Text classification with generic Logistic-Regression classifier

Marc Moreaux

February 20, 2015

1 Introduction

On the 9th semester, in the computer science department of Aalborg university, students are asked to work on a subject related to their specialization. As the student involved on this report is specializing in Machine Learning, the problematic of the thesis will be related to it.

1.1 Research objective

For this semester, I want to focus on the possibility to train a classifier from a dataset of unknown nature. In other words: There is some challenges existing where an operator has no idea on the dataset he receives. Because of this issue, he can't apply state of the art techniques to train a classifier. During this semester I want to find a dataset of unknown nature and train it with standard, re-applicable methods. Obviously, there'll be some constrains on the dataset and the classifier won't be able to learn any dataset.

1.2 Problem description

For this research objective, we needed a dataset with little information. We picked one provided on the *kaggle* competition website [1]. On this website, a company named *Tradeshift* proposed a multi-label classification problem. Competitors have to propose a coded solution to transform an input into a binary output. The inputs of this competition represent elements of a text-document like a date or a name. In section 2 we give a broader explanation on this inputs as described by *Tradeshift* and in section 3 we dig into and present some possible meaning of the inputs.

The unknown nature of the features in the dataset doesn't permit us to use a straight forward method to feature-engineer them. In this report we propose a specific implementation of a logistic regression algorithm able handle a large number of inputs. This model implementation is presented on section 4 and 3 whereas the performance of that model given some hand-engineered inputs are presented on section 6.

[1] `http://www.kaggle.com/`

1.3 Motivations

I have two main reasons to pick a competition. The first reason is the desire to answer to a real world problem : to work on a subject where a company needs a solution for its business. The second reason is to see how companies can outsource their scientific production to a scientific community.

About electing this particular competition, and as mentioned previously, I was motivated by having dataset with very few information about it. In other words I liked to know that we couldn't apply, with certainty, the state of the art feature-engineering methods on the dataset because we didn't knew, with certainty, what was those features.

I chose a competition from the *kaggle* website because it's a qualitative platform hosting this type of events where the input data, the output data and the evaluation criterion are always presented in a very comprehensible way.

Finally my motivation on this research objective is that features often needs to be hand-engineered to feed a model. Here, we propose a lot of automatically-made features to a model and the model will choose the features it best perform with.

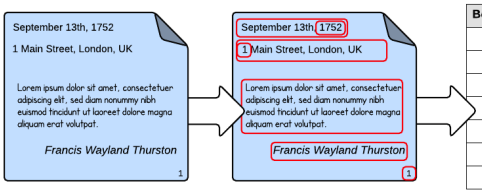
2 Presentation of the competition

A company named *Tradeshift* proposed a multi-label classification problem.

The input data provided represents elements of a text document and the objective of the competition is to classify those elements into 33 possible labels. In this chapter we'll start by defining what does the features of the input data consist of. We will then describe the labels provided. And will finish describing the evaluation criteria of this competition.

2.1 Input data

The competition relies on a dataset representing elements of text documents, we name it X_o. In figure 1(a) you see a letter. For the competition this letter has been analyzed and many elements were taken out of it. The elements are shown on the figure 1(b) as the red boxes and on (c) these elements are listed. In the competition, we see these elements as a 145 feature vector (namely x_{o_i} with i the sample index).



Box nr.	Box text
1	September 13th, 1752
2	1752
3	1
4	1 Main Street, London, UK
5	Lorem ipsum dolor sit amet, consectetuer adipiscing ...
6	Francis Wayland Thurston
7	1

Figure 1: document segmentation from Kaggle website

On figure 7 on page 23 you see The first 55 features of x_{o_1}. We don't know exactly what are the individual features, we just know what could they be. On the official description of the dataset [2], they say that a feature could either be a content feature, a parsing feature, a spatial feature or a relational features.

- "Content feature": is the direct representation of the text element in a hashed format.
- "Parsing feature": indicates which kind of characters are present on the text element. It can be for instance alphanumeric, numeric or text characters.
- "Spatial feature": is about position and size of the text element in the document.
- "Relational feature": gives information about the surrounding of our text element. If there is no values, it means the text element doesn't have a neighbor in that surrounding.

These features can be real valued, discrete valued, boolean valued or text valued. It can also happen that, for a given sample, the feature is *no-valued*. This lack of value is represented by an empty string.

[2] `http://https://www.kaggle.com/c/tradeshift-text-classification/details/evaluation`

id	y1	y2	y3	y4	y5	y6	y7	y8	y9	y10	y11
1	0	0	0	0	0	0	0	0	0	0	0
y12	y13	y14	y15	y16	y17	y18	y19	y20	y21	y22	y23
0	0	0	0	0	0	0	0	0	0	0	0
	y24	y25	y26	y27	y28	y29	y30	y31	y32	y33	
	0	0	0	0	0	0	0	0	0	1	

Figure 2: Labels of sample with ID 1 given by *Tradeshift*

I emphasize here that we still don't know what is on each of the individual features. Taking the example of feature number 1: we don't know what does this feature represent. Reading the file, we see that the feature 1 is either a YES, a NO or a no-value. But we can't say if it's a parsing feature, a spacial feature or a relational feature. We can just eliminate "content feature" as this feature is represented in a hash format.

We can enjoy this last observation to assert that features 3, 4, 34, 35, 64, 65, 94, 95, 61 and 91 are content features as they all are hashed values (partially visible on figure 7 on page 23).

2.2 Labels

The text elements just described represent something. For instance, the text elements seen on figure 1 on the preceding page represents a date, a number, an address, a text-body, a signature and a page number. In the competition we are given a vector of 33 labels for each of the text elements. That one can have one or more labels. For example an element could be labeled as a number and a page number. Figure 2 is the label vector corresponding to the example seen above, where an extract of the 1st sample where shown on figure 7 on page 23.

As for the features, we don't know what a label correspond to. We are not given any names for the 33 label. We just have a number in the range from 1 to 33 as label.

2.2.1 files

All the data mentioned until now is provided through 3 csv files.

- "train.csv": contains the training data X_o. It is an array of 1.7m lines of samples and 145 columns of features. Each cell has a either a real, a discrete, a boolean, a text value or a *no-value*.
- "trainLabels.csv": contains label of data Y. It is an array of 1.7m lines of samples and 33 columns of labels. Each cell is either 0 or 1.
- "test.csv": contains the testing data. It is an array with merely 0.4m lines of samples and 145 columns of features. Each cell has a either a real, a discrete, a boolean, a text value or a *no-value*.

2.3 Evaluation criterion

The work is evaluated by sending a csv file on the *kaggle* website. The csv file is then given a score by the system.

id_label	pred
1_y1	0.01
1_y2	0.05
1_y3	0.98
2_y1	0.1
2_y2	0.88
2_y3	0.92

Figure 3: Prediction file of 2 samples with ID 1 and 2, and with 3 labels.

That csv file has 2 columns and merely $33 * 0.4m = 58$m lines. Figure 3 is and example of this csv file. Each row of this file give an information about the pair {sample i and label j}. The first column is a string encoding giving name this pair and the second column is the prediction corresponding to this same pair.

Once submitted, the file receive a score through a scoring function. This function is the negative logarithm of the likelihood, averaged over N_t test samples and K labels. Mathematically, this function is defined as follows:

$$\begin{aligned} LogLoss &= \frac{1}{N_t \cdot K} \sum_{idx=1}^{N_t \cdot K} LogLoss_{idx} \\ &= \frac{1}{N_t \cdot K} \sum_{idx=1}^{N_t \cdot K} [-y_{idx} \log \hat{y}_{idx} - (1 - y_{idx}) \log (1 - \hat{y}_{idx})] \qquad (1) \\ &= \frac{1}{N_t \cdot K} \sum_{i=1}^{N_t} \sum_{j=1}^{K} [-y_{ij} \log \hat{y}_{ij} - (1 - y_{ij}) \log (1 - \hat{y}_{ij})] \end{aligned}$$

Where $\hat{y}_{idx}$ is the prediction of a sample for a given class and y_{idx} is the true binary value stating if a sample belongs to a class or not.

This function gives punishment for wrong confident predictions.

For instance, the LogLoss of one sample with one label predicting 0.001 instead of 1 (a confident wrong prediction) is $-\log(0.001) = 6.90776$ whereas the log-loss of one sample with one label predicting 0.1 instead of 1 (a less confidently wrong prediction) is $-\log(0.1) = 2.30259$. As an indication, the LogLoss of predicting everything with 0.5 probability is $-log(0.5) = 0.69315$ and the LogLoss of 0.95 (for a $Y = 1$) is $-\log(0.95) = 0.05129$

3 Analysis of the input data

In this section we have a closer look on the input data and we try to bring up some interesting hypothesis given the few informations we have.

The training data is composed of 1.7 million samples. Each samples has 145 features. As described in section 2 the features can either be a content feature, a parsing feature, a spacial feature or a relational feature and they can be real, discrete, boolean, text or no valued. In order to analyze all the features we quantified all of them.

3.1 Input quantification

The ai here is to give a real (or integer) value to eachone of the features. If we take the example visible on figure 8 on page 24, the features 1-4, 7, 10-14, 24-27, 30-35, 41-45 and 55 are non numerical. We modified these inputs as follow:

- The YES/NO values were transformed to -1/1.
- The empty values were transformed to 0.
- And, as there is 986'837 different hash-codes in both the training and testing samples, the hash-codes were transformed to an index integer in range [1;986837].

This transformation holds for the rest of the paper.

3.2 Five blocs

Taking a closer look at the data, one can notice that each features seems to be repeated 5 times. There is 10 hash features, 50 binary features, 50 real features and 35 integer features per samples. All these numbers are divisible by 5. When looking at the composition of the data, the columns were approximately matching. In other words, features 1 to 29 had the same data types in the same order than the features 32 to 60, than feature 62 to 90, than feature 92 to 120 and than features 30, 31, 61, 91, and 121 to 145. The first data sample has been reformatted to feat this observation and is visible on figure 8 on page 24. For simplicity we will now always mention the features as they are ordered in that figure. In other words, the five blocs are now referred as features {1 to 29}, {30 to 58}, {59 to 87}, {88 to 116} and {117 to 145}.

After this observation, an other argument came in favor of this hypothesis: on the competition's website, the authors mentioned that the relational data included information about the surrounding text blocks in the original document. If there were not that surrounding text block (e.g. a text block in the top of the document wouldn't have any other text block upper than itself) it's features would be empty (*no-value*). Looking at the input file, we could see that some of the five blocs mentioned right before were some times lacking values for all of their features. For instance, the sample number 2 given in 'train.csv' has features {1 to 29} equal to -1 or to a *no-value*.

From here we had a new argument in favor of the 5 blocs hypothesis and a new one concerning the meaning of these blocs. Each one of the five blocs would represent an element (e.g. a title or a date). And their order of apparition in the feature order correspond to a specific hierarchical neighbor. Because the

last of the five blocs ({117 to 145}) were never no-valued, we believed that this block was the data to classify.

3.3 features

We keep the *5 blocs hypothesis* as true. We now have $145/5 = 29$ different features.

We are going to see the data distribution of some of these features. To understand how the data distribution was rendered we will consider an example: the distribution of the feature 1. We collect all the samples of features 1, 30, 59, 88 and 117 (which correspond to the first feature of each bloc) and count the amount of time each of the values are repeated. The samples are taken from both the training set and the testing set, there is 2.1 million samples. On figure 7 on page 22 and figure 7 on page 25 are represented all the hashes, real and integer features. The binary features (yes/no) are not represented. The reason for this absence is that the plot only showed that both of the YES and the NO data were used.

Taking a closer look at the features 3 and 4 (the hashed features), we notice that there is merely half less hash values than samples (0.98/2.1). Furthermore, there is 0.98 million hashes for $2.1 * 5 = 10.5$ millions hash entries. From this observation we understand that there is redundancy in the use of hash-values. Later we will see how we tried to take advantage of the redundancy. We also notice that few hashes (the first values on the plot) are extremely present on all the features.

Comes after the integer and float valued features:

- Features 15, 17, 18 and 27 all are integer features. They exist in the range of positive natural numbers and their low values are frequently used.
- Features 22 and 23 are also integer features. They exist in the range of positive natural numbers and few of its values are often reused.
- Features 6, 7 are float features. They are in range [0,1].
- Features 5, 8 are float features. They are in range [0,3]. As they come for together and seeing their distribution, we could believe that features 5, 6, 7, 8 represent positions relative to a page-width and page-height.
- Feature 9, 16, 28, 29 are float feature. They have most of their values in range [0,1] but few exists outside of this bounds.
- Feature 19 and 21 are also float features. Feature 19 is in [-0.5,1] and feature 21 is in range [0,1]

Seeing those distributions doesn't help much on understanding what is the feature meaning.

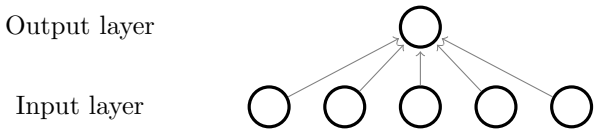


Figure 4: 5 neurons logistic regression model

4 Model

In this section we describe the theory needed for the model we use in our project. We describe, the model, the training criterion, the convergence algorithm and the regularization term we use.

4.1 Logistic regression model

The model is a logistic regression. To better understand this model, we define the two words:

- Regression: Regression is a set of statistical methods often used to analyze the relation of a variable towards one or many others.
- Logistic: A variable is called logic, if it varies in between a true and a false state. A logistic function represent this variation through a function $f : \mathbb{R} \to [0, 1]$. The standard logistic function is called sigmoid and is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Therefore a logistic regression model aims at guessing the state of a logic variable $Y \in \{True; False\}$ knowing the states of many other inputs $X \in \mathbb{R}^n$.

The descriptions that follows is inspired form both a book [4] and a paper [3]. Figure 4 is a representation of a logistic regression model.

Lets consider we have a set of n samples. $x_i \in \mathbb{R}^m, \forall i \in [1...n]$ is the input vector of dimension m for a sample n and $y_i \in \{0, 1\} \forall i \in [1...n]$ is the value to predict corresponding to it. We also note x_{ij} the elements in dimension j of the vector x_i.

The objective of a linear regression is to predict the value y of a new input sample x. The key value in regression is the conditional mean: $E(Y|X = x)$. You read this quantity as "expected value of Y given x" and we'll also refer to it as our "prediction". In order to have a good regression, you want to maximize this quantity.

In our model, we multiply each dimension of the input x_{ij} with a parameter β_j and then aggregate together the result of this multiplication. If we consider $\beta \in \mathbb{R}^m$ the vector composed of the β_j elements, the operation is:

$$g(\beta, x) = \sum_{j=1}^{m} \beta_j x_{ij} = \beta^T x_i$$

After this step, we use a sigmoid function to reduce the output space to $[0, 1]$:

$$\sigma(\beta, x) = \frac{1}{1 + e^{-\beta^T x_i}} = E(Y|X = x_i)$$

This function is the one expressing our expected value of Y given x_i (the prediction). We are now going to describe how to maximize this expected value.

4.2 Maximum log-likelihood [4]

The likelihood function $l(\beta)$ is expressing the the probability of the observed data x as a function of the parameters β. We want to find the parameters β that maximizes the likelihood function.

The likelihood function is either $P(Y = 0|x_i)$ if $y_i = 0$ or $P(Y = 1|x_i)$ if $y_i = 1$. A convenient way to re-write this likelihood for a given sample (x_i, y_i) is:

$$l(\beta_i) = E(Y|x_i)^{y_i}[1 - E(Y|x_i)]^{1-y_i}$$

Since the expected values of Y are assumed to be independent, the likelihood function through the dataset is the product of all the sample likelihoods:

$$l(\beta) = \prod_{i=1}^{n} E(Y|x_i)^{y_i}[1 - E(Y|x_i)]^{1-y_i}$$

An easier expression to work with is the logarithm of this likelihood function, the log-likelihood:

$$\ln(l(\beta)) = \sum_{i=1}^{n} [y_i \ln[E(Y|x_i)] + (1 - y_i) \ln[E(Y|x_i)]$$

In machine learning it's common to write the prediction with a hat on the predicted variable: $\hat{y}$. With this notation the log likelihood is written:

$$\ln(l(\beta)) = \sum_{i=1}^{n} [y_i \ln(\hat{y_i}) + (1 - y_i) \ln(1 - \hat{y_i})]$$

Differentiation of the log-likelihood We now want to find the parameters that maximizes this log likelihood function or, in other words, that best predict our variable to guess Y. To do so, we search for the maximum of the function by differentiating it with respect to its parameter β and search for the zeros of this function. To differentiate, we take advantage of the chain rule:

$$\frac{\partial \ln(l)}{\partial \beta} = \frac{\partial \ln(l)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial \beta} \tag{2}$$

Where $h = \beta^T x_i$. knowing that:

$$\begin{aligned} \frac{\partial \ln(l)}{\partial \hat{y}} &= \sum_{i=1}^{n} \frac{y_i}{\hat{y}_i} + \sum_{i=1}^{n} \frac{1 - y_i}{1 - \hat{y}_i} \\ \frac{\partial \hat{y}_i}{\partial h} &= \frac{\partial \sigma(h)}{\partial h} = \hat{y}_i(1 - \hat{y}_i) \\ \frac{\partial h}{\partial \beta} &= x_i \end{aligned} \tag{3}$$

We get:

$$\frac{\partial \ln(l)}{\partial \beta} = \sum_{i=1}^{n} x_i[y_i(1-\hat{y}_i) + (1-y_i)\hat{y}_i]$$

We notice that, when $y_i = 1$, the differential of $ln(l) = \sum_{i=1}^{n} x_i(1-\hat{y}_i)$ and when $y_i = 0$, the differential of $ln(l) = \sum_{i=1}^{n} x_i(0-\hat{y}_i)$. Therefore we rewrite:

$$\frac{\partial \ln(l)}{\partial \beta} = \sum_{i=1}^{n} x_i(y_i - \hat{y}_i)$$

As mentioned, to find the maximum log likelihood we now need to find the zeros of this function.

$$\frac{\partial \ln(l)}{\partial \beta} = 0$$

To do so, we use the so-called "gradient descent algorithm".

4.3 Gradient descent algorithm

Gradient descent is an algorithm aiming at finding the minimum of a function. It's an iterative algorithm improving at each step its chances to be closer to an optimum value.

Gradient descent works as follow:

Consider a stricly convex function $f : \mathbb{R}^n \to \mathbb{R}$ that you want to minimize. In order to minimize this function, you have to find $\nabla f(x) = 0$ where ∇f is:

$$\nabla f = \frac{\partial}{\partial_x} f \cdot \vec{i}$$

To start searching for the minimums of this function, you initialize your algorithm with a given input vector $x^{(0)}$ and a given learning rate α. While the termination criteria $\nabla f(x) = 0$ or $\nabla f(x) < \epsilon$ is not met, you update the following function:

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

Example

Imagine you have the following function to minimize:

$$f\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = x_1^2 + 2(x_2 - 1)^2 \forall x_1, x_2 \in [-10, 10]$$

The first step you take is to initialize your x vector. For instance we begin at $f(x_1 = 1, x_2 = 1) = 1$ and we choose a learning rate $\alpha = 0.5$. We can compute the gradient of $f(x_1, x_2)$:

$$\nabla f\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2x_1 \\ 2(x_2 - 1) \end{pmatrix}$$

Then we apply the update formula:

$$x_{t+1} = x - \alpha \nabla f(x)$$

$$x_{t+1} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - \alpha \nabla f \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

$$x_{t+1} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} - 0.5 \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

$$x_{t+1} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

We now are at $f(x_1 = 1, x_2 = 1) = 0$ and the gradient $\nabla f(x_1 = 1, x_2 = 0) = (0, 0)$. Now, we could run again the algorithm with other initial parameter to discover whether or not the minimum we just found is the global one or not.

Quadratic approximation

There is other ways of interpreting gradient descent. One of them is considering the quadratic approximation of the loss function f at the current point x_t where we replace the Hessian term of the approximation $\nabla^2 f(x_t)$ by $1/\alpha I$:

$$f(y) = f(x_t) + \nabla f(x_t)^T (y - x_t) + \frac{1}{2\alpha} \|y - x_t\|_2^2$$

To find our x_{t+1} we search for the minimum of this quadratic approximation of f at x. To match our sources, we now note $g \leftarrow \nabla f(x)$ which gives us:

$$\begin{aligned} x_{t+1} &= \operatorname*{argmin}_x f(x_t) + g \cdot (x - x_t) + \frac{1}{2\alpha} \|x - x_t\|_2^2 \\ &= \operatorname*{argmin}_x g \cdot x + \frac{1}{2\alpha} \|x - x_t\|_2^2 \end{aligned} \tag{4}$$

Setting the derivative to zero and developing this expression, we find as before :

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

On expression 4 we see two terms. The first one ($f(x_t) + \nabla f(x_t)^T (x - x_t)$) is a linear approximation to f whereas the second one ($\frac{1}{2\alpha} \|x - x_t\|_2^2$) is a proximity term to x weighted with $\frac{1}{2\alpha}$ stating that we don't want to move too far from our current iterate x_t.

4.4 Follow The Regularized Leader - Proximal

The algorithm we use in our program is "Follow The Regularized Leader - Proximal" (FTPRL). I order to understand this algorithm, we are first going to see a closely-related algorithm: the Composite-Objective MIrror Descent (COMID).

Mirror Descent [3]

Gradient descent is a type of mirror descent. As we saw previously, gradient descent can be understood a quadratic approximation of f at x_t. Now, Mirror descent is different in the sense that, instead of using the L2 norm as distance it uses a Bregman divergence $\Delta B(x, x_t)$ to measure the update distance.

[3] `http://www.cs.cmu.edu/~ggordon/10725-F12/schedule.html`

$$x_{t+1} = \underset{x}{\operatorname{argmin}}\, f(x_t) + \nabla f(x_t)^T (x - x_t) + \Delta B_{1:t}(x, x_t)$$
$$= \underset{x}{\operatorname{argmin}}\, g \cdot x + \Delta B_{1:t}(x, x_t)$$

Composite-Objective Mirror Descent [5]

The COMID adds to the mirror descent a regularization functions noted Ψ. Such that the updates becomes:

$$x_{t+1} = \underset{x}{\operatorname{argmin}}\, f(x_t) + \nabla f(x_t)^T (x - x_t) + \Delta B_{1:t}(x, x_t) + \Psi(x)$$
$$= \underset{x}{\operatorname{argmin}}\, g \cdot x + \Delta B_{1:t}(x, x_t) + \Psi(x)$$

Follow the regularized leader - proximal [5]

FTPRL is a COMID with a Bregman divergence $\Delta B(x, x_t) = \frac{1}{2} \left\| Q_t^{1/2}(x - x_t) \right\|_2^2$. This Bregman divergence is adaptive to the time t and the features as Q_t is chosen such that $Q_{1:t} = diag(\sigma_{t,1}, ..., \sigma_{t,n})$ and $\sigma_{t,i} = \frac{1}{\gamma} \sqrt{\sum_{s=1}^{t} g_{t,i}^2}$. More information is given on the cited paper, page 527. The update formula is:

$$x_{t+1} = \underset{x}{\operatorname{argmin}}\, f(x_t) + \nabla f(x_t)^T (x - x_t) + \frac{1}{2} \left\| Q_{1:t}^{1/2}(x - x_t) \right\|_2^2 + \Psi(x)$$
$$= \underset{x}{\operatorname{argmin}}\, g_t \cdot x + \frac{1}{2} \left\| Q_{1:t}^{1/2}(x - x_t) \right\|_2^2 + \Psi(x)$$

FTPRL we use [6]

The FTPRL algorithm we use in this report is described on a paper published by Google [6]. We use their update formula:

$$x_{t+1} = \underset{x}{\operatorname{argmin}}\, g_{1:t} \cdot x + \frac{1}{2} \sum_{s=1}^{t} \sigma_s \left\| x - x_t \right\|_2^2 + \lambda_1 x_1 \tag{5}$$

The apparition of the term $g_{1:t} = \sum_{s=1}^{t} g_s$ is explained on theorem 2 of paper [5].

5 Logistic regression implementation

For the project, a Python implementation[4] has been realized. The original structure of the code comes from another python code found on the *kaggle website*[5] This code implements the FTPRL mentioned above. Also, the implementation permits the algorithm to train on two types of inputs:

- On a vector of real number.
- And on a set of highly dimensional binary vector of length one (a vector with many 0 and a single 1).

Section 6 present how some of the initial features descried on section 2 have been modified to these highly dimensional binary vector.

5.1 Implementing the model

To learn the weights of our model we use the learning algorithm described previously: the FTPRL. It minimizes the negative log-likelihood function describing the performance of our model. FTPRL consider an example at a time, it's stochastic : "The derivative based on a randomly chosen single example is a random approximation to the true derivative based on all the training data" [3].

The precise description of the FTPRL algorithm is given on the Algorithm-1 of [6]. Our implementation directly follows it.

This is how the model works :

- Initialize the weights with a random distribution.
- Run Algorithm-1 of [6], but after 6k steps :
- Check every 1k steps on validation set if you perform at least 2% better than before. If not, stop.
- Compute the negative log-likelihood on the testing set.

Even though FTPRL is normalized, we use an early stopping method. This early stopping method permit on the one side to avoid over-fitting and, on the other side, it lowers the training time.

The validation and testing set mentioned just before are composed by 20% of the original data each. It leads to a training set of 60% of the original dataset.

After describing the model, we describe the possible input samples.

5.2 An input of real numbers

Here we describe the normal implementation of the logistic regression model seen on section 4.1. On that section we saw that the prediction was defined as:

$$\sigma(\beta, x) = \frac{1}{1 + e^{\beta^T x_i}}$$

[4] All the code is available at `https://github.com/marc-moreaux/text_classification`

[5] `https://www.kaggle.com/c/tradeshift-text-classification/forums/t/10537/beat-the-benchmark-with-less-than-400mb-of-memory`

Because our models predicts 33 values, β is now the weight matrix ($n \times 33$) applied to the vector x_i. In the code, β is a 2 dimension array of size $33 \times n$ and x is a 1 dimension array of size n. the prediction function is described on algorithm 1.

The update function takes into consideration the FTRL-proximal algorithm and is also described on the algorithm 1.

Algorithm 1: Prediction and Update for input vector of real number

Data: The function receives an input array of real values (x of size n), a weight array of binary values (w of size n × 33) and an integer label id (lbl).

Result: Prediction of the model given the inputs

```
1  begin
2      pred = 0;
3      for i in 0:n do
4          pred ← x[i] * w[lbl][i];
5      pred ← 1/(1+e^-pred);
6      return pred ;
```

Result: Update the weight of the model given the previously mentioned Data

```
7  begin
8      for i in 0:n do
9          dim_param[label][i] ← update;
10         w[label][i] ← w[label][i] - dim_param[label][i] * x[i] * f'(p,y) ;
```

5.3 An input of highly dimensional binary vector of length one

The second implementation of the algorithm is a twisted one allowing a highly dimensional binary vector as input. This vector have many 0 and one 1 (eg. a feature with value 1.23 could become 000000100). On the implementation, these binary vectors can have more than a million dimension. Due to that increase in parameters, the training algorithm 1 couldn't work efficiently as it became too slow.

As a solution to this problem, we used an other representation of the inputs. Instead of using the highly dimensional binary vector, we only saved the index of the '1' in a table. On the example seen before, the 1.23 became 000000100 and now is saved as 6, as 6 is the index of the '1' in the binary vector. The algorithm corresponding to this twist is presented on algorithm 2.

Algorithm 2: Prediction and Update for input vector of real number

Data: The function receives an input array of integer values (x of size n), a weight array of real values (w of size n × 33) and an integer label id (lbl).

Result: Prediction of the model given the inputs

```
1  begin
2      pred = 0;
3      for i in 0:n do
4          pred ← 1 * w[lbl][ x[i] ];
5      pred ← 1/(1+e^-pred);
6      return pred ;
```

Result: Update the weight of the model given the previously mentioned Data

```
7  begin
8      for i in 0:n do
9          dim_param[label][i] ← update;
10         w[label][i] ← w[label][i] - dim_param[label][i] * 1 * f'(p,y) ;
```

5.4 Feature selection

We also decided that we wouldn't train our model on all the available feature we have. Therefore we used a feature selector algorithm.

Our algorithm is a greedy Forward Sequential Selection (FSS). The FFS algorithm "selects a subset of features from the data matrix X that best predict the data y by sequentially selecting features until there is no improvement in prediction" [1]. This algorithm is described in paper [2].

In the literature, Backward Sequential Selection (BSS) is preferred to FSS. In our project, the reason for using FSS instead of BSS is related to the memory of our computer. If we were to use the BSS we would need to train our model on the set composed by all the binary vectors (and more). This would impose the computer to store a weight matrix of 10×33 million floating parameters. The computer used for testing didn't have that memory capacity.

We stop accepting new features after we have 15 of them or when accepting a new feature doesn't increase the accuracy of the model of more than 2%.

We call the model (or process) composed by the FSS and the FTPRL algorithm: FSSFTPRL.

6 Feature-engineering the input

As stated in the introduction, the dataset provided by *Tradeshift* doesn't come with a feature by feature descriptions. In section 2.1 we described what could a feature mean and in section 3 we give some hypothesis on these features. Given this few information on the dataset, we tried different input features on our model. At first, we tried using initial features preprocessing them, then we proposed a feature encoding, then a hash-equality feature and finally a feature reflecting the presence of a neighbor.

6.1 Data normalization

When working with logistic regression, it's preferable to normalize the inputs. Right below we present three standard data normalization techniques originating from the "Unsupervised Feature Learning and Deep Learning"[6] wiki tutorial provided by Stanford. We can use these deep learning techniques with our model as logistic regression is also in the family of neural networks. After applying one (or more) of these techniques, the training speed of the model should be increased.

The three data normalization techniques are called: simple rescaling, per-example mean-subtraction and feature standardization.

- "Simple rescaling" aims at having all the data dimension on the same scale. You wouldn't want to have an independent feature with values in range $[10^{-5},3*10^{-5}]$ and another independent feature in range $[-10^{6},10^{6}]$. Depending on the activation function, the data should be in the range [0,1] or [-1,1]. Therefore, you would multiply your initial inputs with a constant for them to be scaled in the desired range.
- "Per example mean-subtraction" consist of subtracting the mean value of a vector. For instance, if you want to examine the stability of a plane in the air and you have an altitude feature then you may reconsider your ground reference. You can instead consider a mean *plane in the air altitude* as reference to subtract the maen altitude to.
- "Feature standardization" consist in two steps: first, doing mean-subtraction and second, setting the variance to a unit variance. To do this, one needs to compute the mean and the variance of the data, then subtract the mean and divide by the variance each data points.

In the project we used the feature-standardization in two different ways. At first, we used a "classic method": normalizing separately all the features. At second, we tried to take advantage of the 5 blocks hypothesis seen back in section 3. To do so, we tested mean-subtraction and feature-standardization on every same kind of features. For instance: the input feature 1, 30, 59, 88 and 117 were merged together resulting in a shared mean value and a shared variance.

results[7]

[6] `http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial`

[7] All the results are available at `https://github.com/marc-moreaux/text_classification`

Algorithm	$x = 18$	$x = 17$
FTPRL[x]_no_norm	0.195626763424	0.204790168023
FTPRL[x]_classic_norm	0.199563944287	0.242193956451
FTPRL[x]_5_bloc_norm	0.199563944287	0.242193956451

Table 1: Model results given the normalization. [x] stands for the feature it has been training on.

step i =	1	2	3	4	5	6	7	8	9
New feature	[18]	[2]	[21]	[11]	[7]	[25]	[19]	[16]	[9]
FTPRL_feat	.1995	.1819	.161	.1509	.1414	.1342	.1293	.1248	.1224

Table 2: Result of feeding FSSFTPRL with all the initial normalized features. At step "i" the model choses a new feature to learn with (FSS)

Table 1 gives a performance comparison between some normalization. All of the 3 algorithms stoped after the 6k initial training samples. This means that after 6k samples, training on the next 1k sample didn't improve the validation result above 2%. Unexpectedly, using the feature standardization (*FTPRL[x]_classic_norm*), does not imply converging faster. On (*FTPRL[x]_no_norm*) we see that the algorithm converged faster without normalization. Also, the attempt to take advantage of the 5 blocks consideration didn't improved the convergence nor did it penalized it. Normalizing on 5 blocks (*FTPRL[x]_5_bloc_norm*) converges at the same speed than normalizing every different features (*FTPRL[x]_classic_norm*).

Even though it seems logical to use the data not normalized, we kept it normalized. Table 2 is the result of training our model with the normalized features. This training will be the baseline we will use to compare the future hand-engineered features.

It is obvious data normalization is not enough for achieving good results with our model. We now present hand-engineered features that might help our system.

6.2 One-hot feature encoding

The features 3 and 4 (of the 5 blocs), are hash-values. Using a linear encoding of these features didn't seem to make sens. We investigated in some techniques to take advantage of these features.

The first method we propose is an indexation represented in a one-hot format. Recall the initial dataset given by *Tradeshift* had 986'837 hash-values. We indexed these values by order of apparition and got an integer value for any of these hashes. The results presented on table 1 considered this integer as it was or normalized. Now, we propose to transform this integer value to a one-hot feature. We do the transformation for the hash-values, but also for every other features in order to take advantage, for instance, of a numeric coding.

One-hot encoding works as follow: Consider a feature vector $[1, 2, 3, 1]$. There is 3 values on the vector. Therefore every occurrences of 1 will be replaced by a 3 dimensions vector $[1, 0, 0]$. As we can see, there is a single "1" and many "0" on that new vector, this is the reason for calling this encoding a "one-hot encoder". That "1" will be the only value of the feature propagating energy

step i =	1	2	3
New feature	{17}	{4}	{3}
FTPRL_feat_1hot	0.0824	0.0722	0.0662

Table 3: Result of feeding FSSFTPRL with all the one-hotted features . At step "i" the model choses a new feature to learn with (FSS). {x} stands for one-hotted feature [x]

on the model. For floating values, this method removes the proximity relation between (for instance) 0.5 and 0.55 or in between 50 and 51 but brings closer in the model all the features with the exact same values.

To build each and every feature "one-hot encoded" we indexed the features. That index is the position of the "1" in the newly created one-hot-vector. The indexation is described on python code 3.

Algorithm 3: This algorithm show how the sample features are one-hot-encoded. 'X' is the initial dataset, from there, we consider each of the features x_{ij}. If this feature x_{ij} is to be encoded, we retrieve its index from a corresponding dictionary and store it in memory.

```
As prerequisite, we need an array telling which are the desired features to encode.
begin
    for x_i ∈ X do
        for x_ij ∈ x_i do
            (...)
            if x_ij is a feature to encode then
                tmp ← index of x_ij in its corresponding dictionary;
                store tmp in a array of indexes;
```

Result: The desired features x_{ij} have been one-hot-encoded. The index of the '1' is stored in an array of indexes

Lets make a concrete example and consider indexing feature 18. We consider the features 18, 47, 76, 105 and 134 (which correspond to feature 18 on each of the blocs) and index them. After summing the amount of elements in all the blocks corresponding to feature 18, we count 111 elements (111 indexes). To separate the five blocks one from another, we create $111 * 5 = 555$ neurons. If for one sample the feature 18, 47, 76, 105 and 134 equals the vector [0.123, 0.312, 0.123, 0.231, 0.312] then its indexed array may have the values [45,72,45,13,72] and we activate the neurons $\{0+45,\ 111+72,\ 222+45,\ 333+13$ and $444+72\}$

results

Using this technique drastically changes the results table 3 of our model. We see that using the one-hotted feature {17} directly beats the best score we had with *FTPRL_feat*. Sadly, the remote computer was switched off during the training so only 3 features were selected by FSS. Even though, it's enough to see how one-hotted features can beat our baseline.

6.3 Hash-equality feature

The second method we propose to take advantage of the hash-codes is a inner pairwise hash-equality feature. This method was considered because, when looking through the original CSV, we noticed that the hash-codes were often repeated inside the features.

It works as follows: out of the 145 features, 10 are hash-coded, therefore, the *binary pairwise hash-equality feature vector* (HEFV) is consisting of $C_{10}^2 = \frac{10!}{(10-2)!2!} = 45$ dimensions. The 45 dimensions represent the 45 possible pairs we can find on the 10 hash features. The HEFV will consist of zeros and ones. For instance, we considered the first dimension of HEFV to reflect the equality of feature 1 and 2, as a result, $\text{HEFV}_1 = 1$ if hash-feature 1 and 2 are identical or 0 if different.

This method is further explained on algorithm 4.

Algorithm 4: This algorithm show how to compute the *binary pairwise hash-equality feature vector* (HEFV). There is a subroutine ('update newFeature') described in the second part.

As prerequisite, we create an array 'newFeature' of size 45 filled with -1. It correspond to the HEFV. (i,j) refers to the index of the pair composed by i and j. An example mentioned earlier used the pair (1,2) corresponding to HEFV_1.

begin
  for $x_i \in X$ **do**
    for $x_{ij} \in x_i$ **do**
      (...)
      if x_{ij} *is a hash-feature* **then**
        hashIdx ← index of hashFeature ∈ [1...10];
        We then update 'newFeature' on the 9 times x_{ij} *appears:*
        for $i \in [1...hashIdx]$ **do**
          update newFeature[(i,hashIdx)] ;
        for $j \in [hashIdx + 1...10]$ **do**
          update newFeature[(hashIdx,j)] ;

Here is the logic beneath the 'update newFeature'

begin
  if *newFeature[(i,hashIdx)] = -1* **then**
    newFeature[(i,hashIdx)] ← x_{ij};
  else
    if *newFeature[(i,hashIdx)]* = x_{ij} **then**
      newFeature[(i,hashIdx)] ← 1;
    else
      newFeature[(i,hashIdx)] ← 0;

Result: The array 'newFeature' is filled with the values of HEFV.

We were also curious to know if a broader consideration of the pairs could have a positive impact on the training. We created a *binary pairwise feature-equality feature vector* (FEFV). The logic beneath that implementation is the same as the HEFV.

step i =	1	2	3	4	5
New feature	[18]	HEVF	[25]	[1]	[11]
FTPRL_feat_HEVF	.1995	.0965	.0883	.0851	.0837

Table 4: Result of feeding FSSFTPRL with all the initial normalized features and HEVF. At step "i" the model choses a new feature to learn with (FSS).

step i =	1	2	3
New feature	FEVF	[18]	[17]
FTPRL_feat_FEVF	.1053	.1003	.0846

Table 5: Result of feeding FSSFTPRL with all the initial normalized features and FEVF. At step "i" the model choses a new feature to learn with (FSS).

We didn't tried further considerations on the other tuples because of memory constrains. We though about looking for all the possible tuples in the 145 feature vector but this consideration would have resulted on a $\sum_{i=2}^{145} C_{145}^{i} > 4e^{43}$ feature vector.

results

The results achieved considering these two new features are presented on table 4 and table 5. Once again, the processes were shut-down. Still we see that both of these features improve the training. HEVF points out the importance of Hash equality in the training whereas FEVF points out that some equalities are important. A deeper look on the weights should reveal the pairs that bring the better performance. It's nice to notice that we can FEFV could be re-used in other dataset without knowledge on the features and might improve the learning.

6.4 Final result

All together FSSFTPRL produces the result visible on table 6. Once again, the training stopped before achieving the end. It's even though our best result. We see that FSSFTPRL selected one-hotted features and hash-equality feature. Funnily he performed better choosing the restricted pairs HEFV than FEFV. From this observation we can hypothesize on the necessity of realizing a FSS or BSS on the FEFV.

step i =	1	2	3	4
New feature	{17}	{4}	{3}	HEFV
FTPRL_all	0.0824	0.0722	0.0662	.0576

Table 6: Result of feeding FSSFTPRL with all features seen so far.

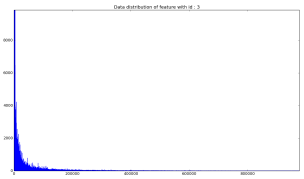
7 Conclusion

We've first tried to gain insight on the dataset to create a model accordingly. After realizing the whole dataset was black-boxed, we created a training process able to learn from two types of feature. The structure and implementation of FTPRL permits us to handle a very large number of one-hot encoded inputs but also, regular input features. Because FTPRL is low regret and converges much faster than gradient descent our process can use FSS to select the features (in a reasonable time frame).

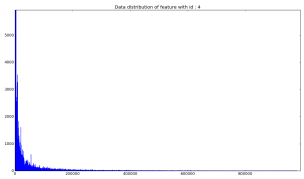
The features we created improve the training of our dataset and are re-applicable to other set of real-featured inputs. This generality makes our process re-applicable to other datasets.

We begun this report stating the desire to produce a method re-applicable to other dataset of unknown nature. I believe that this process answers the initially fixed objective. It is impossible that this process can feat each and every dataset of unknown nature but it can definitely help in the case you want to try a logistic regression on your dataset.

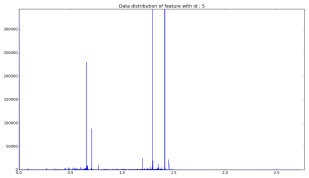
3



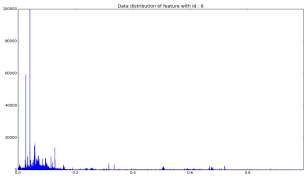
4



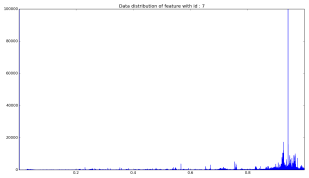
5



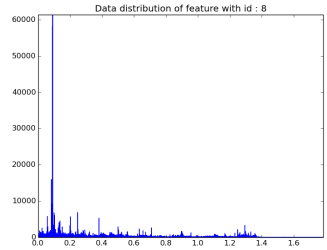
6



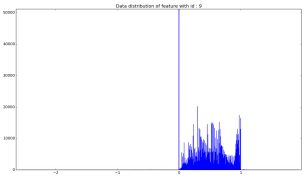
7



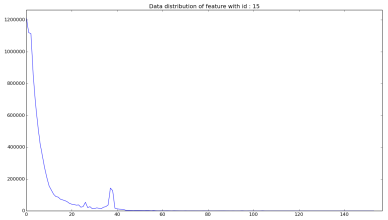
8



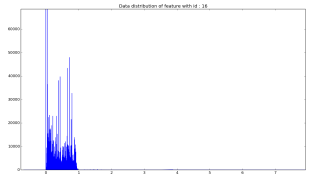
9



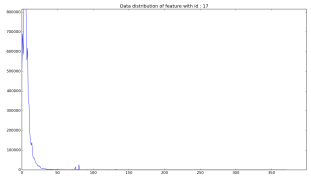
15



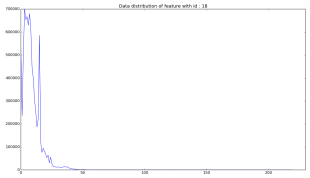
16



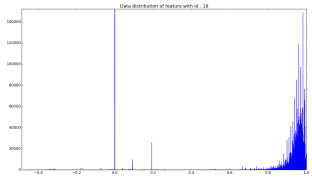
17



18



19

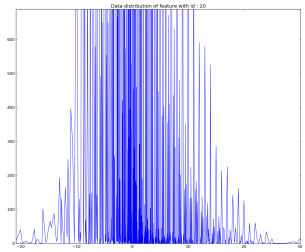


id	x1	x2	x3	x4	x5	x6	x7
1	NO	NO	dqOiM6y(...)8=	GNjrXXA(...)=	0.57656116338	0.073139435414	
x8	x9	x10	x11	x12	x13	x14	x15
0.11569717707	0.47247428917	YES	NO	NO	NO	NO	42
x16	x17	x18	x19	x20	x21	x22	x23
0.3960650128	3	6	0.99101796407	0	0.82	3306	4676
x24	x25	x26	x27	x28	x29	x30	x31
YES	NO	YES	0	0.40504704875	0.46460980036	NO	NO
x32	x33	x34	x35	x36	x37	x38	x39
NO	NO	mimucPm(...)=	s7mTY62(...)=	0.57656116338	0.073139435414	0.48139435414	0.11569717707
x40	x41	x42	x43	x44	x45	x46	x47
0.45856019358	YES	NO	YES	NO	NO	9	0.36826347305
x48	x49	x50	x51	x52	x53	x54	x55
2	10	0.99272882805	0	0.94	3306	4676	YES

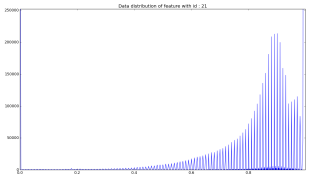
Table 7: Extract of 50 (out of 145) features of sample with ID 1 given by *Tradeshift*

x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15
x16	x17	x18	x19	x20	x21	x22	x23	x24	x25	x26	x27	x28	x29	
NO	NO	dqOiM6(...)=	GNjrXX(...)=	0.57656	0.07313	0.48139	0.11569	0.47247	YES	NO	NO	NO	NO	42
0.39606	3	6	0.99101	0	0.82	3306	4676	YES	NO	YES	0	0.40504	0.464609	
x32	x33	x34	x35	x36	x37	x38	x39	x40	x41	x42	x43	x44	x45	x46
x47	x48	x49	x50	x51	x52	x53	x54	x55	x56	x57	x58	x59	x60	
NO	NO	mimucP(...)=	s7mTY6(...)=	0.57656	0.07313	0.48139	0.11569	0.45856	YES	NO	YES	NO	NO	9
0.36826	2	10	0.99272	0	0.94	3306	4676	YES	NO	YES	1	0.3755	0.451300	
x62	x63	x64	x65	x66	x67	x68	x69	x70	x71	x72	x73	x74	x75	x76
x77	x78	x79	x80	x81	x82	x83	x84	x85	x86	x87	x88	x89	x90	
NO	NO	Op+X3a(...)=	GeerC2(...)=	0.57656	0.07313	0.48139	0.11569	0.48759	YES	NO	NO	NO	NO	42
0.36313	6	10	0.98759	0	0.71	3306	4676	YES	NO	YES	0	0.3755	0.479733	
x92	x93	x94	x95	x96	x97	x98	x99	x100	x101	x102	x103	x104	x105	x106
x107	x108	x109	x110	x111	x112	x113	x114	x115	x116	x117	x118	x119	x120	
NO	NO	+dia7t(...)=	f4Uu1R(...)=	0.57656	0.07313	0.48139	0.11569	0.47307	YES	NO	NO	NO	NO	37
0.33361	4	6	0.98716	0	0.89	3306	4676	YES	NO	YES	1	0.34644	0.464609	
x30	x31	x61	x91	x121	x122	x123	x124	x125	x126	x127	x128	x129	x130	x131
x132	x133	x134	x135	x136	x137	x138	x139	x140	x141	x142	x143	x144	x145	
NO	NO	+2TNtX(...)=	bxU52t(...)=	0.57656	0.07313	0.48139	0.11569	0.47307	YES	NO	NO	NO	NO	42
0.36313	5	6	0.98759	0	0.81	3306	4676	YES	NO	YES	2	0.3755	0.464609	

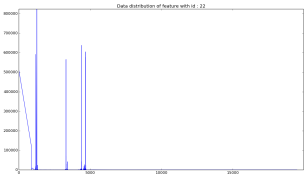
Table 8: First sample in a five bloc format



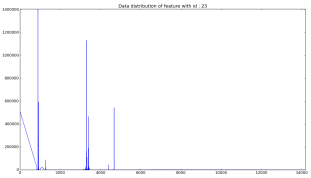
20



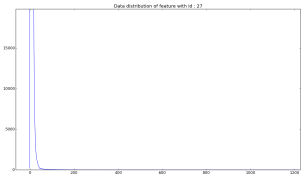
21



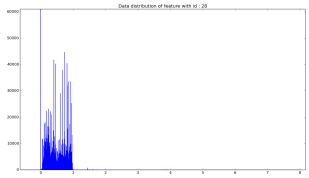
22



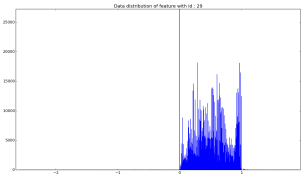
23



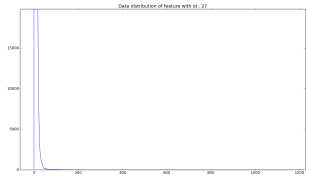
27



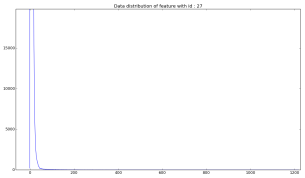
28



29



27



27

References

[1]

[2] David W Aha and Richard L Bankert. A comparative evaluation of sequential feature selection algorithms. In *Learning from Data*, pages 199–206. Springer, 1996.

[3] Charles Elkan. Maximum likelihood, logistic regression, and stochastic gradient training, 2013.

[4] David W Hosmer Jr and Stanley Lemeshow. *Applied logistic regression*. John Wiley & Sons, 2004.

[5] H Brendan McMahan. Follow-the-regularized-leader and mirror descent: Equivalence theorems and l1 regularization. In *International Conference on Artificial Intelligence and Statistics*, pages 525–533, 2011.

[6] H Brendan McMahan, Gary Holt, David Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, et al. Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2013.