can see that almost all of the variance of the data D can be described using only the first principal component. In fact, the only variance that is not retained is the noise introduced by ϵ.

5.4.2 Fast Correlation Based Filter

Fast Correlation Based Filter (FCBF) [23] is a feature selection algorithm designed to select a subset of relevant features from a larger set of features, such that the selected relevant features are not redundant. It can therefore be used to combat the feature redundancy and irrelevancy problems described in Chapter 4. The complexity of the algorithm is $O(N \cdot U \log U)$ [23].

We will in the following section describe how FCBF selects features in 2 phases: Phase 1 selects relevant features and phase 2 removes redundancies in the selected features.

Symmetrical Uncertainty

We have previously defined these problems from a probabilistic perspective. FCBF uses a different definition with an information theoretic perspective called *symmetrical uncertainty* for the same problems. Whereas the probabilistic perspective expressed the relationship as a probability, the information theoretic approach defines the relationship in terms of entropy. Symmetrical uncertainty is defined as:

$$
\begin{aligned}
SU(X,Y) &= 2\left[\frac{IG(X \mid Y)}{H(X)+H(Y)}\right], \text{where} \\
IG(X \mid Y) &= H(X) - H(X \mid Y), \\
H(X) &= -\sum_i P(x_i)\log_2 P(x_i), \\
H(X \mid Y) &= -\sum_j P(y_j)\sum_i P(x_i \mid y_j)\log_2(P(x_i \mid y_j))
\end{aligned}
$$

Symmetrical uncertainty is a symmetric measure of correlation between two discrete random variables X and Y, i.e. $SU(X,Y) = SU(Y,X)$. $IG(X \mid Y)$ is the information gain about X when Y is known, $H(X)$ is the entropy of X, and $H(X \mid Y)$ is the conditional entropy. $SU(X,Y) = 1$ means that one variable can be derived from the other. $SU(X,Y) = 0$ means X and Y are independent.

The use of entropy in this context, can be seen as a measure of uncertainty of a random variable. Consider a random variable A uniformly distributed over integers from 1 to 10, i.e. $P(A = i) = 0.1, \forall i \in \{1..10\}$. If we were to randomly draw a sample from A, we would have a high uncertainty of the sampled value as any integer from 1 to 10 is equally likely. Compare this with another random variable B, also distributed over integers from 1 to 10, where $P(B = 1) = 0.5$ and $P(B = i) = \frac{0.5}{9} = 0.056, \forall i \in \{2..10\}$. The entropy for A is $H(A) = 3.32$, whereas for B it is $H(B) = 2.5$. This shows that there is less uncertainty in B, and that we have a better chance of predicting the outcome of a random sample from B than from A.

The joint entropy $H(X,Y)$ describes the uncertainty that is present across the two random variables X and Y. If we obtain information on the outcome of the variable Y, we have removed $H(Y)$ amount of uncertainty. The remaining uncertainty in $H(X,Y)$ can then be calculated as $H(X,Y) - H(Y) = H(X \mid Y)$, i.e. the joint uncertainty minus the uncertainty we removed. $H(X \mid Y)$ is called conditional entropy. When the uncertainty of $H(Y)$ is removed from the joint entropy, we might gain information about the variable X. This information gain is calculated as $IG(X|Y) = H(X) - H(X|Y)$. Here we subtract the uncertainty that remains regarding X when Y becomes known, $H(X|Y)$, from the original uncertainty $H(X)$. In order to express this as $SU(X,Y)$ we normalize the information gained by the original amount of uncertainty in the random variables.

Phases of FCBF

Phase 1 of the FCBF algorithm selects relevant features. According to FCBF, a feature f is relevant to a label l if $SU(f,l) > \delta$, where δ is a user-defined threshold value. The selected features are ordered according to their relevance to the label, such that for $SU(f_i,l) > SU(f_j,l)$ it holds that $i < j$, where i and j are the sorted indices.

After phase 1, all irrelevant features have been eliminated, but the selected features can still be redundant. Phase 2 of FCBF tries to remove these redundant features. According to FCBF, features f_1 and f_2 are redundant if $SU(f_1,f_2) \geq \alpha$, where α is a threshold value. There is a trade-off when deciding what α should be. A too low value will eliminate many features, but the eliminated features might have been beneficial to retain. A too high value will eliminate few highly redundant features, but the selected features may still be redundant.

FCBF uses $\alpha = SU(f_2,l)$ based on the following belief. Consider two features f_1 and f_2, with a higher similarity than f_2 and the label l, i.e. $SU(f_1,f_2) > SU(f_2,l)$. Since the two features are more similar to each other than f_2 and l, it is likely that the information given by f_2 about l is already covered by f_1. This relies on the fact that the features are sorted in the first phase, such that $SU(f_1,l) > SU(f_2,l)$, which makes f_2 redundant.

FCBF Labels in Uppaal Stratego

FCBF requires both feature vectors and labels to function. A label is a prescribed value of a feature vector that FCBF uses to understand and organize them. We want the label to reflect the cost of taking an action in UPPAAL STRATEGO, such that FCBF can determine which features are influencing the cost.

Recall that a run π through a PTMDP is defined as:

Definition 4. *An alternating sequence of priced action and delay transitions of its priced transition system* $S : \pi = q_0 \xrightarrow[p_0]{d_0} q_0' \xrightarrow{a_0}_0 q_1 \xrightarrow[p_1]{d_1} q_1' \xrightarrow{a_1}_0 \dots \xrightarrow[p_{n-1}]{d_{n-1}} q_{n-1}' \xrightarrow{a_{n-1}}_0 q_n \dots$, *where* $a_i \in \Sigma_c \cup \Sigma_u$, $d_i \in \mathbb{R}_{\geq 0}$ *is a delay,* $p_i \in \mathbb{R}_{\geq 0}$ *is a cost, and* q_i *is a state in the system.* $\sum_i p_i$ *is the total cost for the run.*

We will use this definition to describe different choices of labels, their characteristics and finally conclude on the label we will use.

The first possibility is using immediate cost as a label. This is defined from π as p_i, when delaying d_i in state q_i. The problem with using immediate cost as a label becomes apparent, when the immediate cost is constant for a location. FCBF can not determine the relevant features, as the label is constant, regardless of the feature values.

Another possibility is to use the total cost of a run: $\sum_i p_i$. The problem with this approach, is that other actions in the run, both controllable and uncontrollable, also influences the total cost. This can make total cost a noisy label for actions, especially since the action in question has no impact on the development of the cost preceding it.

We can mitigate the above issues by using cost-to-completion as a label. This is defined from π as $p_f - p_i$, where p_f is the final cost in the run, and p_i is the cost of taking action-delay pair (a_i, d_i) in state q_i. Using cost-to-completion as a label resolves the situation

where the immediate costs are constant for a location. The noise problem of using total cost is reduced, as we only consider the future changes in cost after taking action-delay pair (a_i, d_i) from state q_i. Cost-to-completion does however have a significant problem, in that actions that occur later in a run will have a lower cost-to-completion. We therefore normalize the cost-to-completion based on the relative position in the run. This is defined as:

$$l = \frac{p_f - p_i}{f - i + 1}, \text{where } l \text{ is the label, and } f \text{ is the last state index}$$

Discretization in Uppaal Stratego

Symmetrical uncertainty is only defined for discrete variables, but features and costs in UPPAAL STRATEGO can be continuous. Consequently we have to apply a discretization step before FCBF that divides the continuous features into n classes, where n is a user-defined parameter. The discretization step of a feature is implemented by first normalizing the feature values to be between 0 and 1. These normalized values are then assigned to bins with equal sized intervals. So with 4 bins, bin 1 would contain values from 0 to 0.25, bin 2 from 0.25 to 0.5, bin 3 from 0.5 to 0.75 and bin 4 from 0.75 to 1.0.

5.4.3 Integrating Preprocessing in Uppaal Stratego

The algorithms described above both support dynamic variation of the dimension of the preprocessed data, based on the input data. In FCBF, dominant features can remove redundant features, and δ can be used to prune low-scoring features. Likewise, in PCA, we can choose principal components until 90% of the variance is retained. Letting the algorithms decide the dimensionality dynamically allows for optimal preprocessing in cases where the number of relevant features of two sub-strategies are different. However, this dynamic nature poses a problem for the existing UPPAAL STRATEGO algorithm. This section describes the problem as well as the chosen solution.

The Problem

Recall that UPPAAL STRATEGO uses an iterative algorithm for synthesizing strategies. In each iteration of the algorithm, a set of runs are simulated, and then filtered. With the addition of the preprocessing step, the runs are transformed, before the data is sent to the learning method. The iterative nature of the algorithm means that a learning methods can continuously update its understanding of the data, by reusing information from previous iterations.

Consider the following scenario of the UPPAAL STRATEGO algorithm with preprocessing:

Iteration 1

1. Runs are simulated. The dimensionality of the data is 10.

2. The data is now preprocessed such that the output dimensionality is 5.
3. The learning method learns from the preprocessed data. This entails that internal representations from now on expect a data dimensionality of 5.

Iteration 2

1. Runs are simulated. The dimensionality of the data is 10.
2. The data is now preprocessed such that the output dimensionality is 3.
3. The learning method learns from the preprocessed data. However, the expected dimensionality of 5 does not match the input dimensionality of 3. It is unclear how the learning method should interpret this data, as it is inconsistent with previously observed features.

The above scenario illustrates how varying dimensionality can pose as a problem to the current UPPAAL STRATEGO algorithm. A different issue from dimensionality change, is the fact that the meaning of features can also change across iterations. The FCBF algorithm can, for example, change the internal ordering of feature importance such that f_1 and f_2 are selected in iteration 1, but f_2 and f_3 are selected in iteration 2. In PCA, unstable axes can also change the meaning of features. If the direction of the principal components change across iterations, the features will have different meanings.

The Solution

To combat the problem, we need to make sure that the learning method is never given data that can change dimensionality or meaning across iterations. We suggest the following solution, which is also illustrated as pseudo code in Algorithm 1.

We divide each iteration in the original UPPAAL STRATEGO algorithm into two phases. The first phase is very similar to the original algorithm. In Line 8, we use the current best strategy μ^c to simulate a run. We then in Line 9 add a recording step to this phase, where every run simulated in the iteration is added to a history list. A preprocessing function is applied to the simulated run in Line 10, which μ^c is trained on in Line 11.

The second phase begins by resetting μ^c to a new untrained strategy in Line 14. In Line 15, a preprocessing transformation is learned from the recorded history of the current and all previous iterations. This transformation is then applied to the history of runs in Line 17. μ^c is then trained on the preprocessed run in Line 18.

The proposed solution solves the problem, but does so by making some trade-offs. More work is being done per iteration in phase 1, as there is the extra work of recording each run. In addition, in phase 2, a strategy is trained anew based on the complete history of runs, which will result in an increased run time. Memory requirements will also increase as the complete history of runs has to be kept in memory. Overall the algorithm will require more resources, but as run-time performance is not the primary focus of this thesis, we consider it an acceptable trade-off. In Chapter 9 we outline potential approaches that could reduce the overhead of the proposed solution.

Algorithm 1: Pseudo code for the UPPAAL STRATEGO algorithm with preprocessing

```
1  M ← The PTMDP we are learning
2  μ^c ← New uniform strategy
3  T ← Identity preprocessing function
4  Π ← ∅
5  foreach iteration do
6      // Phase 1
7      for NumberOfRuns do
8          π ← Simulate M with μ^c
9          Π = Π ∪ {π}
10         π_p ← T(π)
11         Train μ^c on π_p
12     end
13     // Phase 2
14     μ^c ← New uniform strategy
15     T ← Learn preprocessing on Π
16     foreach π in Π do
17         π_p ← T(π)
18         Train μ^c on π_p
19     end
20 end
```

6 Experiments

This chapter presents the experiments that we conducted to examine the data issues and suggested solutions from Chapter 5. These experiments are intended to explore the severity of the data issues for the learning methods, and to which degree the suggested solutions resolve the issues.

The chapter will start by outlining the measurement metrics for the experiments, and the reason for choosing them. Then the general test setup is introduced, before each experiment is outlined and evaluated. We will interpret and discuss the results in Chapter 7.

6.1 Evaluation Metrics

In Chapter 3 we state that preprocessing has potential to improve UPPAAL STRATEGO. However, without further specification, improvement is at least partially a subjective matter. Therefore, we will in the following section discuss three areas of improvement, and argue the choice of metrics within them.

Memory Requirements The first area with potential for improvement is memory. There are two memory concerns for UPPAAL STRATEGO: Memory used during the synthesis and the size of the learned strategy. We have chosen to delimit the project from considering the usage of memory during synthesis, as we have not focused on an efficient implementation of preprocessing. However, the size of the learned strategy is an interesting metric. Given that the strategies are represented as decision trees, preprocessing could reduce the number of nodes in the tree and thereby reduce the size of the strategy. We therefore use the amount of splits in a tree as the metric for strategy size.

Learning Speed Since preprocessing itself takes time, it would be interesting to see if the time it takes to learn a strategy is ultimately reduced, or if the time spent preprocessing exceeds the time saved during learning. However, once again, due to the lack of an efficient implementation of preprocessing, we will not use time as a metric for the experiments.

Quality of Strategies The ability to synthesize strategies that approach optimal behavior, is the most important feature of UPPAAL STRATEGO. Therefore, we also find it to be the most important metric for improvement in terms of preprocessing. Once a strategy has been synthesized, we will rate the quality of it by the expected cost. The expected cost is calculated by taking the highest value of cost in a run π and then averaging across

multiple repeated runs.

$$maxE = \frac{1}{N}\sum_{i=1}^{N} max_{cost}(\pi)$$

6.2 Setup of Experiments

Before moving on to the individual experiments, there are a few points to be made about the tests in general, which we will do in this section.

Choice of Models Recall that we want to explore the applicability of various preprocessing techniques for UPPAAL STRATEGO. As the work presented here is a preliminary study, we restrict ourselves to test the effects of the preprocessing techniques on the simple models from Chapter 4, that are constructed to exhibit common data issues. This is done in order to focus our evaluation on the issues in isolation. The observed effects can then be used to suggest appropriate preprocessing techniques for more complex UPPAAL models.

This raises the valid concern, that the techniques might not generalize very well, or that they lose effectiveness when the models become more complex. We discuss this issue further in the discussion in Chapter 7, and also refer to future work in Chapter 9.

Reliability of Results UPPAAL STRATEGO utilizes randomness in its algorithm, primarily during the generation of runs for learning or evaluation. This is problematic for evaluation as performance metrics can vary in-between experiments. To combat this, we want to factor out the effects of randomness by repeating experiments multiple times. Each repetition will be done with a different seed for the random actions, making them deterministic and therefore repeatable. Multiple repetitions of the experiments with different seeds, should give an understanding of the holistic performance of the preprocessing methods.

Another important thing to consider, is the amount of runs used in different parts of the algorithm. When given more runs to learn from, UPPAAL STRATEGO generally produces better strategies. However, additional runs also increases the time it takes to synthesize a strategy, and can therefore not be increased without caution.

We wish to show the effects of preprocessing in UPPAAL STRATEGO, when an appropriate amount of runs are available. However, deciding on an appropriate amount of runs is not trivial. If too many runs are used, UPPAAL STRATEGO might find equally good strategies with and without preprocessing. This could falsely imply that preprocessing had no benefit, even if preprocessing might have enabled UPPAAL STRATEGO to synthesize the same strategy with a lower amount of runs. On the other hand, it is also important to not use too few runs, as we do not want the outcomes of the experiments to be impacted by a significant lack of information during the synthesis.

To combat this, we will repeat the tests with different amounts of runs, in order to ensure that the evaluation of the results are consistent across different amounts of runs.

For these reasons, the experiments were conducted with 25, 50, and 100 runs, and all configurations repeated with 50 different seeds in order to factor out the effects of randomness during the learning process. The results from the experiments with 25 and 100 runs are consistent with the ones that have 50 runs, so only those will be shown in the report itself. The results from 25 and 100 runs can be found in Appendix A.

Analysis of Experiments In order to analyze the results of repeated experiments, we visualize the results as box plots [24]. An example of such a box plot can be seen in Figure 6.1.

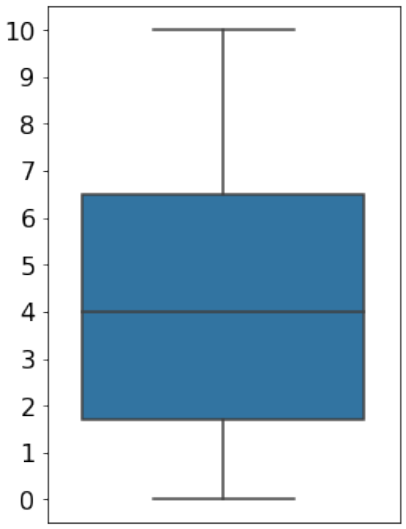

Figure 6.1: Box plot example.

In this plot we can see the median at 4, marked by the line through the blue box. The top and bottom ends of the blue box, at 6.5 and 1.8 respectively, marks the Inter Quartile Range (IQR), which is between the third and first quartiles, $Q3$ and $Q1$. Beyond the box, the whiskers at 10 and 0 mark the highest and lowest data point between $Q1 - 1.5(Q3 - Q1)$ and $Q3 + 1.5(Q3 - Q1)$. Data points beyond these are considered outliers that are not representative of expectable strategy performance, and are therefore not included.

We choose to visualize the data with box plots, as it allows us to consider the variety in the strategies produced during an experiment, and compare two different configurations, e.g. with preprocessing and without preprocessing. The whiskers shows us the range of strategy performance i.e. how stable the configuration is. The height of the box can then give us an idea, of how the performance is distributed within that range.

Every experiment will include a box plot for each configuration we test, for both of the learning methods we test. The blue box plot means Q-learning, while the orange box plot means model-learning. This means that there will always be two box plots for configurations without any preprocessing, two for the configuration with an identity

function for preprocessing, and then two for each of the preprocessing techniques we apply. The identity preprocessing function simply returns the feature vector without any changes, in order to see if the solution to integrating preprocessing in UPPAAL STRATEGO, explained in Section 5.4.3, has any an impact.

6.3 Collinear Redundancy Experiment

In order to evaluate the effects of collinear redundancy and our approach to alleviate it, we have conducted an experiment. In this section we will introduce the experiment, and afterwards present the results.

6.3.1 The Model

The experiment involves 16 different versions of the same model, for which we will synthesize strategies both with and without preprocessing. These variations are intended to explore the ability of the synthesis when dealing with different kinds of collinearity. The base model is the same that was introduced in Section 4.2, and shown here again in Figure 6.2.

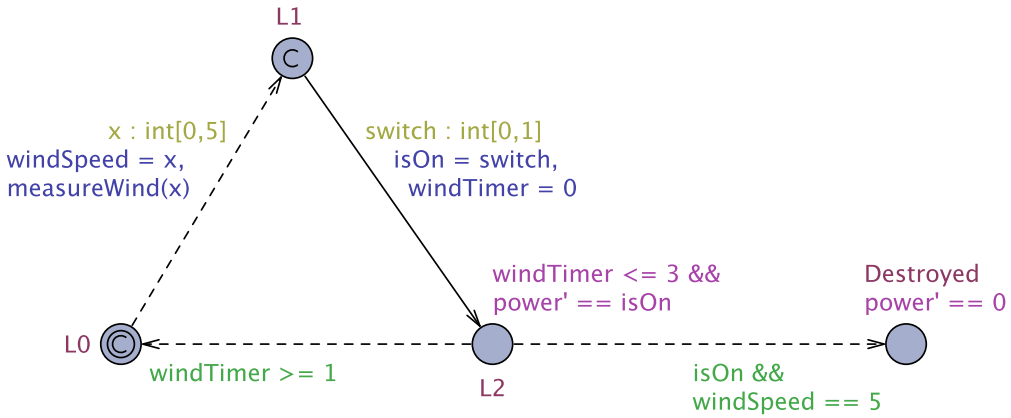

Figure 6.2: Wind Turbine model with redundant wind measurements. The variable declarations can be seen in Listing 6.1.

As mentioned previously, the model differs from the other wind turbine examples by introducing multiple anemometers instead of just a single one. This could model a wind farm where every turbine can get wind readings from the others. The anemometers are set with the *measureWind* function, shown in Listing 6.2.

The 16 different versions are found in the cartesian product of four different queries and four variations in the *measureWind* function. The four queries differ in their manually selected features. The variations in the *measureWind* function controls which type of collinearity that is present in the model, and does so through two variables. Any unique details in the variations will be discussed together with the evaluation of the results. Below is a more detailed description of the four queries along with the two variables.

```
const int num_anemometer = 30;
const double epsilon = 0.0;
double anemometers[num_anemometer];
int windSpeed;
int isOn = 0;
clock time, windTimer;
hybrid clock power;
```

Listing 6.1: Declarations for the model showcasing redundancy.

```
void measureWind(int x) {
    int i = 0;
    double x_noisy = 0.0;
    for (i = 0; i < num_anemometer; i++) {
        double noise = random(2*epsilon);
        // we want the noise to be between -epsilon to epsilon
        noise = noise - epsilon;
        x_noisy = x + noise; // With scaling: x*i + noise;
        anemometers[i] = x_noisy;
    }
}
```

Listing 6.2: Implementation of measureWind(x)

Select Perfect This query manually selects only the `windSpeed` variable, which is the only variable required to learn the optimal strategy. This query is included to show the results of learning when no redundancy is present.

> strategy opt = maxE(power) [<=100] { } -> { windSpeed } : <> time > 99 under safe

Select Wind With this query both the `windSpeed` variable and all the anemometer variables are selected. Results of this query shows the ability to learn when the required information is present, but among collinear features, making it harder to pick out the correct variable.

> strategy opt = maxE(power) [<=100] { } -> { windSpeed, anemometers[0], anemometers[1], ..., anemometers[29] } : <> time > 99 under safe

Select Noise This query only considers the anemometer variables. As such perfect information is no longer guaranteed, but can be inferred to some degree from the noisy variables.

> strategy opt = maxE(power) [<=100] { } -> { anemometers[0], anemometers[1], ..., anemometers[29] } : <> time > 99 under safe

Select All This query selects every variable in the model. This represents a scenario where some but not all features are collinear.

> strategy opt = maxE(power) [<=100] { } -> {windSpeed, time, windTimer, power, isOn, anemometers[0], anemometers[1], ..., anemometers[29] } : <> time > 99 under safe

Epsilon The first variable that introduces differences in *measureWind* is `epsilon`. If we want perfect collinearity we set this epsilon to 0. When we instead want to introduce some noise, we set epsilon to 1.0.

Scale The second variation is introduced by choosing which version of Line 8 in *measureWind* we use. If we use the version in the comment, we scale wind readings by their index. This is done to explore different types of collinearity.

The "safe" strategy that is referred to here, is the same that was introduced in Section 2.2, i.e.:

> strategy safe = control: A[] not(WindTurbine.Destroyed)

Every strategy created in this experiment is tested with the same query:

> E[<=100;100] (max:power) under opt

This asks UPPAAL to create 100 runs of 100 time units where it follows the strategy "opt", and report the expected maximum value of power. For this experiment we have chosen to use 30 anemometers. While this might seem excessive, we wanted to make sure that the results revealed the ability to handle collinearity and not just multiple features.

6.3.2 Presentation of the Results

We now present the results of the experiments, and evaluate the performance differences. We present the variations in groups of four with the same query, such that we can compare the impact of different levels of collinearity on the learning methods. Recall that the optimal strategy will switch on the wind turbine whenever the `windSpeed` is not 5, which gives an average expected cost of $\frac{5}{6}(100) = 83.333$. A lower median expected cost is the result of a sub-optimal strategy. A lower amount of splits is also preferable to a higher number of splits, with the caveat that it might have an adverse effect on expected cost if it is too low.

Select Perfect

The first results are the select perfect variations. The expected costs can be seen in Figure 6.3 and the number of splits in Figure 6.4. As the select perfect query causes the feature vector to only include the `windSpeed` variable, all of the variations present the same problem and the results are therefore expectedly identical. The only thing to note

here, is that the inclusion of superfluous preprocessing does not have an impact on the expected cost, and only a minor influence on range of splits, even though the median stays the same.

Select Wind

The next results are those where we used manual state transformation to learn on *windSpeed* and all the anemometers. The expected cost is shown in Figure 6.5 and number of splits in Figure 6.6. We see that the first and third variations with epsilon set to 0, also manage to find near-optimal strategies across all configurations. In the first variation the lack of both noise and scaling makes the anemometers perfectly describe wind, meaning that there are 31 variables with the same exact values. The third variation also has zero noise but does include scaling, which also results in near-optimal strategies across all configurations.

In the second and fourth variations, there is however a noticeable decrease in the ability of model-learning to handle the noise that is introduced, if no preprocessing is applied. If we look at the number of splits, there is also a significant increase in the number of splits for the strategies that perform badly.

Select Noise

The third results are shown in Figures 6.7 and 6.8, which cover the experiments where only the noisy anemometers are available to the learning methods. Here we again see that the first and third variations achieve near-optimal strategies across all configurations. As no noise is introduced, these two variations are expectedly almost identical to their counterparts for the select wind queries from Figures 6.5 and 6.6, and as such they do not reveal any new insights.

In the second and fourth variation we also see similar outcomes as the select wind results, where the model-learning method has lower expected costs and more splits without preprocessing. However, in the second variation we see that FCBF is not able to make model-learning reach a near-optimal strategy. Note that the `windSpeed` variable is not available and can therefore not be selected. PCA keeps its performance at the same level as the select wind query. In the fourth variation, the introduction of scaling enables FCBF to select features that allow model-learning to find a near-optimal strategy again. This is despite scaling negatively impacting the expected cost for model-learning strategies without preprocessing.

Select All

The last four variations of this experiment, which covers the select all queries, can be seen in Figure 6.9 and Figure 6.10.

Here we start to see a degradation in Q-learning, compared to the previous variation. The range is bigger, with some samples falling below 80, and the median has degraded from 83.5 to 82.5. Both PCA and FCBF brings Q-learning back to the near-optimal

expected cost, it had in the previous results. Recall that FCBF is designed to remove irrelevant features, while PCA is not. When scaling is introduced, there is also an increase in the number of splits in the strategies produced by Q-learning without preprocessing.

Model-learning continues to show a lower expected cost when no preprocessing is applied. Generally it seems to work best in combination with PCA, but FCBF also seems to make a noticeable improvement. An interesting thing to note, is that the second variation, where model-learning is combined with PCA, shows a dip in expected cost compared with the same configuration in other variations. If we look at the number of splits for the same variation and configuration, we see a significant spike in the number of splits.

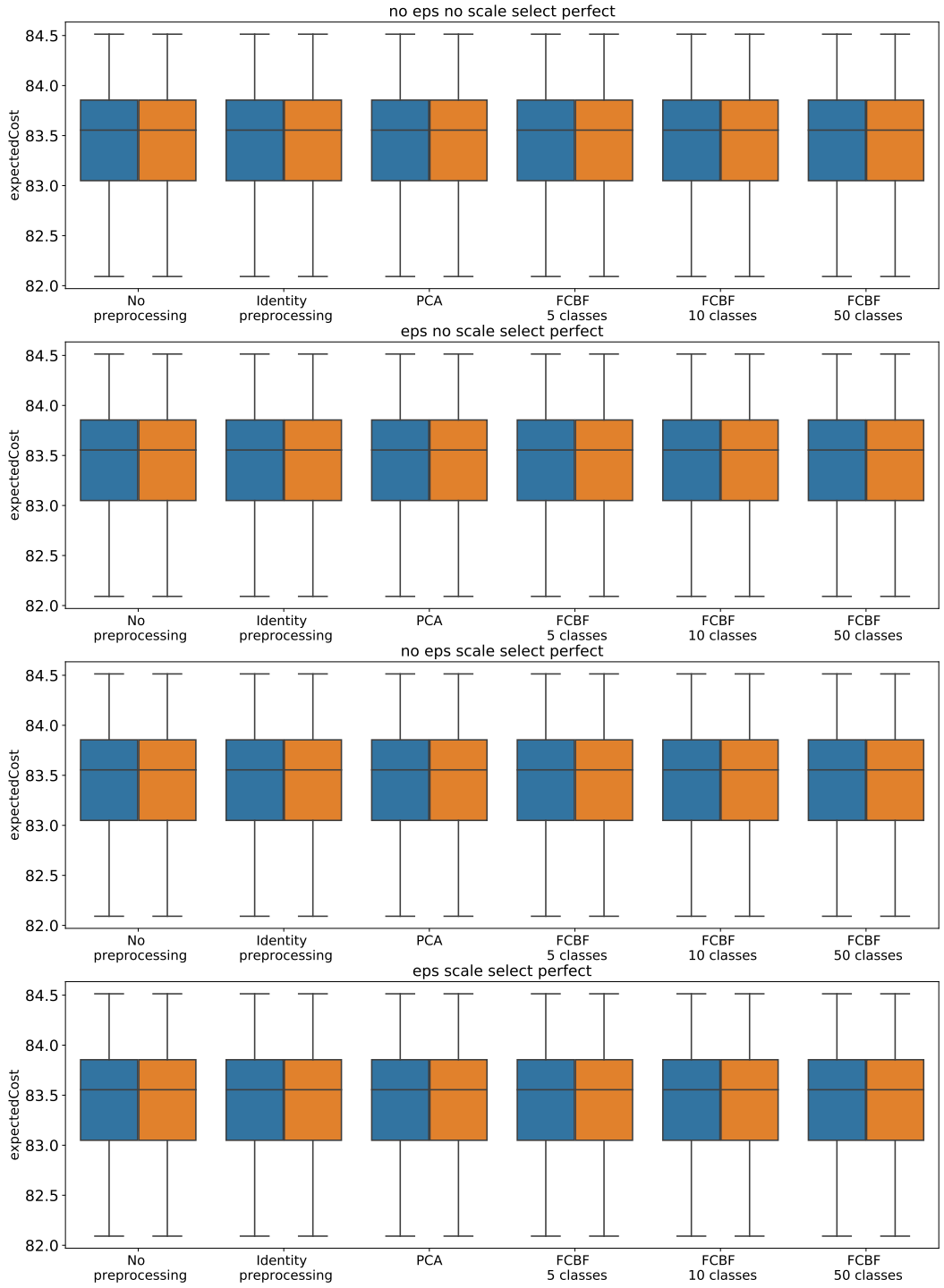

Figure 6.3: Expected Cost for redundancy experiment with perfect manual feature selection

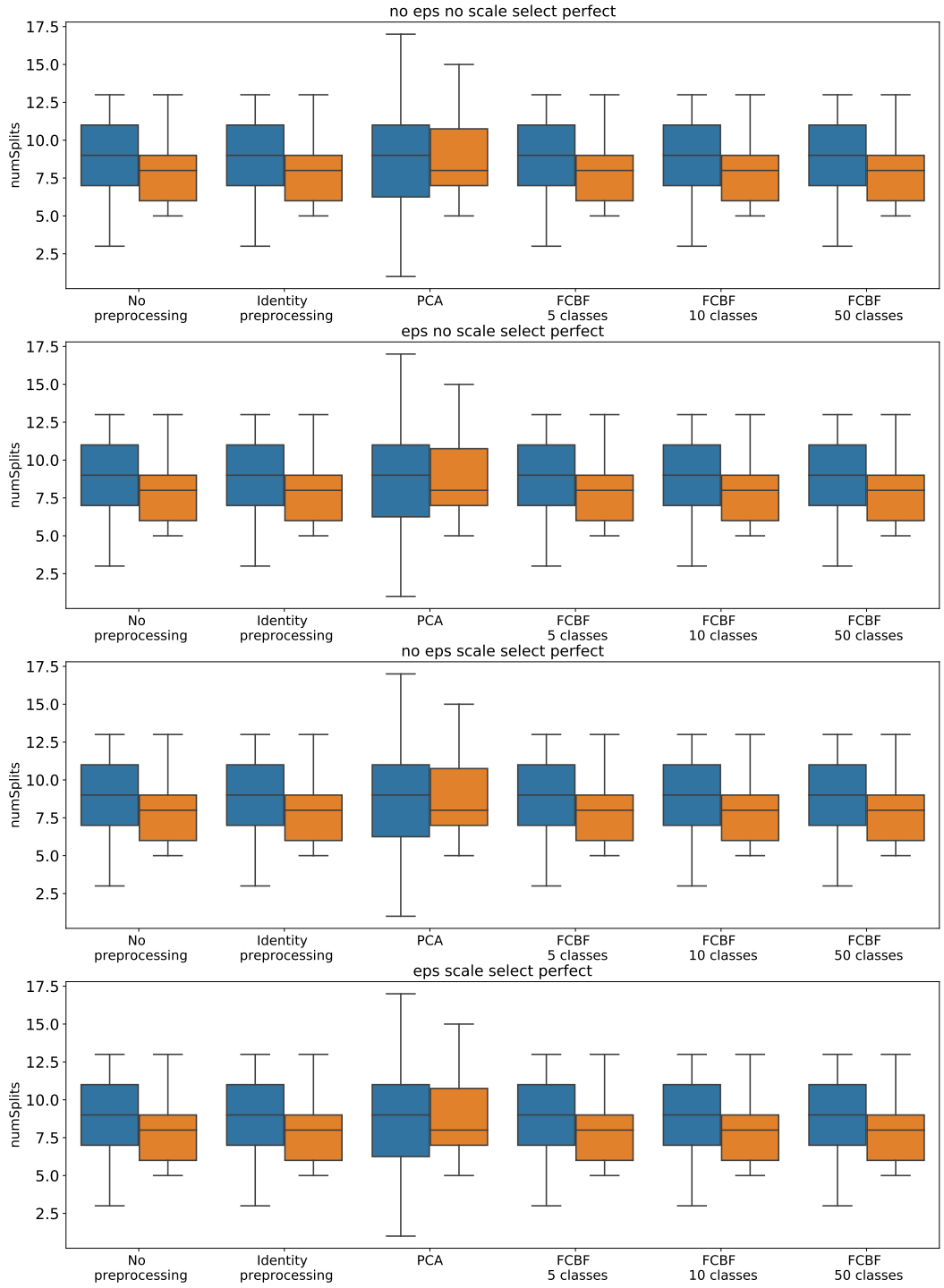

Figure 6.4: Number of splits for redundancy experiment with perfect manual feature selection

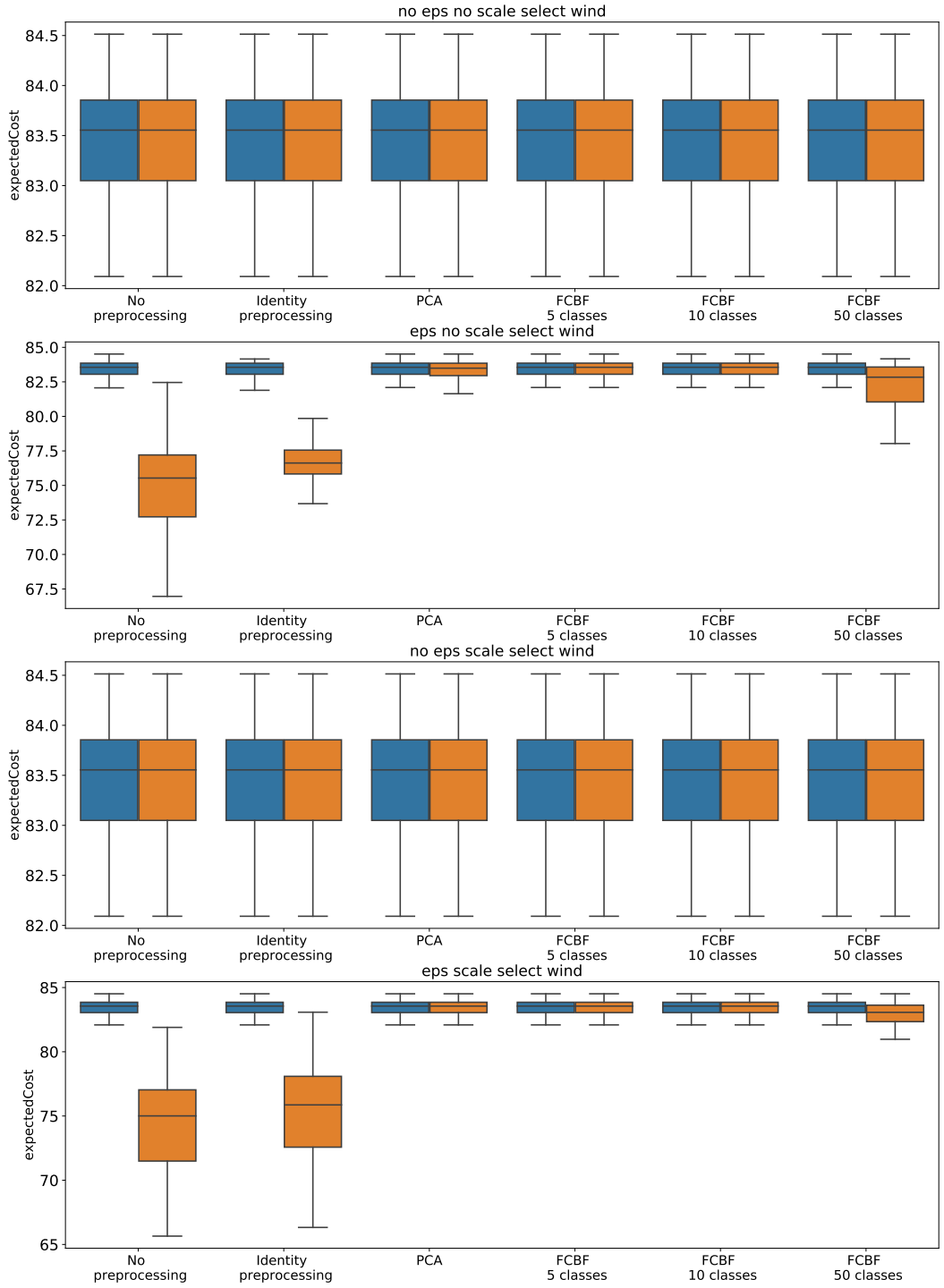

Figure 6.5: Expected Cost for redundancy experiment with manually selected wind features

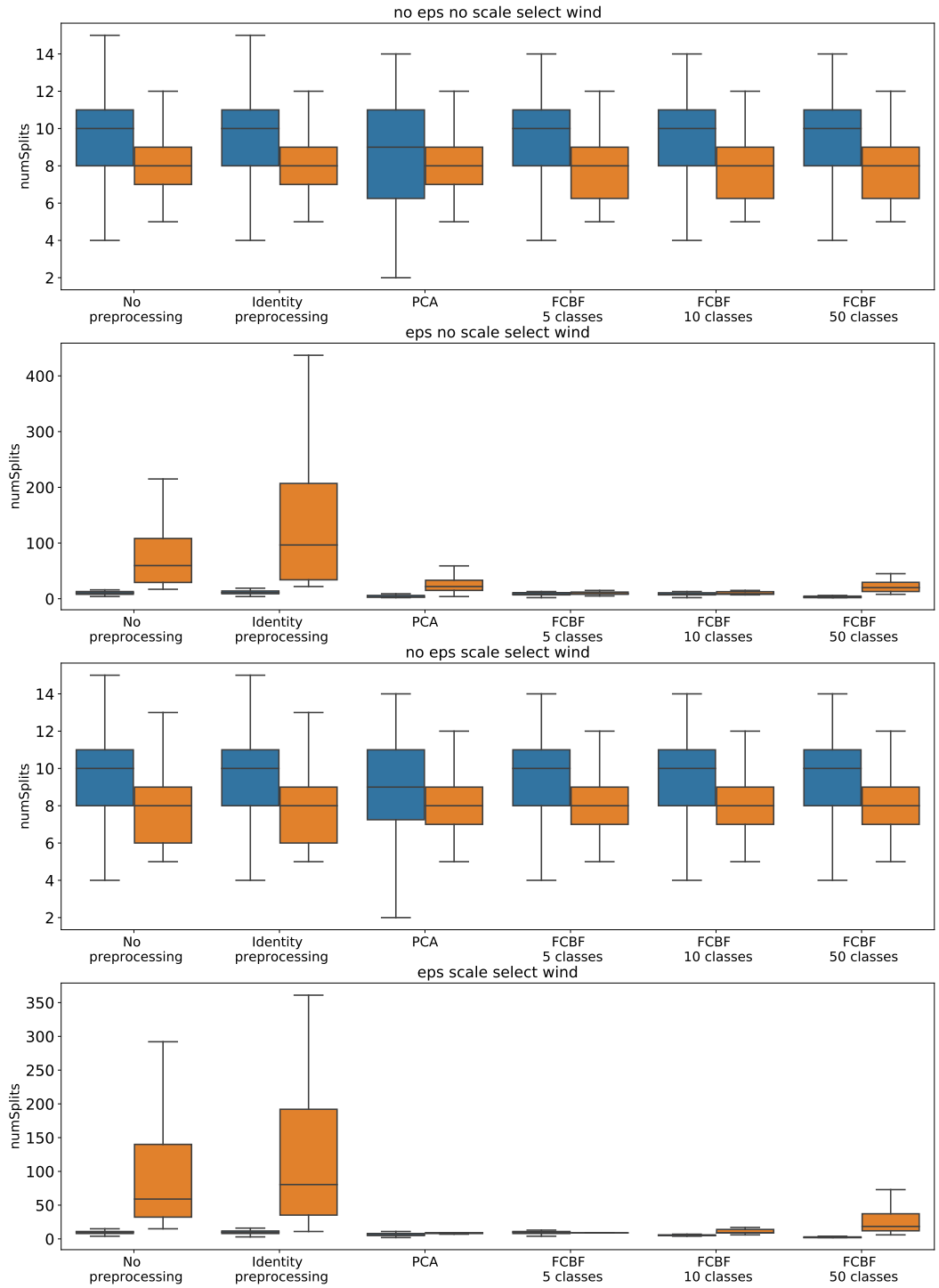

Figure 6.6: Number of splits for redundancy experiment with manually selected wind features

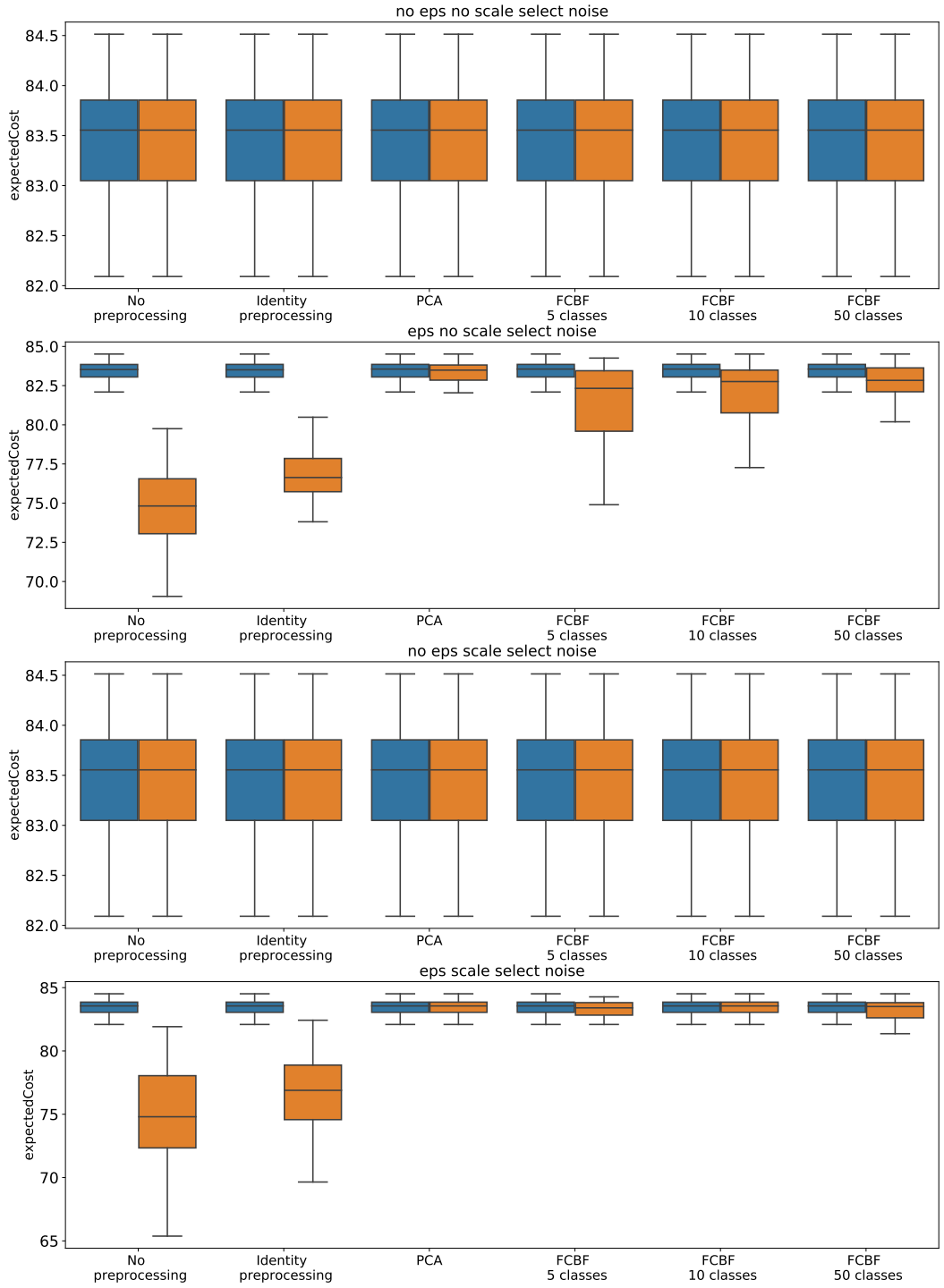

Figure 6.7: Expected Cost for redundancy experiment with manually selected anemometers

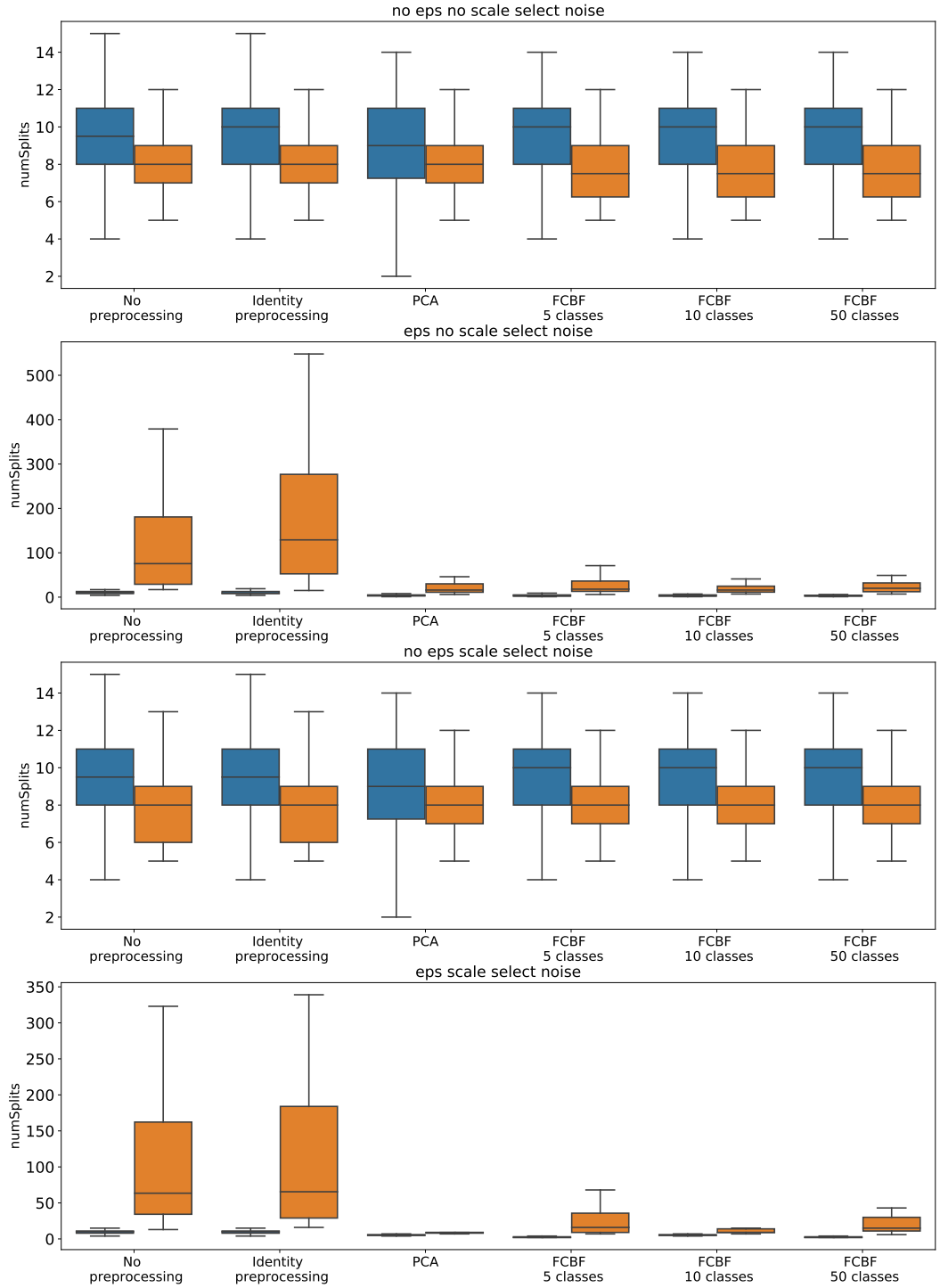

Figure 6.8: Number of splits for redundancy experiment with manually selected anemometers

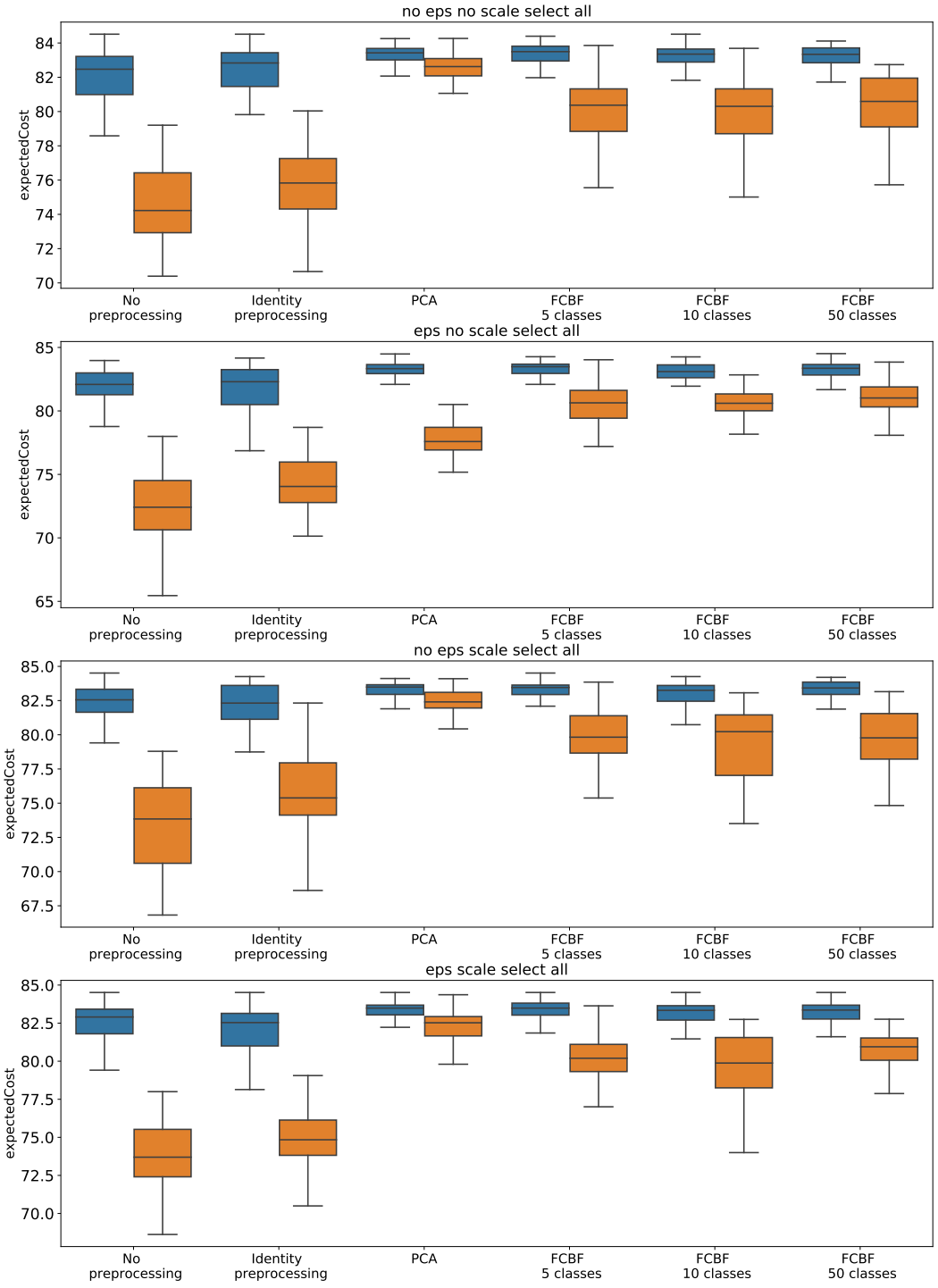

Figure 6.9: Expected Cost for redundancy experiment with all features manually selected

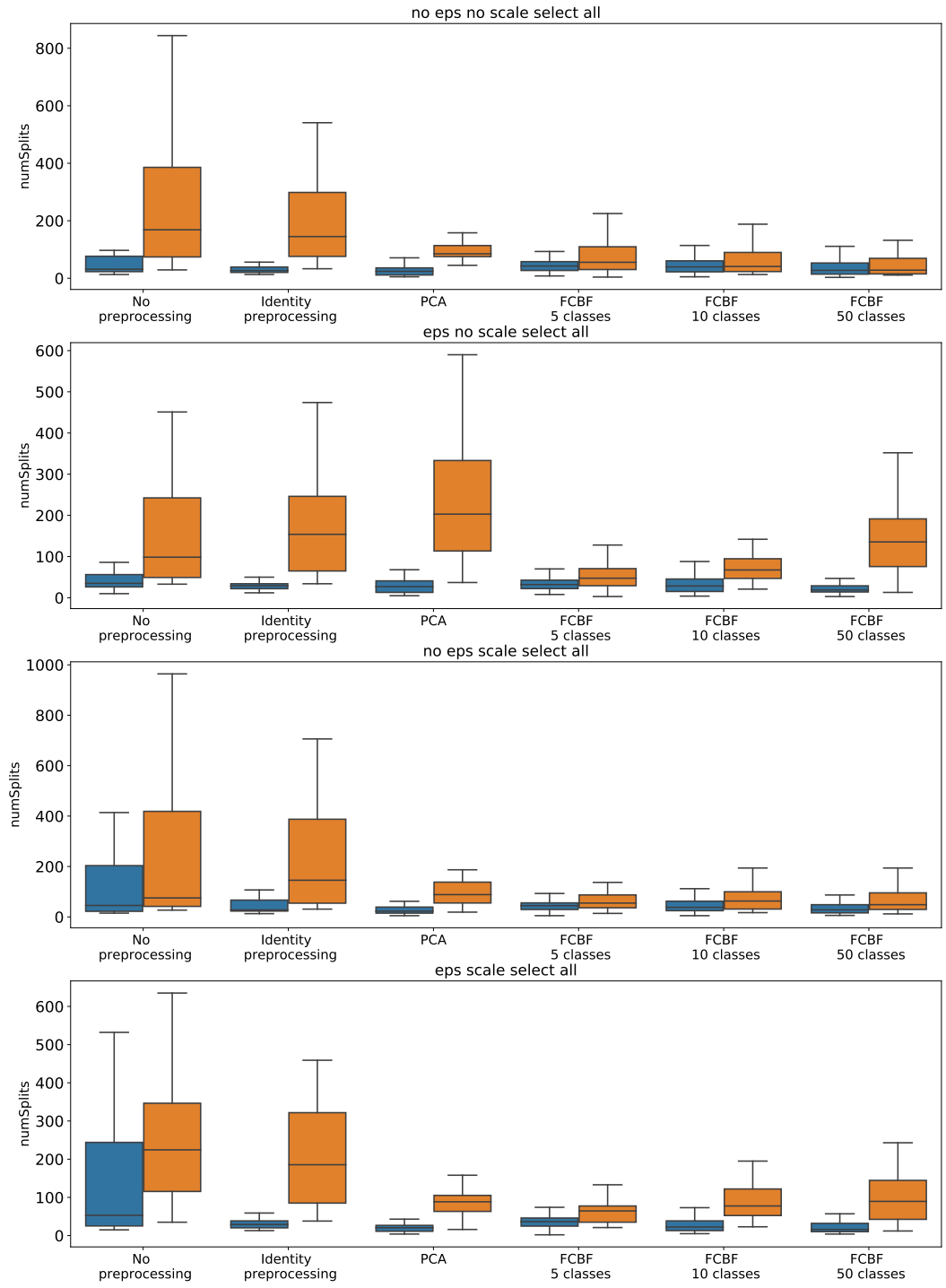

Figure 6.10: Number of splits for redundancy experiment with all features manually selected

6.4 Irrelevance Experiment

This experiment intents to show how the preprocessing algorithms handle varying degrees of irrelevant features. We will first introduce the model we are testing on, and then present the results along with commentary.

6.4.1 The Model

We will use three variations of the same model, for which we will test the quality of synthesized strategies, both with and without preprocessing. The variations differ in the number of variables that are available to the learning method. The model variants are based on the example given in Section 4.1, which is repeated in Figure 6.11. Recall that the optimal strategy considers different features for different sub-strategies. In L2, `windSpeed` is the only required feature for learning the optimal strategy, and `serviceTimer` is likewise the only one required for the optimal strategy in L1. The *`GenerateNoise`* function adds irrelevant features in the form of random values to the model. It is defined in Listing 6.4.

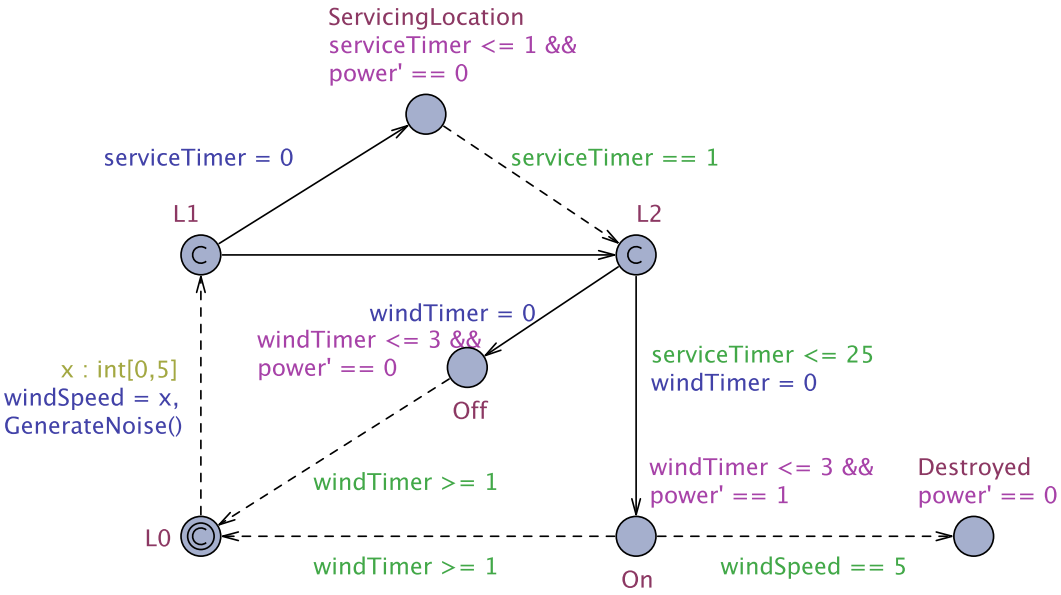

Figure 6.11: Wind turbine model with required periodic service, illustrating feature irrelevance. The variable declarations can be seen in Listing 6.3.

The 3 model variants are listed below along with their query. They are all trained under the UPPAAL TIGA strategy:

> strategy safe = control: A[] not(WindTurbine.Destroyed)

and tested with the same query as the redundancy experiment:

```
int windSpeed = 0;
clock windTimer, serviceTimer, time;
hybrid clock power;
const int num_noise_vars = 10;
double noise[num_noise_vars];
```

Listing 6.3: Variable declarations for the wind turbine with service.

```
void GenerateNoise() {
    int i = 0;
    for (i = 0; i < num_noise_vars; i++) {
        noise[i] = random(100);
    }
}
```

Listing 6.4: Implementation of GenerateNoise()

> E[<=100;100] (max:power) under opt

Select Smart In this variant, we select the best possible features that can be achieved using manual state transformation.

> strategy opt = maxE(power) [<=100] { WindTurbine.location } -> { windSpeed, serviceTimer } : <> time > 99 under safe

Select All We manually select all variables in the model, including the irrelevant features `time`, `power` and `windTimer`.

> strategy opt = maxE(power) [<=100] { WindTurbine.location } -> { windSpeed, serviceTimer, time, power, windTimer } : <> time > 99 under safe

Extra Irrelevance This is the only model variant that uses the *`GenerateNoise`* function to add irrelevant features. All the features of the previous variations are selected, in addition to the generated noise variables.

> strategy opt = maxE(power) [<=100] { WindTurbine.location } -> { windSpeed, serviceTimer, time, power, windTimer, noise[0], noise[1], ..., noise[9] } : <> time > 99 under safe

6.4.2 Evaluation of Results

The experiments has given results of both the expected cost and the number of splits in the strategies. We will first present the expected cost results, and then afterwards the number of splits results.

Expected Cost

Figure 6.12 shows the results using expected cost as a metric. A higher expected cost is better than a lower, with an expected cost of 80 being optimal.

Select Smart The results show no noticeable difference between no preprocessing and identity preprocessing. Both configurations have a large IQR, which indicates that there are multiple clusters of samples. Some samples in Q-learning reach the optimal expected cost, but the results are very unstable as can be seen in the ranges.

FCBF with 5 classes for Q-learning shows a lower median, than without preprocessing and a narrower IQR. FCBF with 10 classes also shows a low median for Q-learning, but the wider IQR indicates more scattered samples. FCBF with 50 classes improves the median compared to the other FCBF configurations. We can also observe a drop in the lower whisker. In all FCBF configurations, model-learning is close to no preprocessing and identity preprocessing.

Select All The median of identity preprocessing is slightly better than no preprocessing for both Q-learning and model-learning. Compared to select smart, the performance has degraded for these two configurations, as neither is able to reach optimal expected cost.

All FCBF configurations show a significant improvement relative to no preprocessing and identity preprocessing. Q-learning with FCBF is in this variant performing near-optimally, which is in stark contrast to the previous variant. FCBF also seems to be an improvement for model-learning, as every configuration where it was applied has a higher IQR, compared to no preprocessing.

Extra Irrelevance No preprocessing and identity preprocessing are both similar in this variant, with identity preprocessing showing a slightly better median in both Q-learning and model-learning. Compared to the select all variant, the performance of Q-learning has degraded for these two configurations.

The FCBF configurations show similar relative performance compared to the previous variant, with Q-learning performing better than model-learning. The median values for Q-learning with FCBF are however a little lower, than in the select all variant, and the range of model-learning and FCBF with 50 classes is smaller.

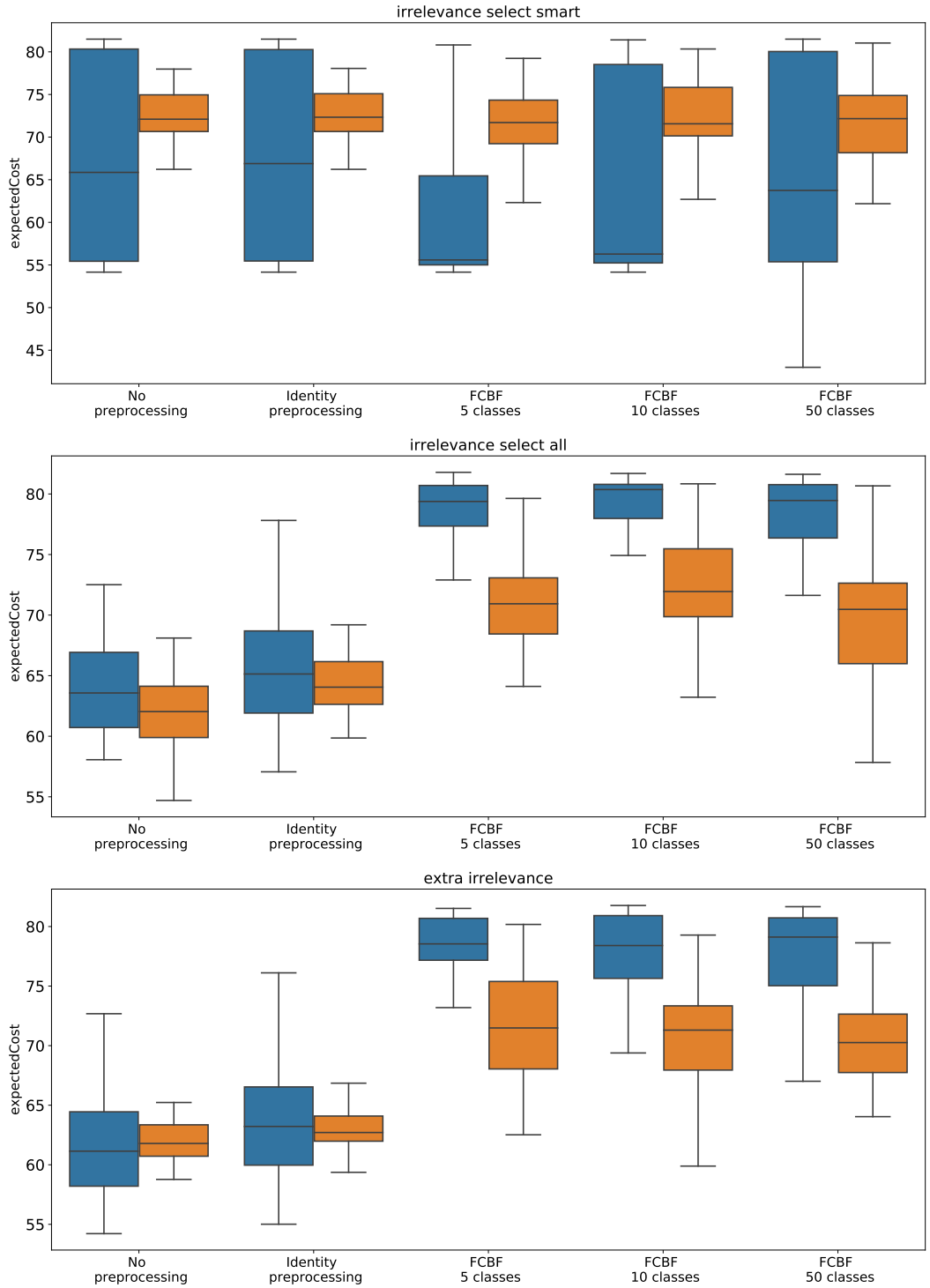

Figure 6.12: Experiment results for irrelevant features showing expected cost

Number of Splits

The results using number of splits as a metric are shown in Figure 6.13. A lower number of splits is better than a higher.

Select Smart All configurations show similar amounts of splits, with Q-learning producing fewer splits than model-learning.

Select All In this variation, the no preprocessing and identity preprocessing show more splits compared to the previous variant. Q-learning still produces strategies with fewer splits than model-learning.

The FCBF configurations show a significant reduction in the number of splits compared to no preprocessing and identity preprocessing. Q-learning is also producing fewer splits than model-learning. There seems to be a significant correlation between the number of splits and the expected cost, with an increase in splits, decreasing the expected cost.

Extra Irrelevance Here we see that the range of splits with no preprocessing and identity preprocessing has increased to more than 2000 splits, up from 800 in the select all variant. No preprocessing and identity preprocessing show similar performance in this variant, and worse performance relative to the previous variant.

All FCBF configurations show significantly fewer splits produced than no preprocessing and identity preprocessing.

The correlation between a lower amount of splits, and a higher expected cost, is still present in this variation.

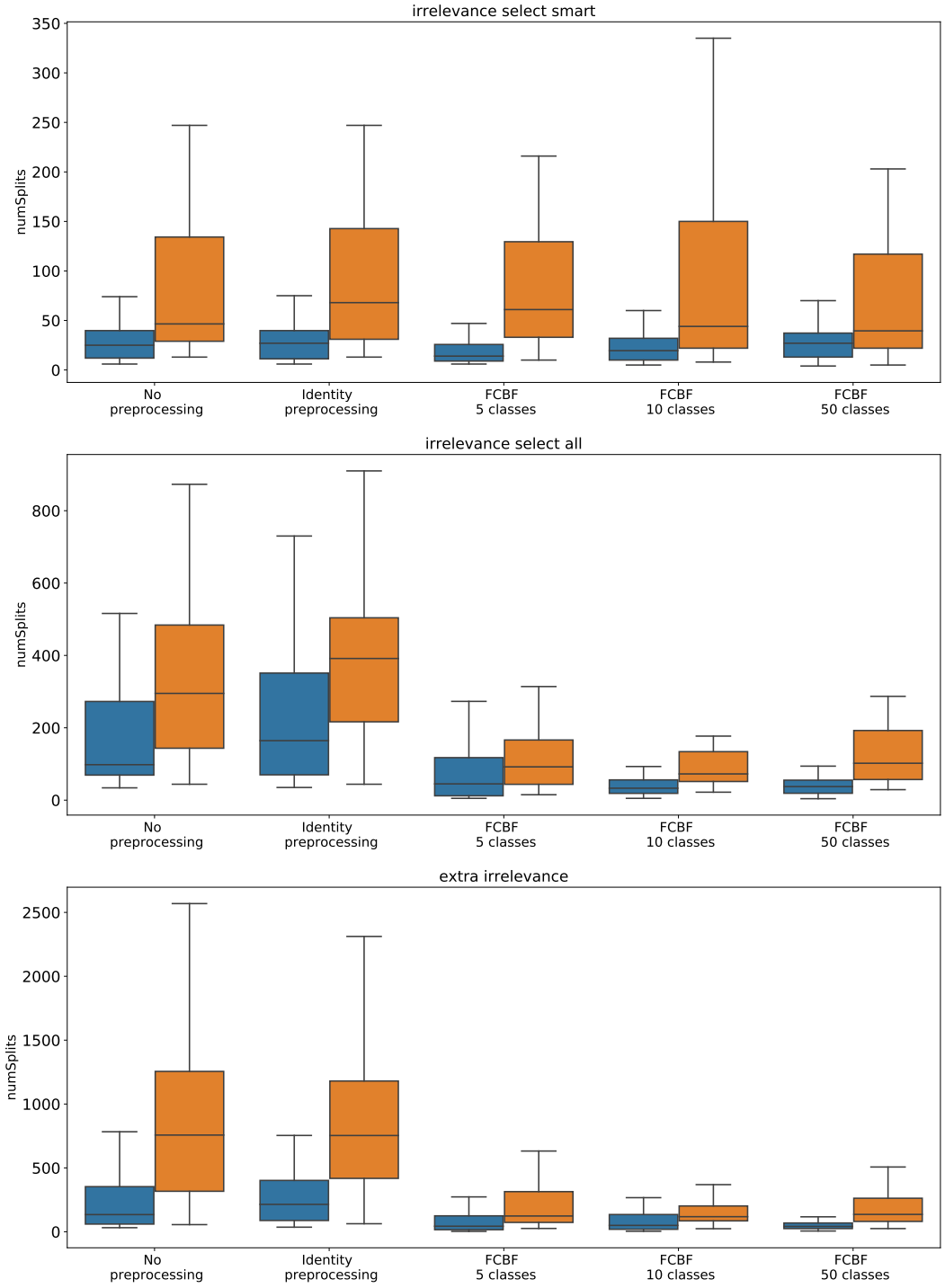

Figure 6.13: Experiment results for irrelevant features showing number of splits

7 Discussion

In this chapter we will review the results of the experiments, and discuss the implication that they may bring with them.

We will first present the results themselves, and give our interpretations of what they reveal. Then we will relate this to the problem statement of this thesis, and consider which parts are answered by the results and which parts still remain unaccounted for. Afterwards we discuss a few remaining concerns regarding preprocessing and potential alternative benefits.

7.1 Results of the Experiments

We analyze the results in two parts. First we give an overall perspective on the general observations, followed by a more detailed look at specific interesting parts of the results.

Overall Results The primary observation that should be made regarding the redundancy experiment is that, across all variations, the configurations that applied preprocessing either improved the expected cost or kept them at an equal level, compared with the configurations where no preprocessing was applied. This is also largely true for the amount of splits used to represent the strategies, with the single exception being model-learning with PCA, in the second variation of the select all redundancy experiment with noise and no scaling, shown in Figure 6.10.

A similar observation can be made for the irrelevance experiment, with the significant exception being the select smart variation in Figure 6.12, where the FCBF configurations all had lower medians than without preprocessing. We were curious as to why this is the case, and also noted that the IQR of all the Q-learning configurations are suspiciously wide. This prompted us to visualize the data with a violin-plot [25], which is similar to a box plot, but visualizes the distribution of observations across the range. This plot can be seen in Figure 7.1. The violin plot shows the box plot as the black bar inside of the colored shapes. The white dot is the median, the thick black bar is the IQR and the thin black bars are the whiskers. The shapes are formed by density estimations over the observed expected costs. This gives us a better understanding of the distribution of the results across the range.

Here we see that the results for Q-learning either reaches a near-optimal expected cost or gets a rather low expected cost around $50-60$ across all configurations. The tendency in the data to split in two groups, explains the wide IQR in the box plot. We also see that the increased width in the range of FCBF with 50 classes, is due to a single observation that falls below the rest, which we consider to be an outlier. For the FCBF configuration with 5 classes, there is a bigger portion of strategies in the lower group. Even so, it is still visible that the groupings of the expected costs are similar for Q-learning across all

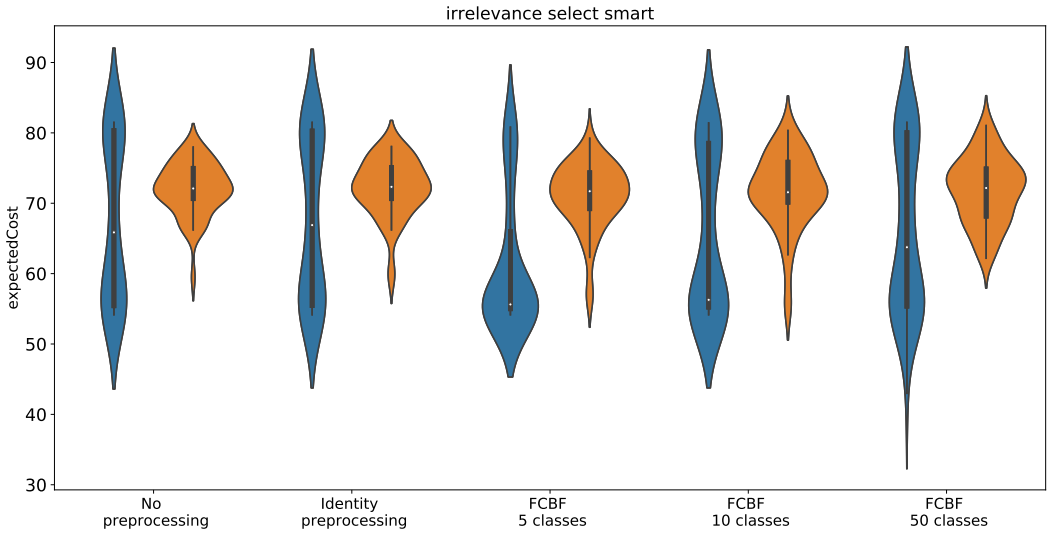

Figure 7.1: Experiment results as a violin-plot for irrelevant features showing expected cost

configurations. This result is more in line with our expectation, that the FCBF algorithm does not degrade the expected cost, when there are no irrelevant features to remove. However, we can not disregard the fact that FCBF does worsen the performance, in a variation where we had expected FCBF to improve the results by removing locally irrelevant features. This calls for further inquiry as to why this is the case.

Variation Specific Observations If we look closer at the variations in the redundancy experiment, we see in Figures 6.3 and 6.4 that the select perfect variations are very consistent. This is exactly as expected, as the *windSpeed* variable has been manually selected, leaving no doubt as to which features the learning methods should consider. This was also expected for the no epsilon variations of the select wind and select noise queries from Figures 6.5 and 6.7, as any of the available features can be used to find the optimal strategy. If we then consider the variations where epsilon is used to introduce noise to the wind readings, we see a clear correlation between lower expected cost and a high number of splits. In fact, if we look across all the results in both experiments we see this correlation in many variations for model-learning without preprocessing. This is evidence for a tendency to overfit the strategy to the noise when using model-learning, which seems to be resolved with the applied preprocessing techniques.

Another interesting thing to note for the select noise variations with epsilon, is that when scaling is not included, only PCA enables model-learning to reach consistent optimal strategies, with 50 classes FCBF coming close. This is expected as PCA, being a feature extraction method, can extract a common trend among the noise variables, while FCBF can only hope to select the least noisy variables. However, when scaling is introduced, FCBF unexpectedly manages to equal the expected cost of the PCA configuration.

This could be due to the way we discretize the features and labels, however further investigation is required to determine if this is the case.

In regards to Q-learning in the redundancy experiment, almost all variations and configurations manage to reach near-optimal behavior, with only the select all query causing a slight reduction in expected cost for no preprocessing and identity preprocessing. This suggests that Q-learning has little to no issue with collinearity. Another possible explanation could be that Q-learning finds an easy solution to this particular model, which is to turn on the wind turbine whenever the "safe" strategy from UPPAAL TIGA allows it.

The last observation we wish to make, is in regard to the model-learning with PCA configuration in the second variation of the select all query, shown in Figures 6.9 and 6.10. In all other configurations where PCA is applied, the results are better or equal to those of the other configurations. However, in this particular case, PCA dips below FCBF in expected cost, and has a significant increase in the number of splits. The cause for this is uncertain, and would be interesting to further pursue.

7.2 Relating the Results to the Problem Statement

In Chapter 3 we stated the problem of the thesis to be:

> How can preprocessing be used to alleviate model issues and thereby improve the synthesis of strategies for Priced Timed Markov Decision Processes by UPPAAL STRATEGO?

We then delimited the thesis from improvements in terms of run time and memory required for the synthesis, in order to focus on the quality of the strategies that UPPAAL STRATEGO produces.

The results of the experiments show, that the Q-learning and model-learning techniques in UPPAAL STRATEGO are not currently able to resolve both of the presented data issues. They also show that preprocessing with PCA and FCBF may enable UPPAAL STRATEGO to produce strategies that have a higher expected cost and a lower number of splits, for models with collinear and irrelevant features.

It is not certain whether the demonstrated benefits of preprocessing will generalize to all models with these data issues. Neither has it been shown whether the two preprocessing methods can be applied on models that does not exhibit the data issues, without an adverse effect on the synthesis.

7.3 Remaining Concerns and Alternative Benefits

While the results provide us with a good indication of the effects that preprocessing can have on strategy synthesis, there are still concerns that need to be addressed.

First of, the learning methods are still undergoing development, and the results of the experiments may be the product of errors in the synthesis, that have yet to be discovered and resolved. It is also possible that with further development, the learning methods

might be able to resolve the given data issues directly, thereby making preprocessing of them irrelevant.

An interesting effect of preprocessing, is that it might encourage creators of models to worry less about introducing data issues in their models, if they trust that the synthesis can automatically remove them. This could allow them to focus more on producing an accurate model of the system they are working with.

8 Conclusion

This section will conclude the thesis by first summarizing the problem and our solution to it. We will then outline how we reached the solution and finally the implications of this thesis.

As previously mentioned, UPPAAL STRATEGO is a tool that synthesizes strategies for UPPAAL models. These models can, however, easily contain superfluous features. We show that these features can decrease the quality of the synthesized strategies. Based on this observation, we present the following problem statement:

> How can preprocessing be used to alleviate model issues and thereby improve the synthesis of strategies for Priced Timed Markov Decision Processes by UPPAAL STRATEGO?

The contribution of this thesis is the demonstration of how existing preprocessing techniques, FCBF and PCA, can be used to combat the data issues of feature irrelevance and redundancy in UPPAAL models.

Experiments with the application of preprocessing show that, on specific models, the synthesis of strategies can improve the resulting strategy performance and size. When the models do not exhibit the data issues, experiments do not show degradation of strategy performance and size with the application of preprocessing.

We start by presenting the problem domain of this thesis, UPPAAL STRATEGO and the underlying automata theory, which we base our work on. After defining the above problem, we go on to identify and describe feature irrelevance and redundancy as two potential data issues in UPPAAL models. These issues guide us towards choosing PCA and FCBF as preprocessing techniques. We then present the results of experiments, demonstrating the applicability of our approach.

Although we have yet to evaluate our approach on other UPPAAL models, we believe this preliminary work shows promising results. We expect further research in this topic can yield even better results. We hope that our thesis will provide a basis for further research on the topic, and that it will result in an improvement to UPPAAL STRATEGO.

9 Future Work

In this last chapter we will outline ideas that would be interesting to pursue in future projects.

Alternative Discretization in FCBF In Section 5.4.2 we mention that in order to apply FCBF in UPPAAL STRATEGO, we need to discretize the continuous labels and features, as FCBF is only defined for classification problems. The discretization algorithm we use is very simple, which may cause subpar performance of FCBF. A continuous version of FCBF has been proposed [26], which applies a different discretization technique [27]. Future work should investigate what kind of other discretization techniques are suitable.

Testing on Real-world Models We have in this thesis only evaluated our solution on two UPPAAL models. It would be interesting to evaluate the preprocessing algorithms on additional models, preferably ones that are not explicitly constructed to showcase data issues.

Other Data Issues The data issues that are considered in this thesis, were chosen as the focus points, due to their presumed prevalence in UPPAAL models, but there are plenty other issues of interest, that could be interesting to evaluate. Examples could be multicollinearity or feature interaction.

Multicollinearity [28] is a general form of collinearity, where a feature is linearly dependent on a combination of other features, such that:

$$c_0 + c_1 f_1 + c_2 f_2 + ... + c_n f_n + \epsilon = 0$$

Where c_i is a constant, f_i is a feature, and ϵ is a small amount of noise. PCA is also capable of reducing this kind of redundancy, so it would be interesting to test if the current solution is sufficient.

Feature interaction [29][1] is the idea that features, which individually provide little information, may provide a lot of information when together. The prime example is when trying to learn the Boolean XOR operation $XOR(A, B) = C$. Neither A nor B have any correlation with C, but together they enable a perfect prediction. This can sometimes be an issue for feature selection methods such as FCBF, as they might discard useful features based on their direct interaction with the label.

Combination of Preprocessing Methods The experiments presented in Chapter 6 have only tested each preprocessing method in isolation. It would be interesting to see how they perform, combined. For example, PCA could be used to find the trend-line of a noisy variable, and FCBF could then remove irrelevant features.

[1] Feature interaction is also known as attribute interaction

Reduce Strategy Re-training In Learning Algorithm We currently re-train a strategy on each iteration of the learning algorithm, as described in Section 5.4.3. This is done as the dimensionality and meaning of the transformed data can change across iterations. A better, more efficient method could detect when the dimensionality and meaning of features has not changed significantly, such that the existing strategy can be improved, instead of a new strategy being trained anew.

Efficient Implementation The current implementation of preprocessing is implemented in Python, and is not designed to be efficient. Future work could re-implement the preprocessing algorithms in C++, such that they can efficiently integrate with the existing UPPAAL STRATEGO code.

Bibliography

[1] David, A., Jensen, P. G., Larsen, K. G., Mikučionis, M., and Taankvist, J. H., "Uppaal stratego", in *Tools and Algorithms for the Construction and Analysis of Systems*, Baier, C. and Tinelli, C., Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 206–211, ISBN: 978-3-662-46681-0.

[2] Behrmann, G., David, A., and Larsen, K. G., "A tutorial on UPPAAL", in *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, Bernardo, M. and Corradini, F., Eds., ser. LNCS, Citation is to the updated version from November 28, 2006, Springer–Verlag, Nov. 2004, pp. 200–236. [Online]. Available: http://www.it.uu.se/research/group/darts/uppaal/documentation.shtml.

[3] David, A., Jensen, P. G., Larsen, K. G., Legay, A., Lime, D., Sørensen, M. G., and Taankvist, J. H., "On time with minimal expected cost!", in *Automated Technology for Verification and Analysis*, Cassez, F. and Raskin, J.-F., Eds., Cham: Springer International Publishing, 2014, pp. 129–145, ISBN: 978-3-319-11936-6.

[4] Cassez, F., David, A., Fleury, E., Larsen, K. G., and Lime, D., "Efficient on-the-fly algorithms for the analysis of timed games", in *CONCUR 2005 – Concurrency Theory*, Abadi, M. and Alfaro, L. de, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 66–80, ISBN: 978-3-540-31934-4.

[5] Brihaye, T., Bruyère, V., and Raskin, J.-F., "On optimal timed strategies", in *Formal Modeling and Analysis of Timed Systems*, Pettersson, P. and Yi, W., Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 49–64, ISBN: 978-3-540-31616-9.

[6] Jenkins, N., Burton, A., Sharpe, D., and Bossanyi, E., *Wind Energy Handbook*. Wiley, 2001, ISBN: 0-4714-8997-2.

[7] Watkins, C. J. C. H. and Dayan, P., "Q-learning", *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992, ISSN: 1573-0565. DOI: 10.1007/BF00992698.

[8] Strehl, A. L. and Littman, M. L., "An analysis of model-based interval estimation for markov decision processes", *Journal of Computer and System Sciences*, vol. 74, no. 8, pp. 1309–1331, 2008, Learning Theory 2005, ISSN: 0022-0000. DOI: https://doi.org/10.1016/j.jcss.2007.08.009.

[9] Ladha, L. and Deepa, T., "Feature selection methods and algorithms", in *International Journal on Computer Science and Engineering (IJCSE)*, 2011.

[10] John, G. H., Kohavi, R., and Pfleger, K., "Irrelevant features and the subset selection problem", in *Proceedings of the Eleventh International Conference on International Conference on Machine Learning*, ser. ICML'94, New Brunswick, NJ, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 121–129, ISBN: 1-55860-335-2.

[11] Bishop, C. M., *Pattern Recognition and Machine Learning*. New York: Springer-Verlag New York, 2006, ch. 12, ISBN: 978-0-387-31073-2.

[12] In, *Data Mining (Third Edition)*, ser. The Morgan Kaufmann Series in Data Management Systems, Han, J., Kamber, M., and Pei, J., Eds., Third Edition, Boston: Morgan Kaufmann, 2012, ISBN: 978-0-12-381479-1.

[13] Li, J., Cheng, K., Wang, S., Morstatter, F., Trevino, R., Tang, J., and Liu, H., "Feature selection: A data perspective", English (US), *ACM Computing Surveys*, vol. 50, no. 6, Dec. 2017, ISSN: 0360-0300. DOI: 10.1145/3136625.

[14] Guyon, I. and Elisseeff, A., "An introduction to feature extraction", in *Feature Extraction: Foundations and Applications*, Guyon, I., Nikravesh, M., Gunn, S., and Zadeh, L. A., Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–25, ISBN: 978-3-540-35488-8. DOI: 10.1007/978-3-540-35488-8_1.

[15] Rumelhart, D. E., Hinton, G. E., and Williams, R. J., "Learning internal representations by error propagation", in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, Rumelhart, D. E. and Mcclelland, J. L., Eds., Cambridge, MA: MIT Press, 1986, pp. 318–362.

[16] Murao, H. and Kitamura, S., "Q-learning with adaptive state segmentation (qlass)", in *Computational Intelligence in Robotics and Automation, 1997. CIRA'97., Proceedings., 1997 IEEE International Symposium on*, Jul. 1997, pp. 179–184. DOI: 10.1109/CIRA.1997.613856.

[17] Goldberger, J., Hinton, G. E., Roweis, S. T., and Salakhutdinov, R. R., "Neighbourhood components analysis", in *Advances in Neural Information Processing Systems 17*, Saul, L. K., Weiss, Y., and Bottou, L., Eds., MIT Press, 2005, pp. 513–520.

[18] Sprague, N., "Basis iteration for reward based dimensionality reduction", in *2007 IEEE 6th International Conference on Development and Learning*, Jul. 2007, pp. 187–192. DOI: 10.1109/DEVLRN.2007.4354032.

[19] Collins, M., Dasgupta, S., and Schapire, R. E., "A generalization of principal component analysis to the exponential family", in *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, ser. NIPS'01, Vancouver, British Columbia, Canada: MIT Press, 2001, pp. 617–624.

[20] Roy, N. and Gordon, G., "Exponential family pca for belief compression in pomdps", in *Proceedings of the 15th International Conference on Neural Information Processing Systems*, ser. NIPS'02, Cambridge, MA, USA: MIT Press, 2002, pp. 1667–1674.

[21] Jolliffe, I., *Principal Component Analysis*, ser. Springer Series in Statistics. Springer, 2002, ISBN: 9780387954424.

[22] Scikit-learn. (May 2018). Decomposing signals in components (matrix factorization problems), [Online]. Available: http://scikit-learn.org/stable/modules/decomposition.html.

[23] Yu, L. and Liu, H., "Feature selection for high-dimensional data: A fast correlation-based filter solution", English (US), in *Proceedings, Twentieth International Conference on Machine Learning*, Fawcett, T. and Mishra, N., Eds., vol. 2, 2003, pp. 856–863, ISBN: 1577351894.

[24] McGill, R., Tukey, J. W., and Larsen, W. A., "Variations of box plots", *The American Statistician*, vol. 32, no. 1, pp. 12–16, 1978, ISSN: 00031305.

[25] Hintze, J. L. and Nelson, R. D., "Violin plots: A box plot-density trace synergism", *The American Statistician*, vol. 52, no. 2, pp. 181–184, 1998. DOI: 10.1080/00031305.1998.10480559.

[26] Kannan, S. S. and Ramaraj, N., "An improved correlation-based algorithm with discretization for attribute reduction in data clustering", *Data Science Journal*, vol. 8, pp. 125–138, 2009. DOI: 10.2481/dsj.007-044.

[27] Tsai, C.-J., Lee, C.-I., and Yang, W.-P., "A discretization algorithm based on class-attribute contingency coefficient", *Inf. Sci.*, vol. 178, no. 3, pp. 714–731, Feb. 2008, ISSN: 0020-0255. DOI: 10.1016/j.ins.2007.09.004.

[28] Chatterjee, S. and Hadi, A. S., *Regression Analysis by Example*, 4th ed. John Wiley & Sons, Inc., Hoboken, New Jersey, 2016, ch. 9: Analysis of Collinear Data, ISBN: 978-0-470-05545-8.

[29] Jakulin, A. and Bratko, I., "Analyzing attribute dependencies", in *Knowledge Discovery in Databases: PKDD 2003*, Lavrač, N., Gamberger, D., Todorovski, L., and Blockeel, H., Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 229–240, ISBN: 978-3-540-39804-2.

A Experiment Figures

This appendix holds the remaining results from the experiments described in Chapter 6, i.e. the ones with 25, and 100 runs. Apart from the differences in runs, the setup for all three experiments are identical.

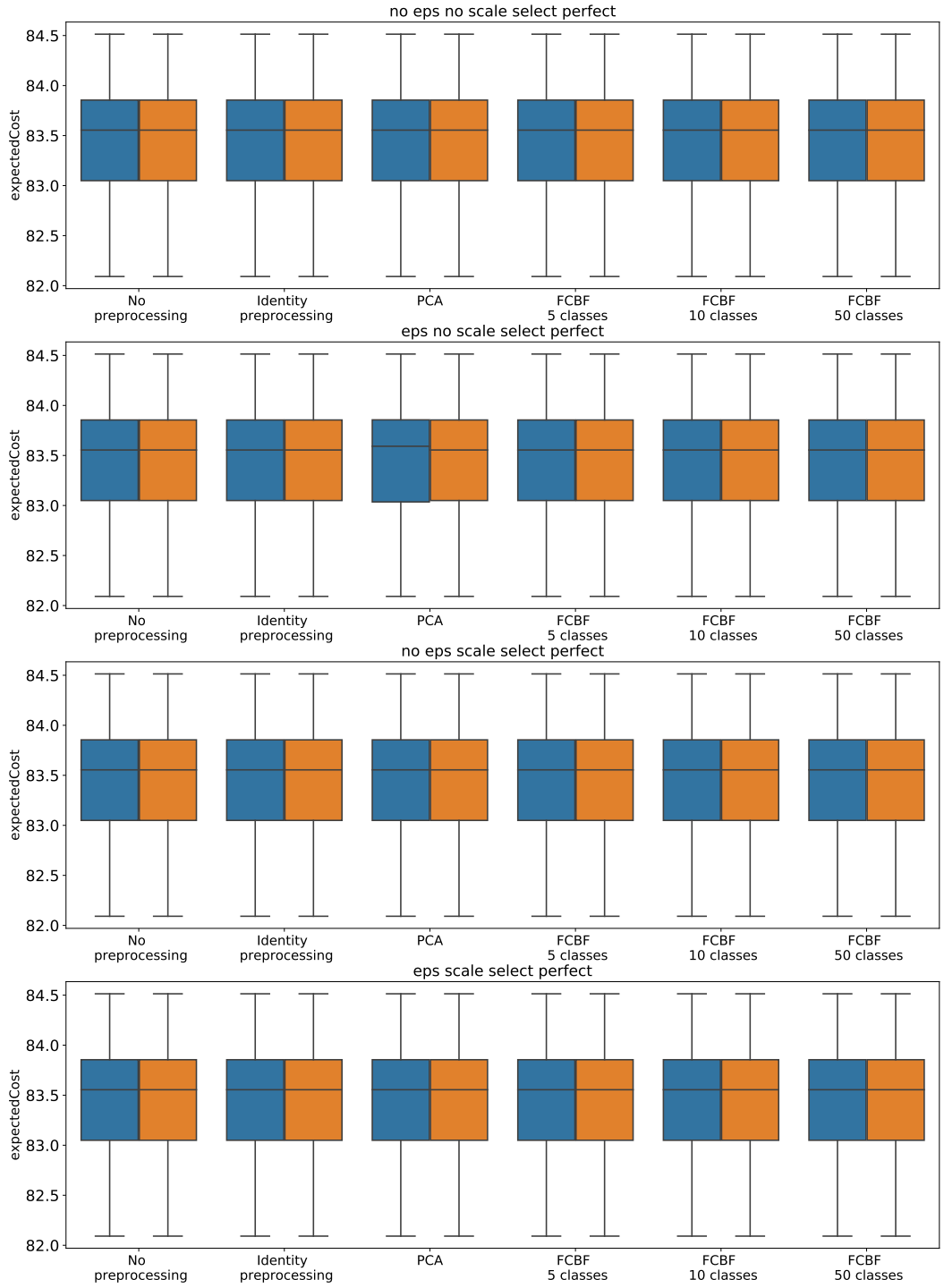

Figure A.1: Expected Cost for redundancy experiment with manually selected perfect features and runs set to 100

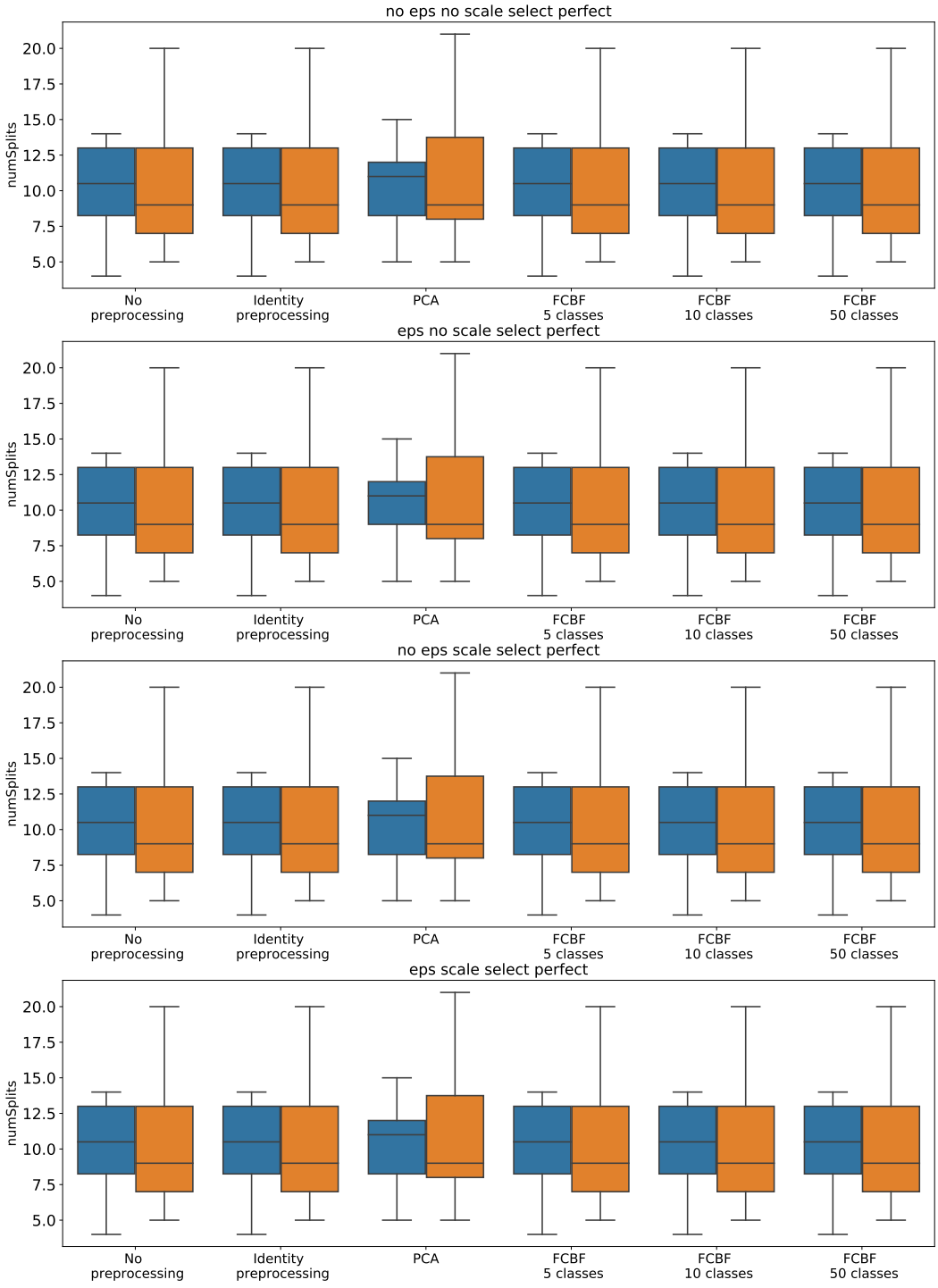

Figure A.2: Number of splits for redundancy experiment with manually selected perfect features and runs set to 100

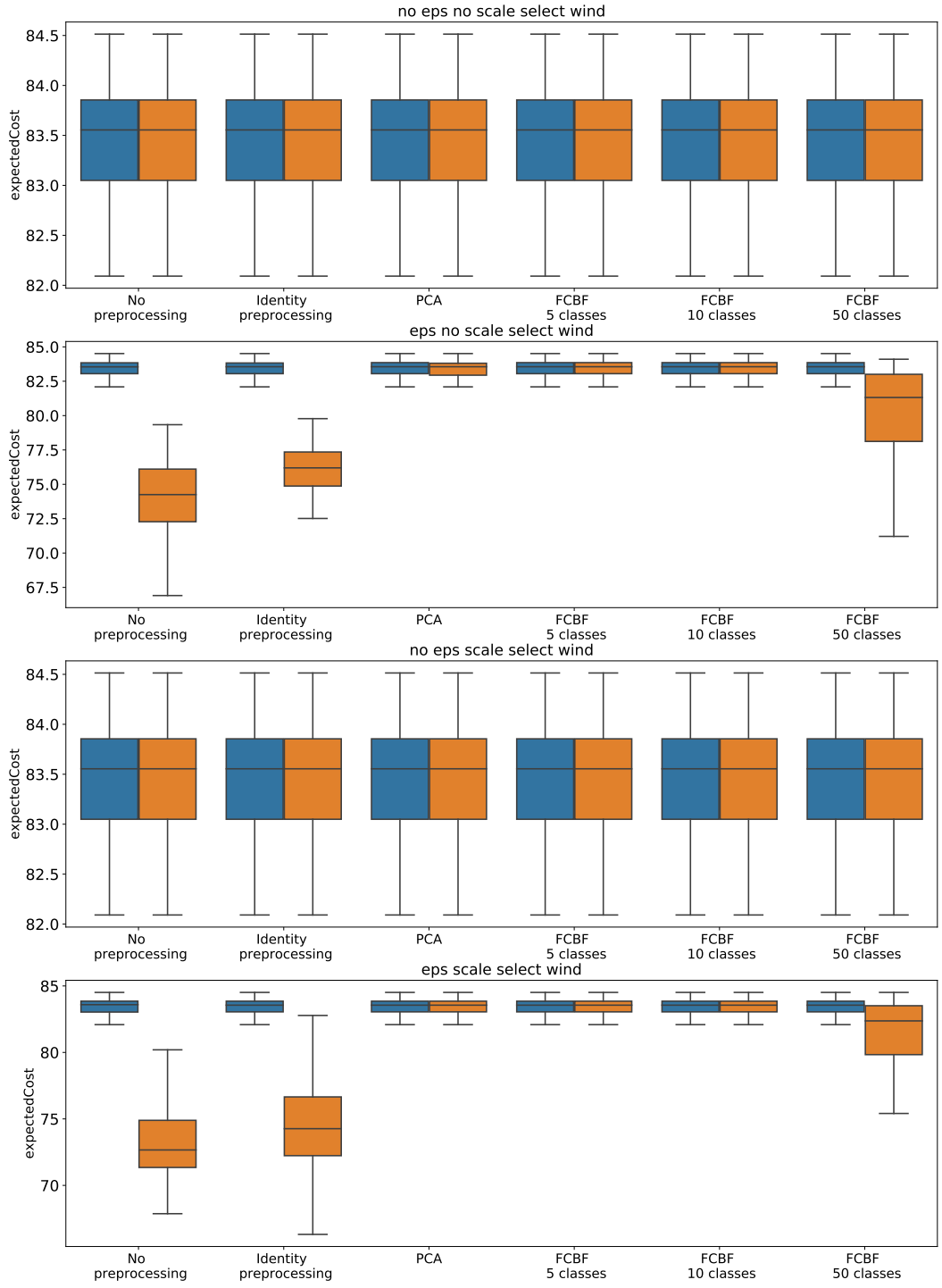

Figure A.3: Expected Cost for redundancy experiment with manually selected wind features and runs set to 100

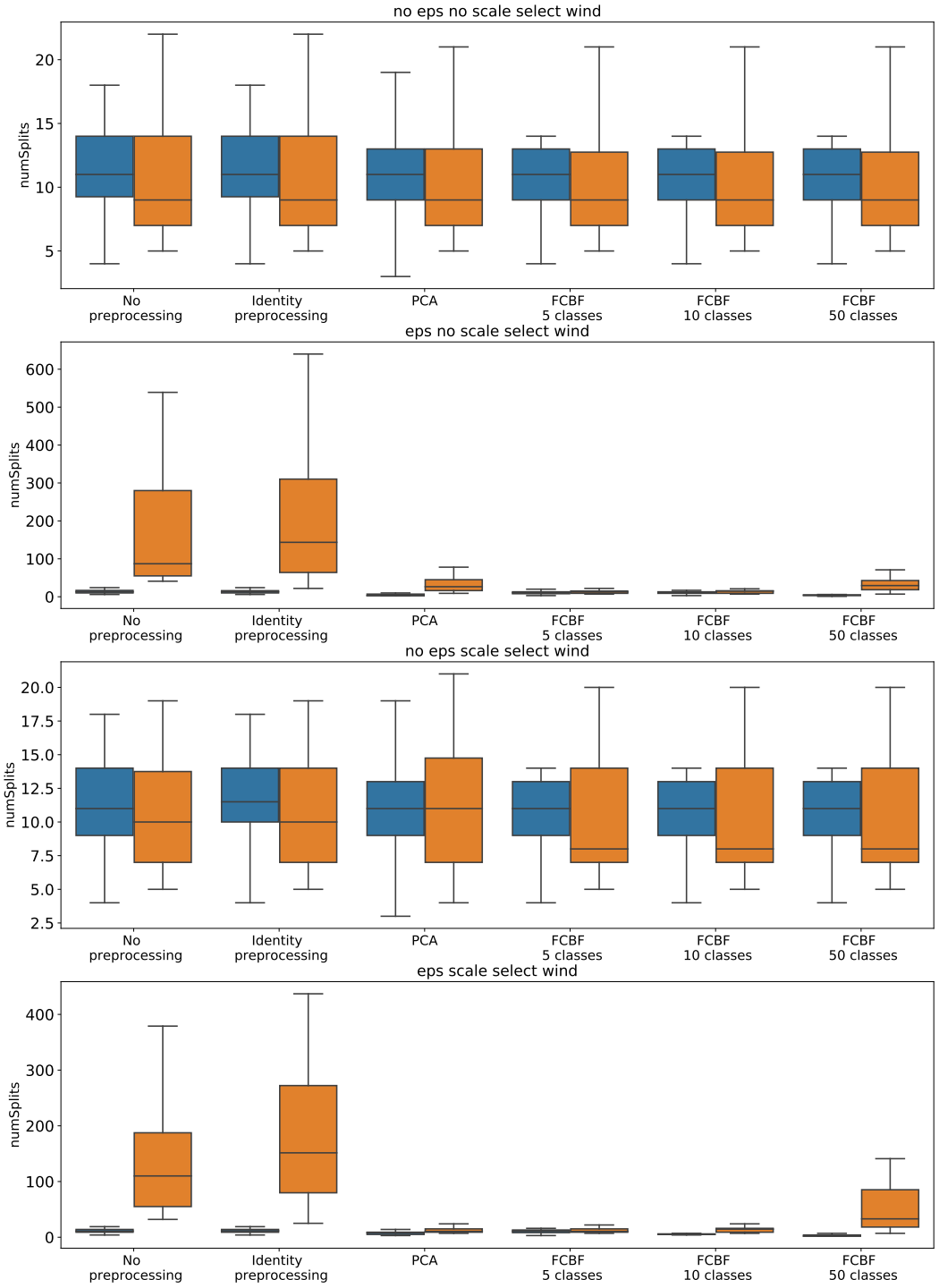

Figure A.4: Number of splits for redundancy experiment with manually selected wind features and runs set to 100

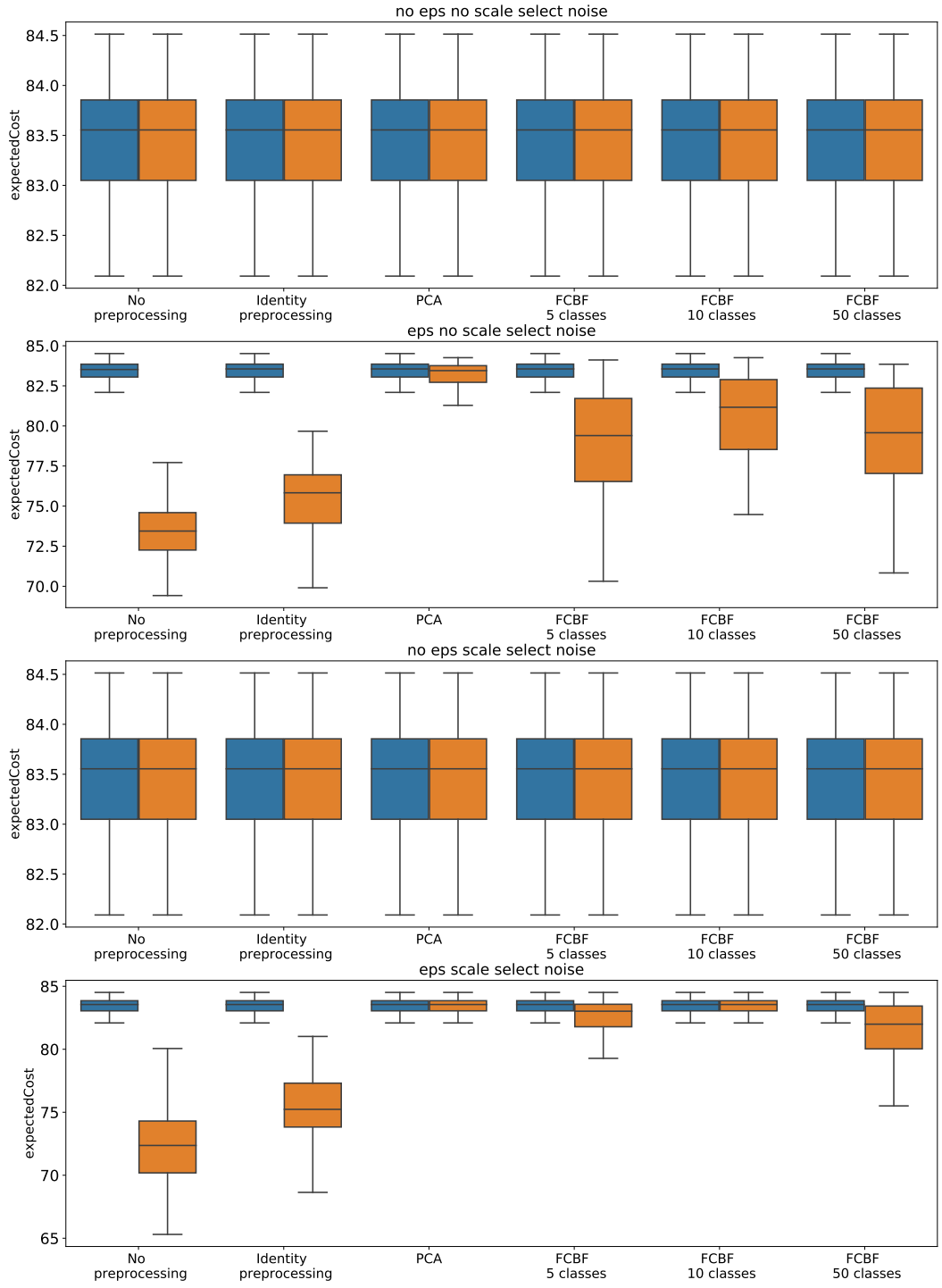

Figure A.5: Expected Cost for redundancy experiment with manually selected noise features and runs set to 100

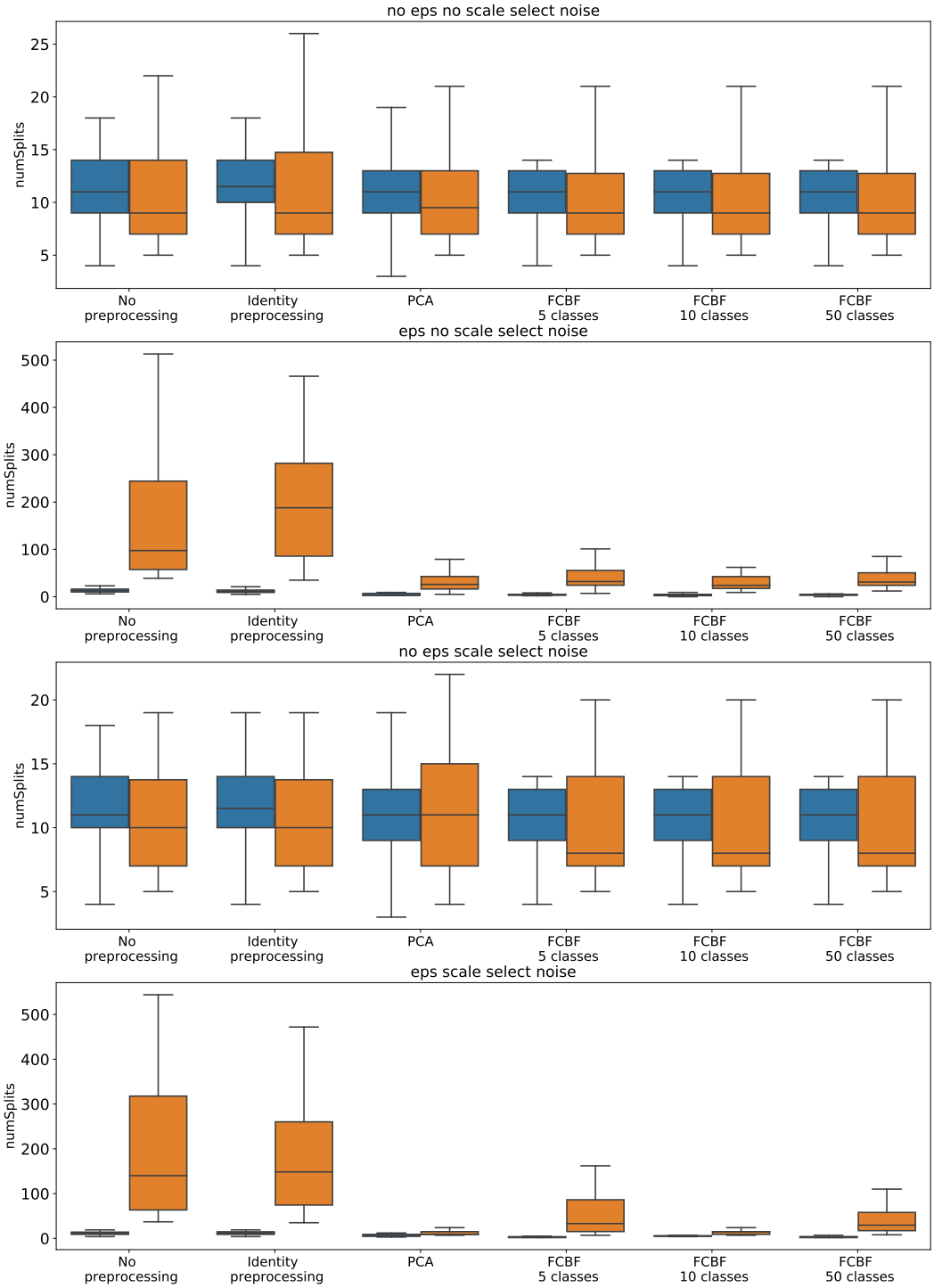

Figure A.6: Number of splits for redundancy experiment with manually selected noise features and runs set to 100

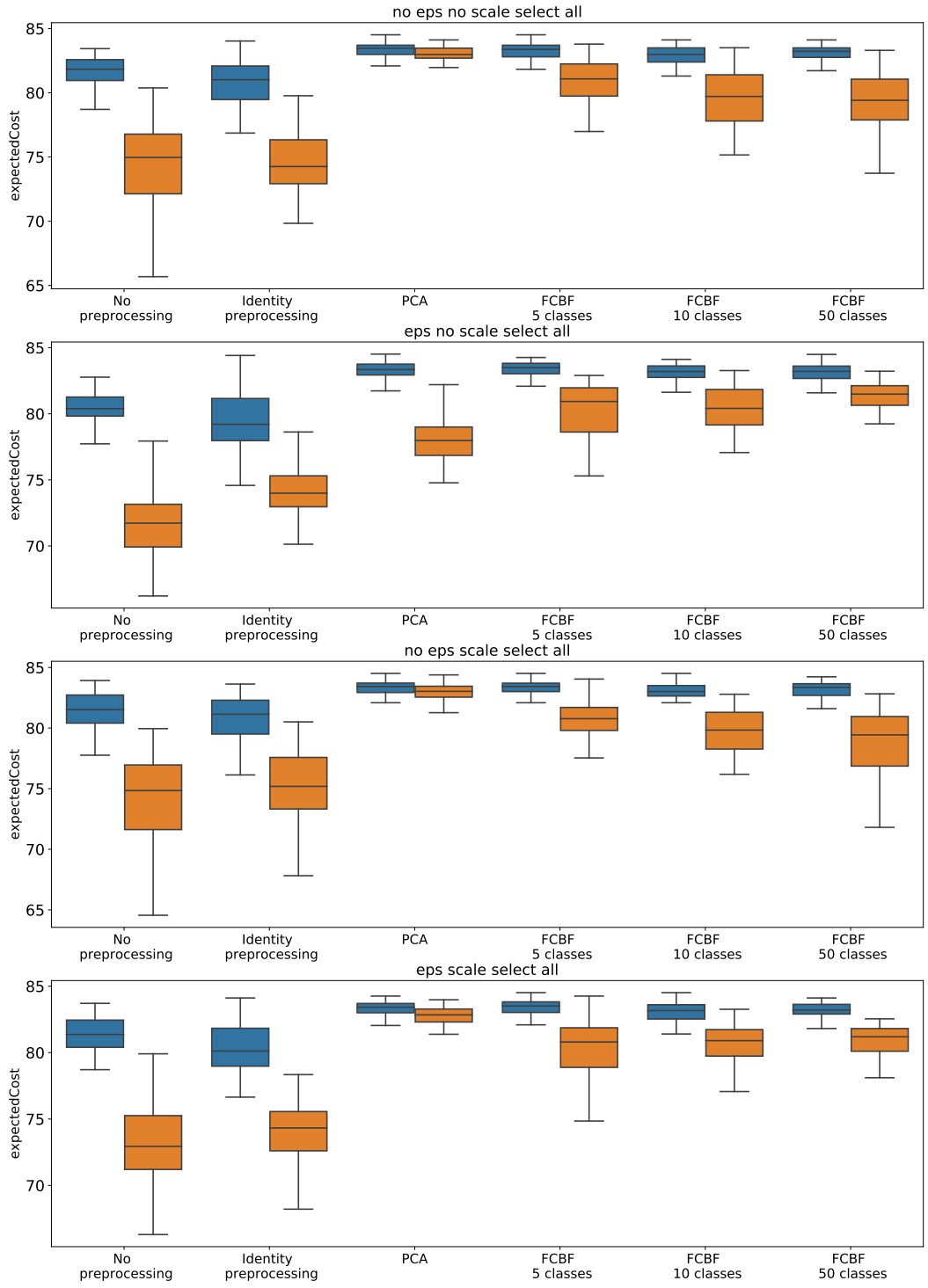

Figure A.7: Expected Cost for redundancy experiment with manually selected all features and runs set to 100

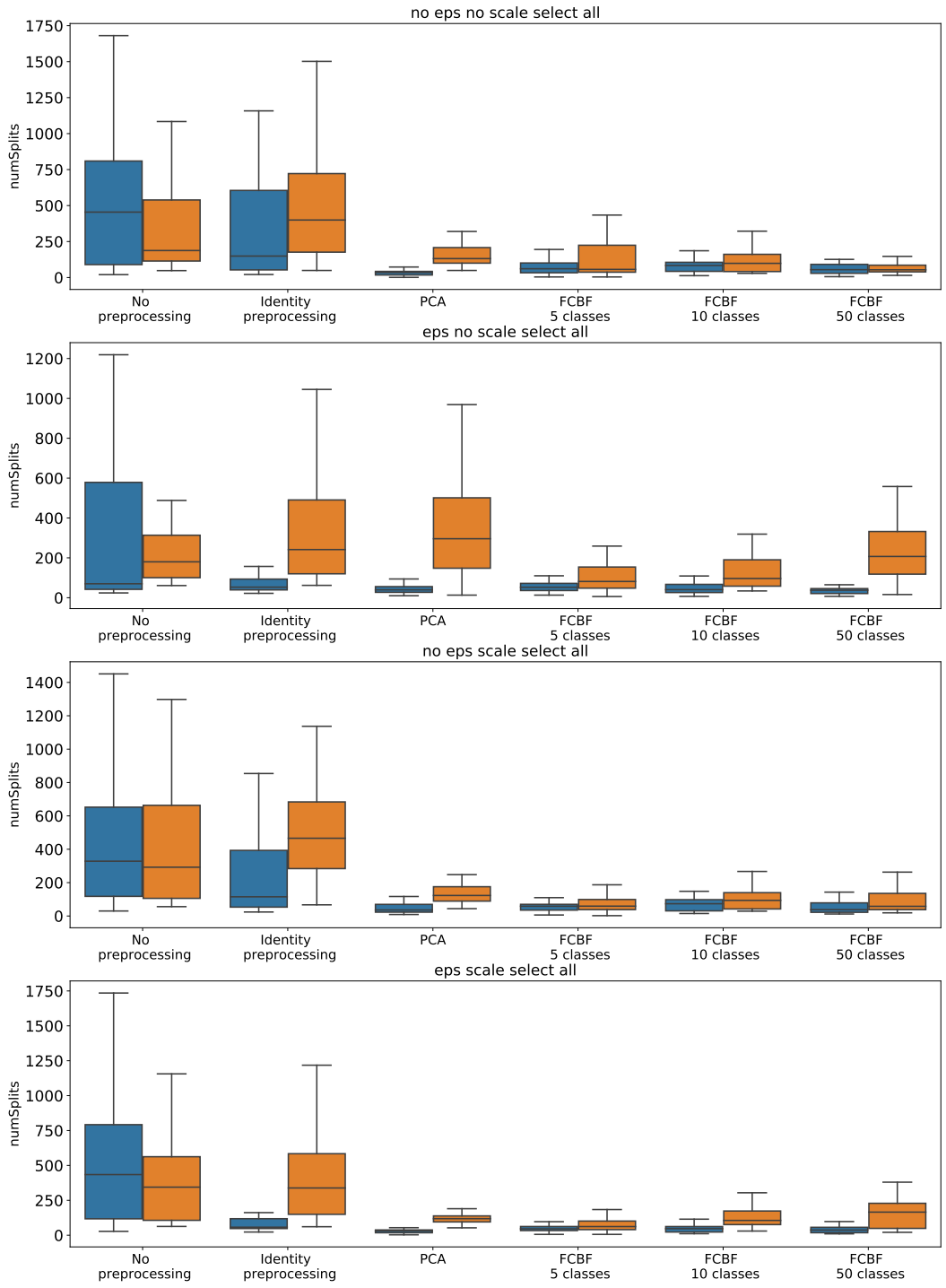

Figure A.8: Number of splits for redundancy experiment with manually selected all features and runs set to 100

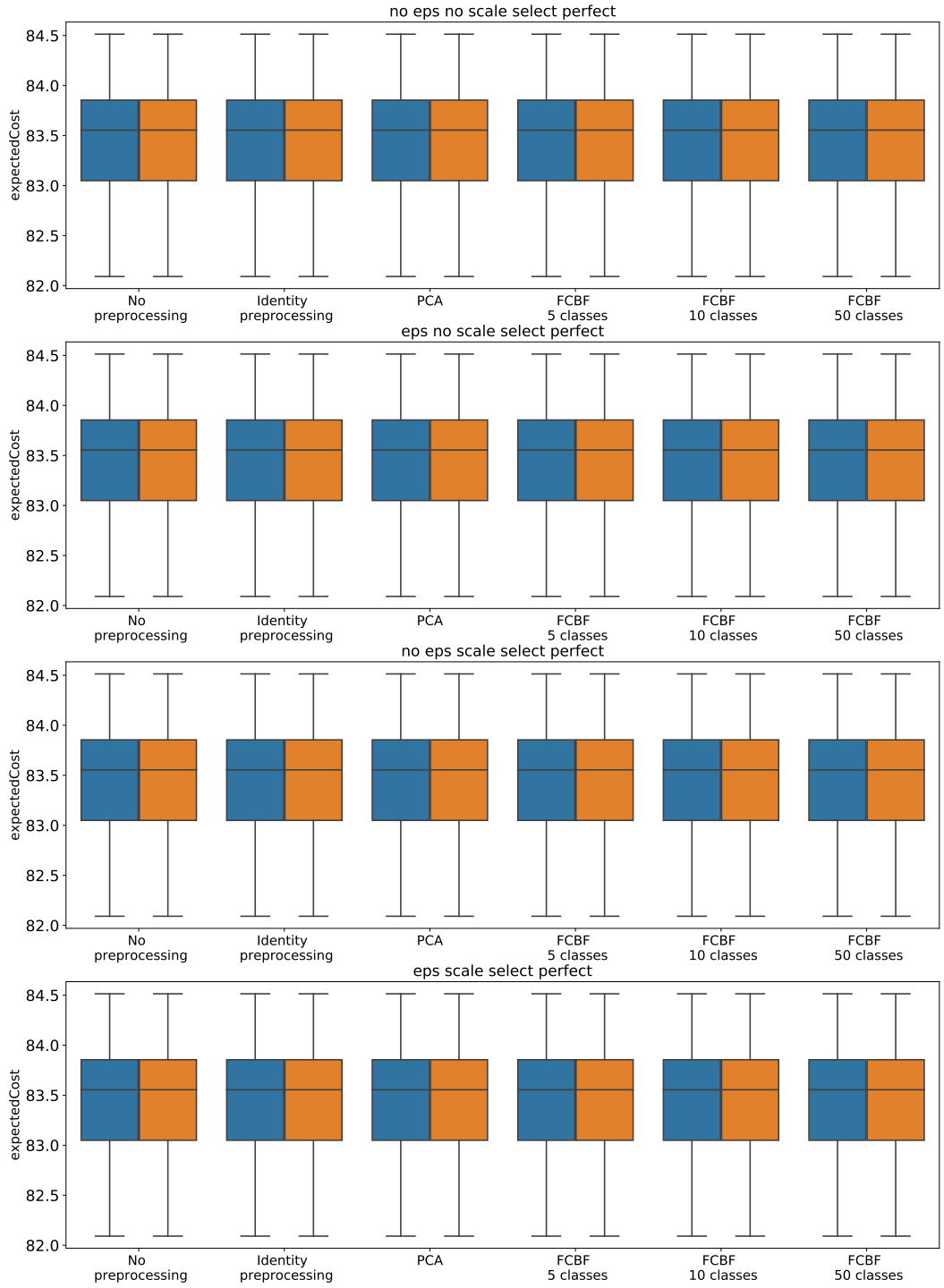

Figure A.9: Expected Cost for redundancy experiment with manually selected perfect features and runs set to 25

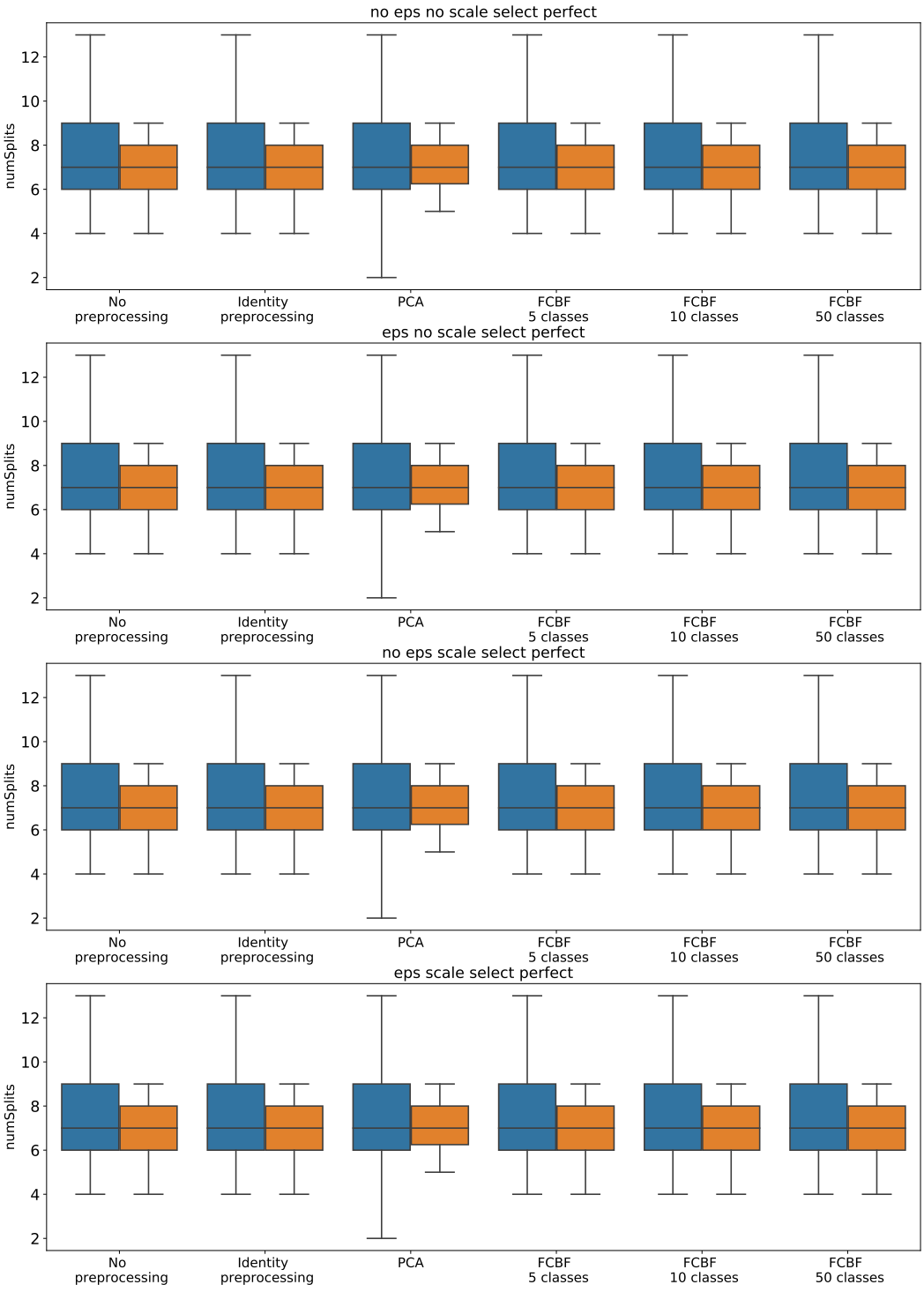

Figure A.10: Number of splits for redundancy experiment with manually selected perfect features and runs set to 25

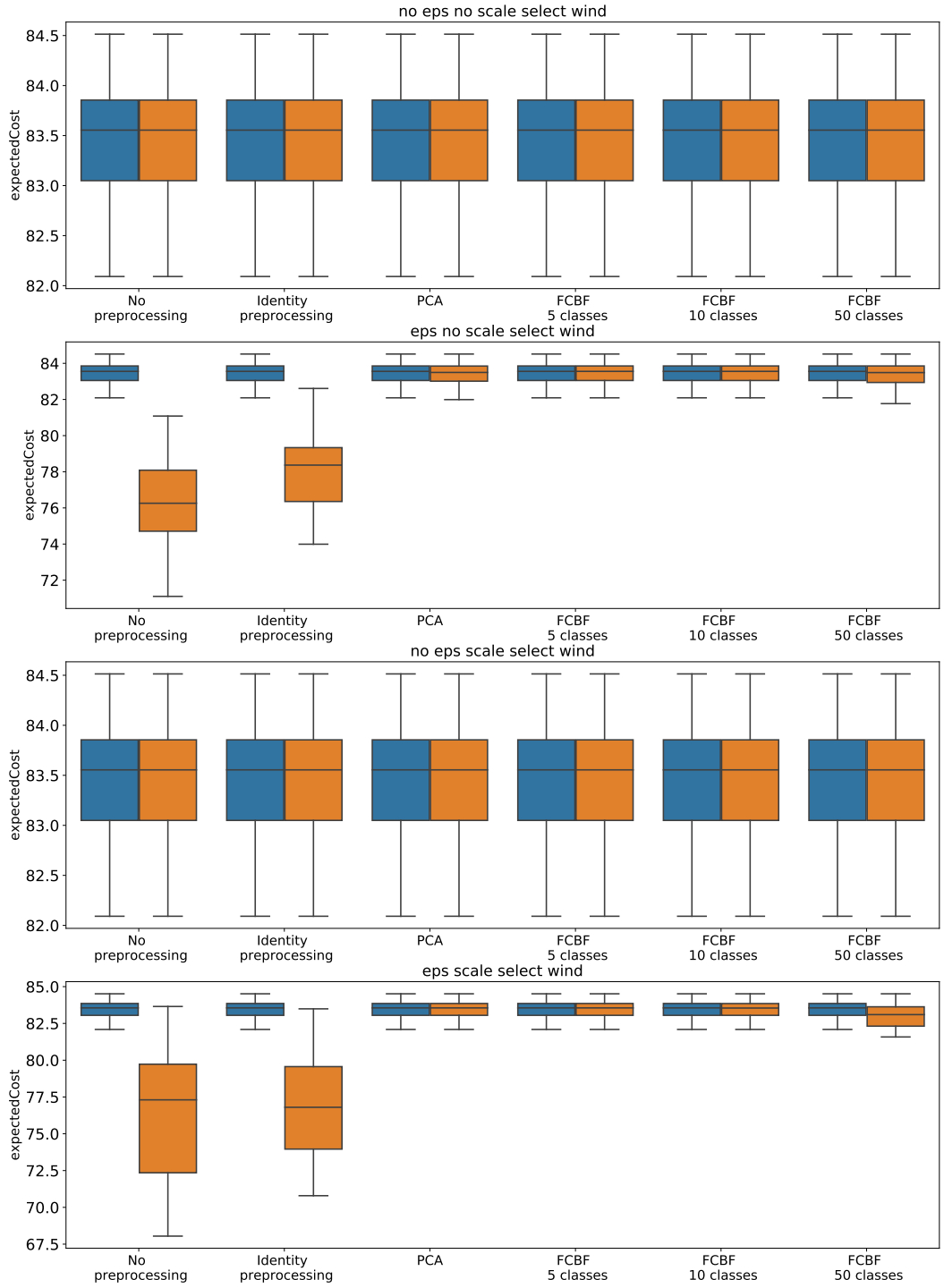

Figure A.11: Expected Cost for redundancy experiment with manually selected wind features and runs set to 25

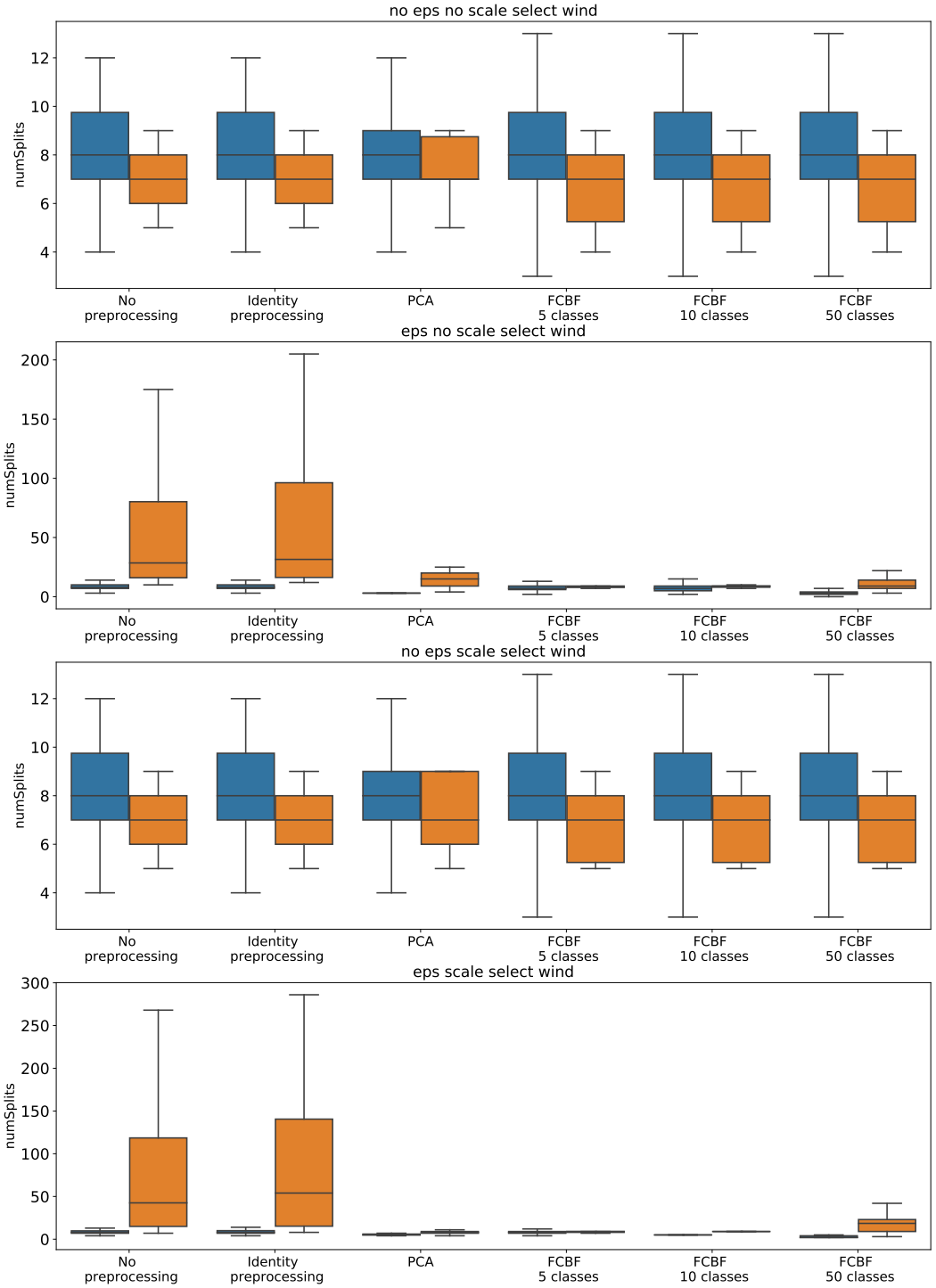

Figure A.12: Number of splits for redundancy experiment with manually selected wind features and runs set to 25

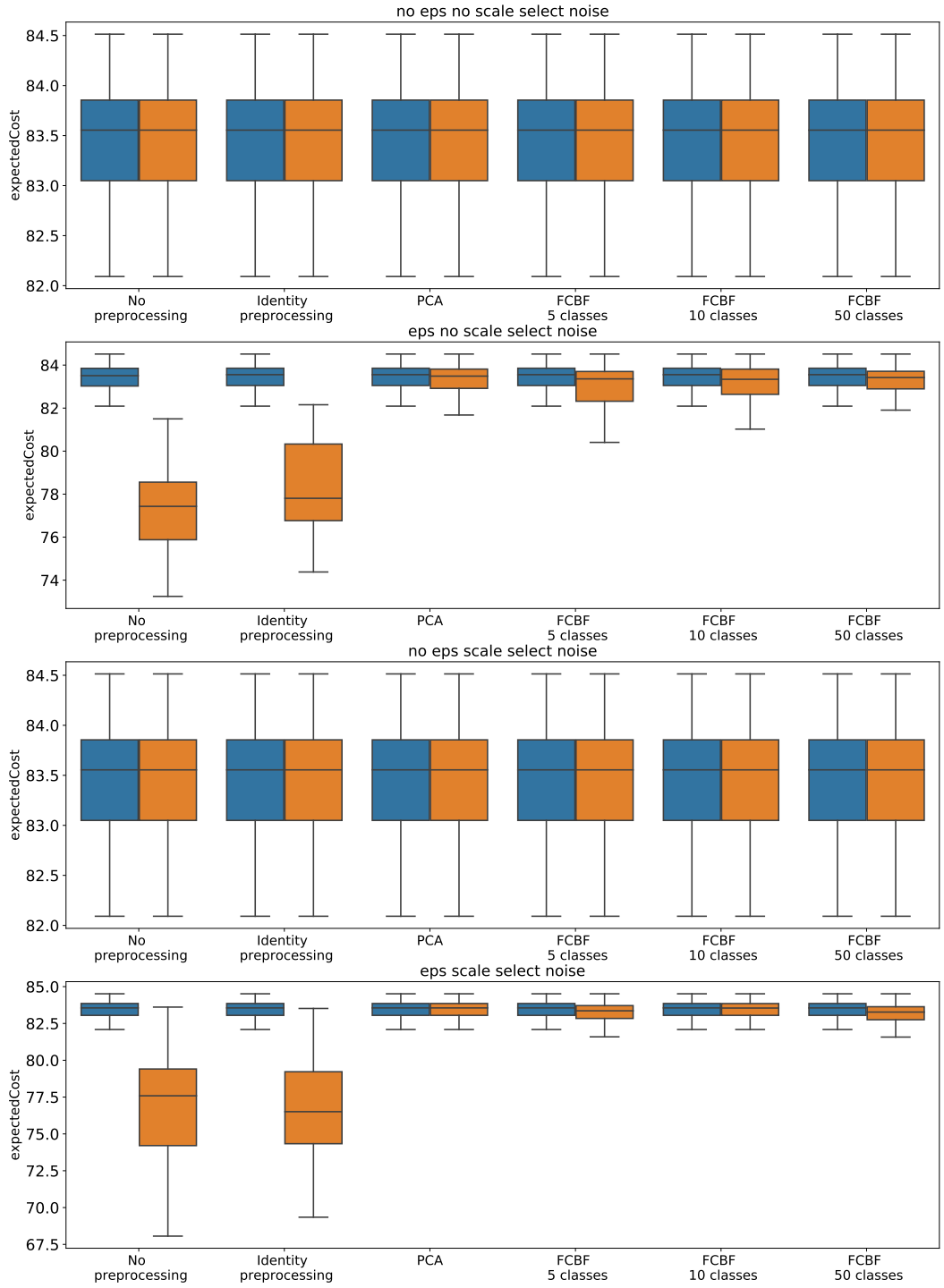

Figure A.13: Expected Cost for redundancy experiment with manually selected noise features and runs set to 25

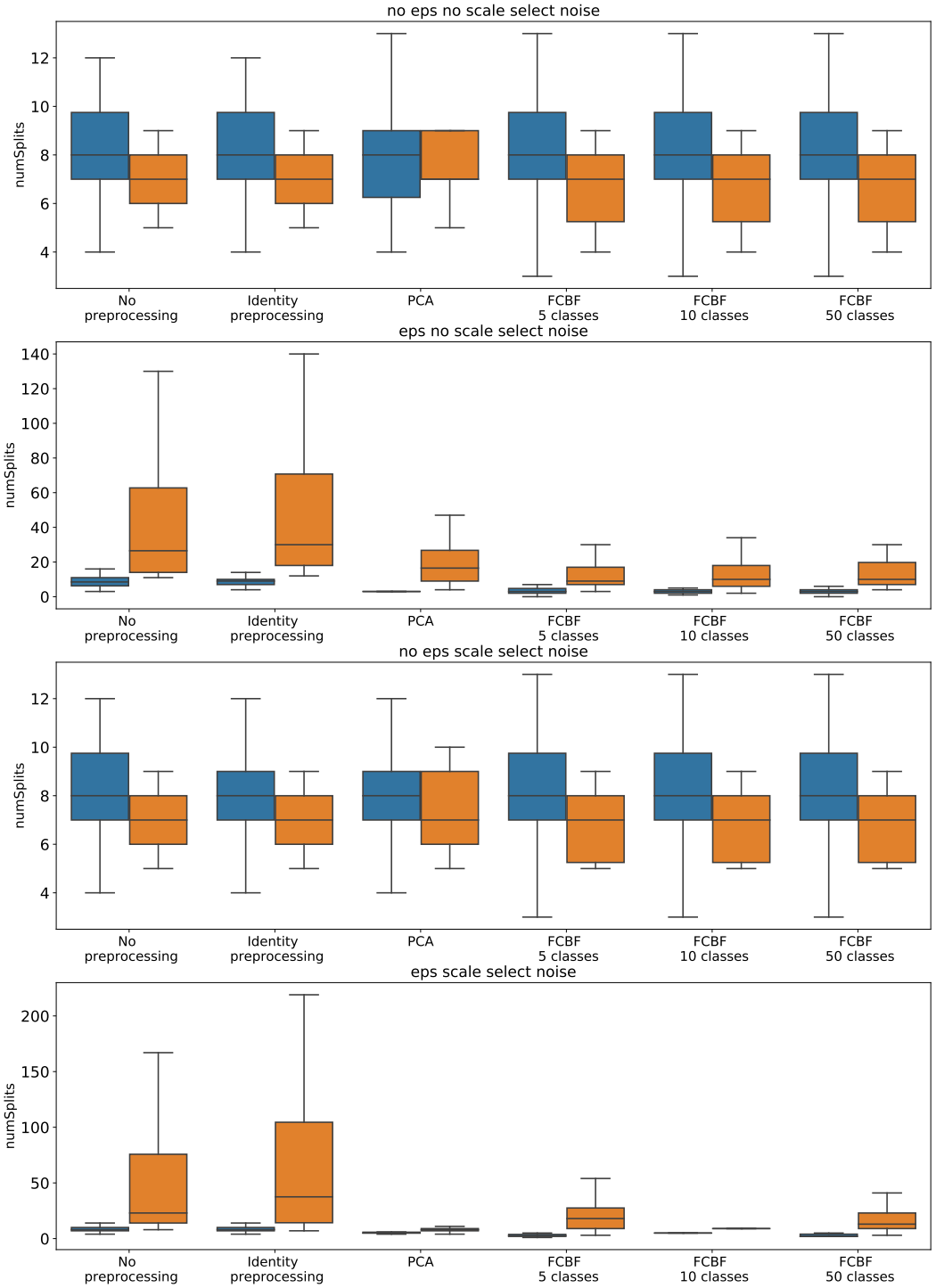

Figure A.14: Number of splits for redundancy experiment with manually selected noise features and runs set to 25

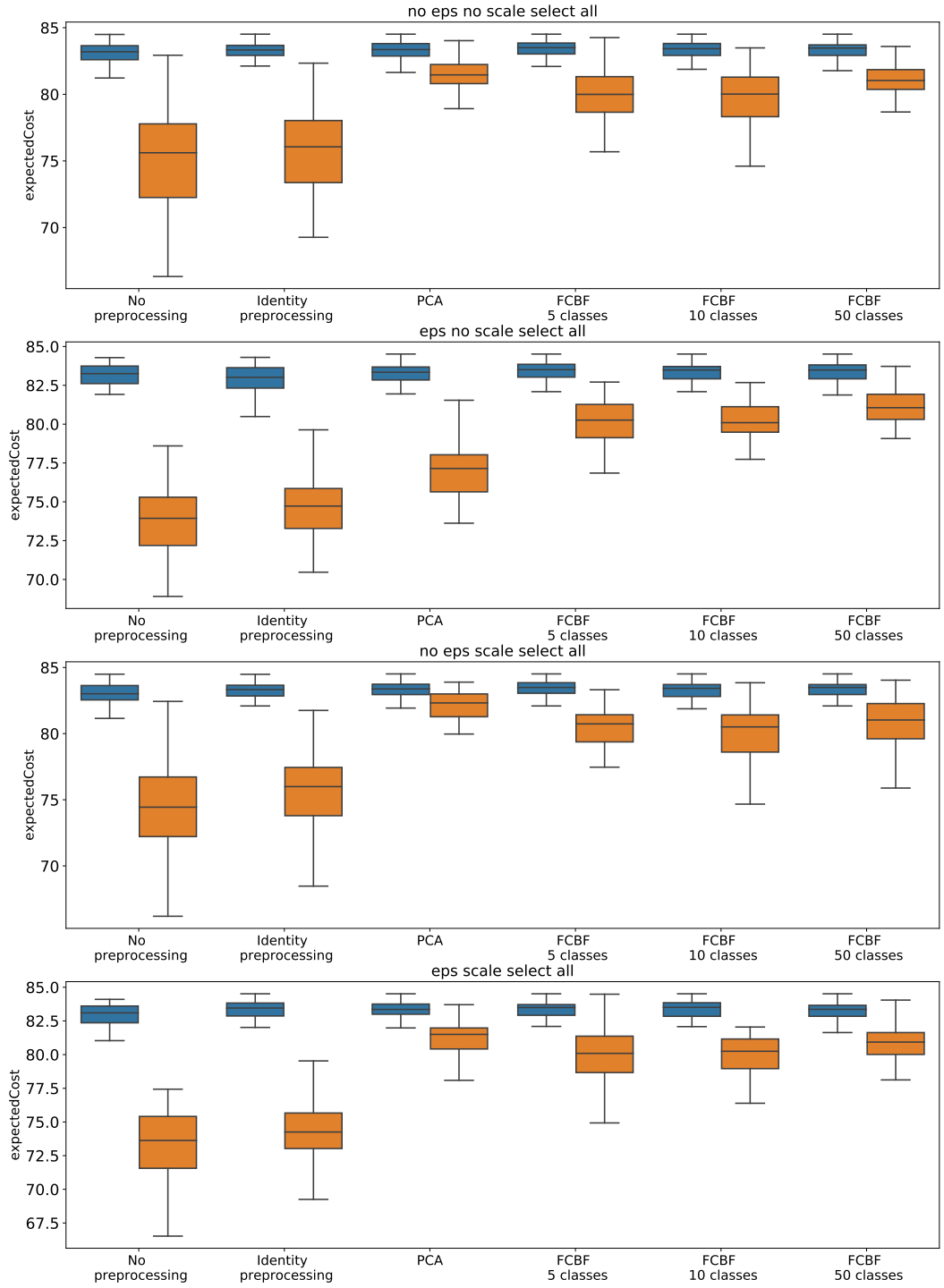

Figure A.15: Expected Cost for redundancy experiment with manually selected all features and runs set to 25

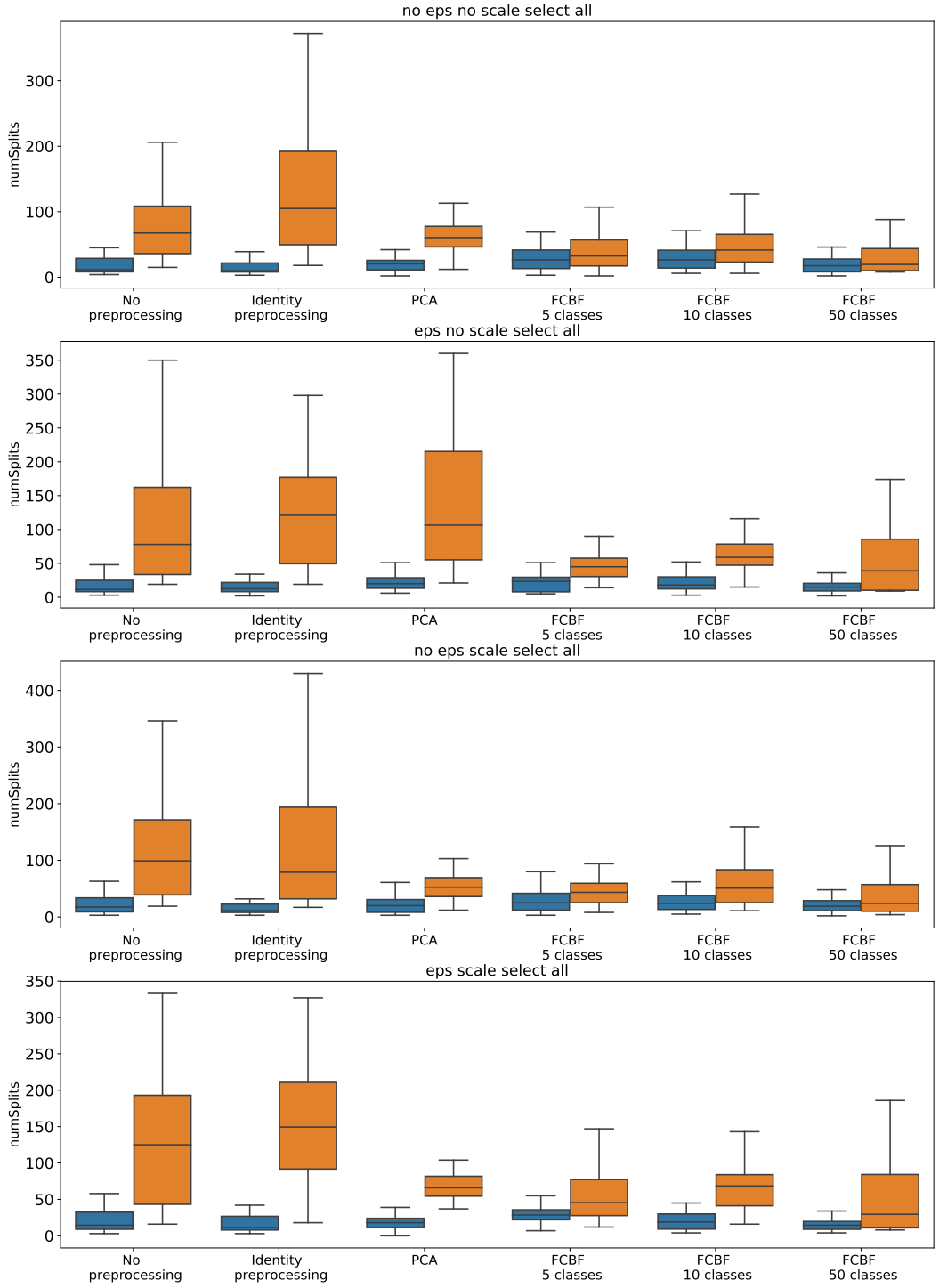

Figure A.16: Number of splits for redundancy experiment with manually selected all features and runs set to 25

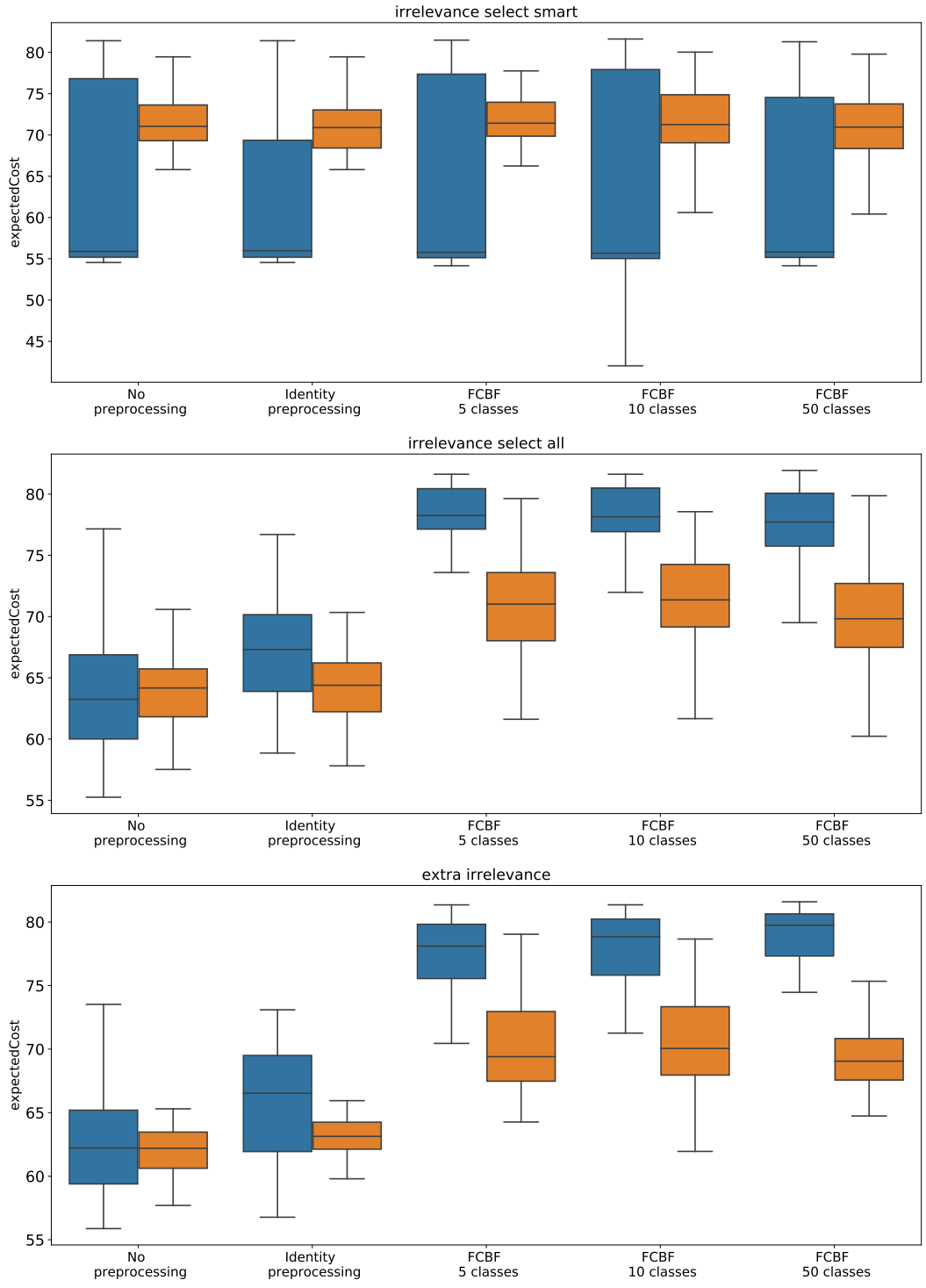

Figure A.17: Experiment results for irrelevant features showing expected cost for runs set to 100

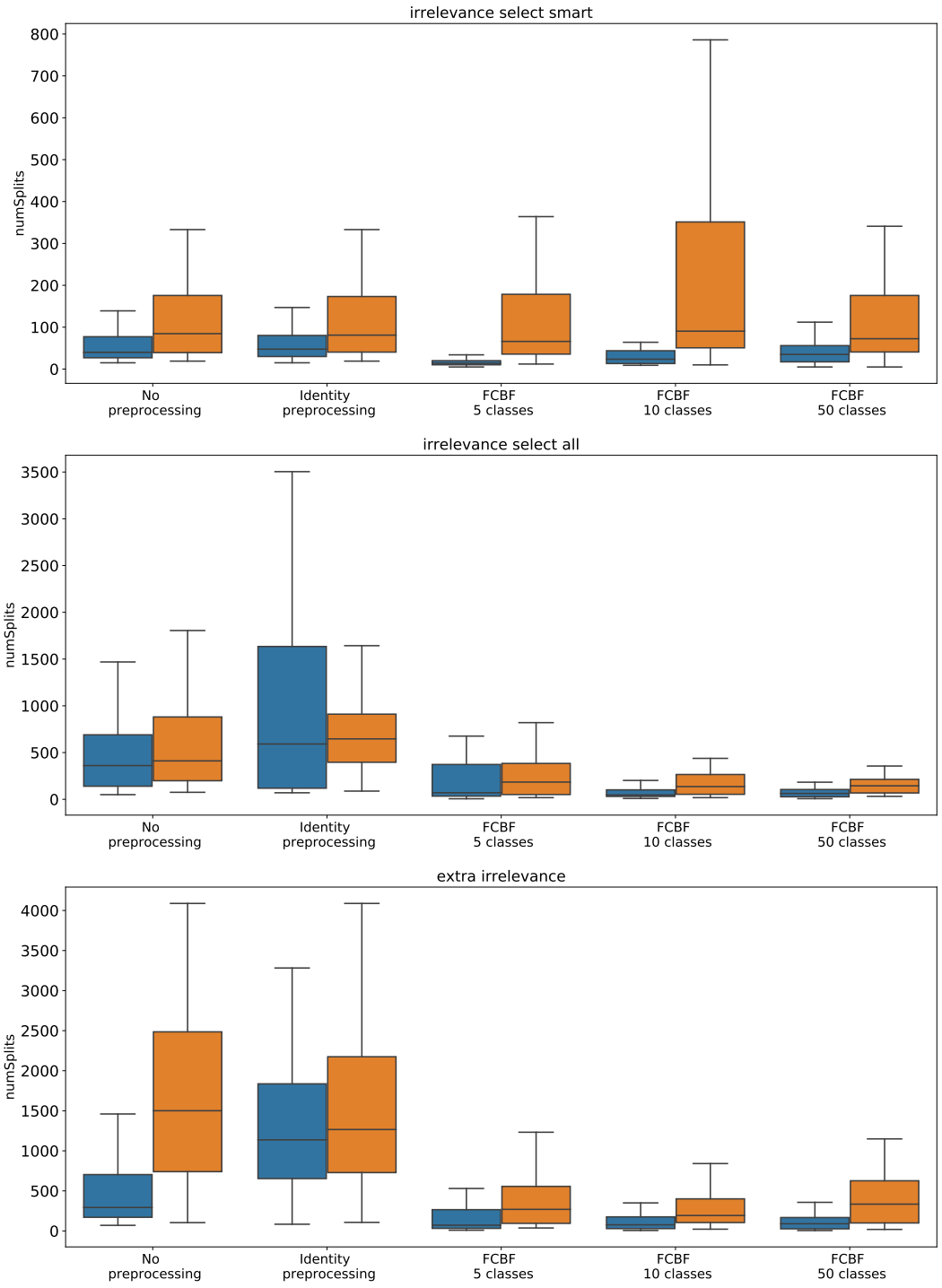

Figure A.18: Experiment results for irrelevant features showing number of splits for runs set to 100

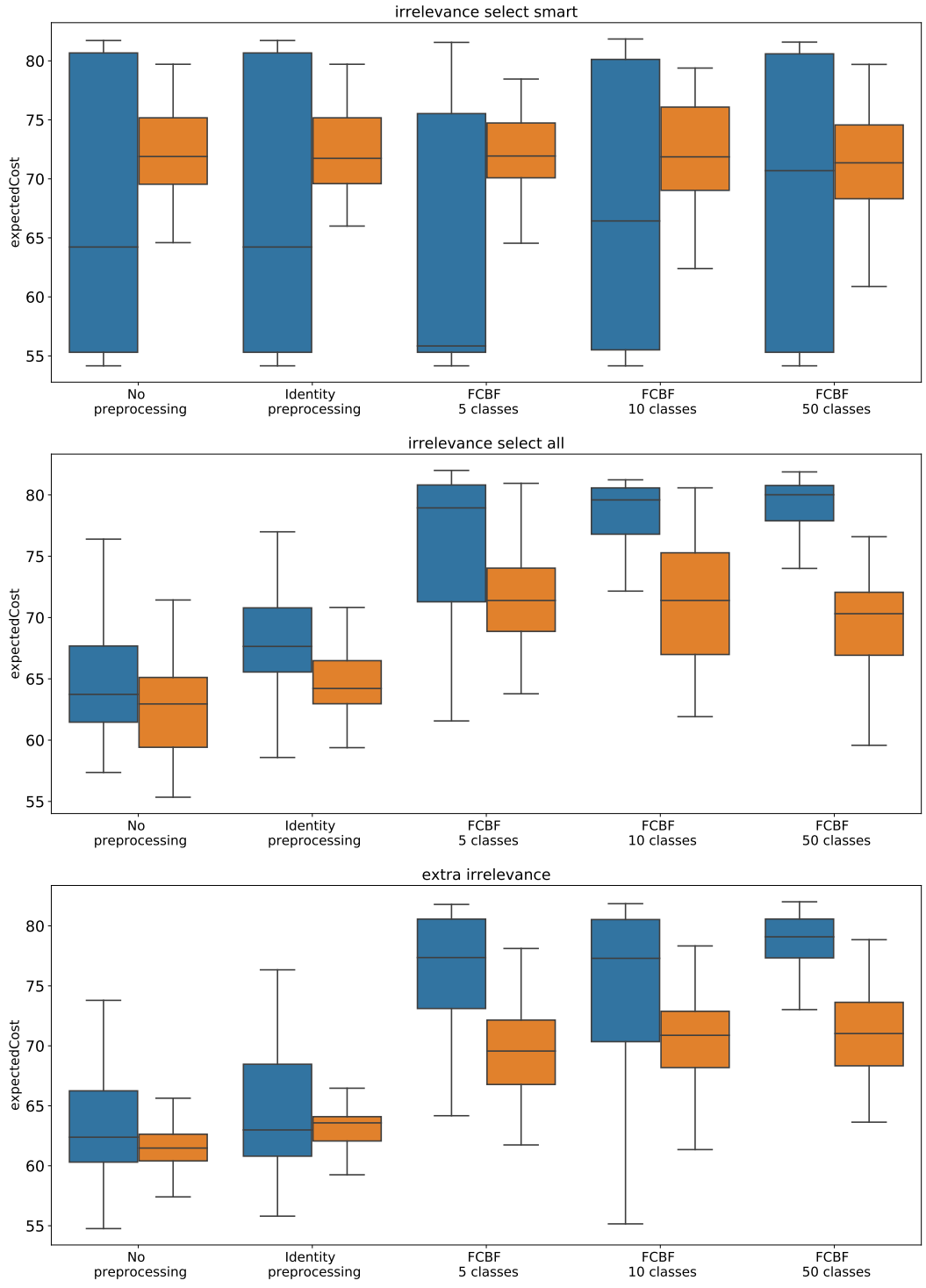

Figure A.19: Experiment results for irrelevant features showing expected cost for runs set to 25

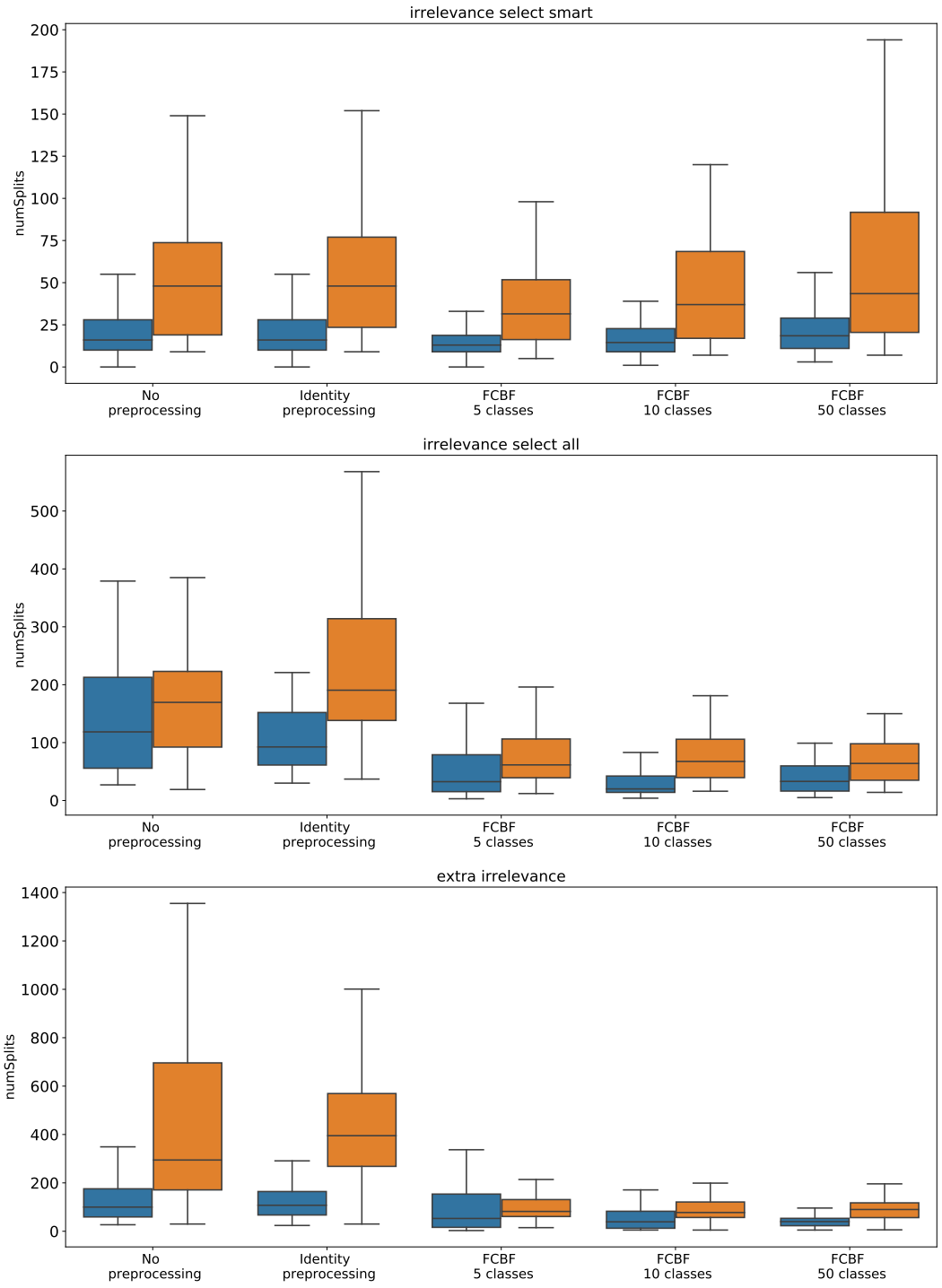

Figure A.20: Experiment results for irrelevant features showing number of splits for runs set to 25