Logistic Regression for Sentiment Analysis on Large Scale Social Media Posts via Apache Spark

Peng Shi
University of Waterloo
peng.shi@uwaterloo.ca

Wei Yang
University of Waterloo
w85yang@uwaterloo.ca

Masijia Qiu
University of Waterloo
m23qiu@uwaterloo.ca

ABSTRACT

The project presents simple distributed implementations of logistic regression (LR) with several variants, including Gradient Descent (GD) based, Stochastic Gradient Descent (SGD) based and Mini-batch Stochastic Gradient Descent (MBSGD) based ones. These implementations are tested on the sentiment analysis tasks, where detailed hyper-parameters analysis is provided.

KEYWORDS

Logistics regression, gradient decent, sentiment analysis

ACM Reference Format:
Peng Shi, Wei Yang, and Masijia Qiu. 2018. Logistic Regression for Sentiment Analysis on Large Scale Social Media Posts via Apache Spark. In *Proceedings of Course Project (CS 651),* Jimmy Lin (Ed.). ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Logistic Regression (LR) is a widespread and useful machine learning algorithm in academia and industrial for its simplicity and effectiveness. However, as the data is inflating rapidly, these machine learning algorithms and tools are needed to be adapted to fit the big-data era. MapReduce [8] is a popular algorithm in distributed computing area while Hadoop [19] and Spark [23] are two open-source framework for large-scale data processing.

Today, big data community and machine learning community have sharply developed and several machine learning frameworks for distributed computing have emerged, including MLLib, Mahout and RHadoop [22]. However, for newbie to this area, these mature but complicated machine learning libraries are hard due to the compact code structure and they have no idea of how to "get the hand dirty".

In this paper, we provide simple but comprehensive implementations for logistic regression and its variants including different optimization methods and regularization. These implementations provide the beginners a good chance to learn the basic concepts of logistic regression and its spark implementation. Furthermore, they can learn how to modify these simple code to enhance the implementation and even provide more functions.

In this paper, we have following contributions:

We offer several simple implementations for logistic regression and its variants. These implementations could be a guidance of distributed computing and machine learning.

We create a large tweets sentiment dataset with four-byte hashing feature, which can be used for further experiments.

We provide rigorous hyper-parameter analysis for three variants of logistic regression implementations.

2 BACKGROUND AND RELATED WORK

Let's start with an overview of machine learning. Given X to be the input space while Y the output space, the set of training samples $D = \{(x_1, y_1), (x_2, y_2)...(x_n, y_n)\}$ from the X x Y space, naming the labeled examples. Usually, x_i represents a featured vector where $x_i \in \mathbf{R}^d$. In supervised classification task, y_i comes from a finite set. when in the binary classification, $y \in \{-1, +1\}$. A function $f \colon X \Rightarrow Y$ describing the data characteristics is introduced in the supervised machine learning tasks. The optimized case will minimize the "loss" function L that quantifiably measures the discrepancy between predicted $f(x_i)$ and the actual result y_i. Minimizing the quantity $\sum_{(x_i, y_i) \in D} L(f(x_i), y_i)$, the best f in the learned model is selected from a hypothesis space. Then it can be employed on previously unknown data to make predictions or offer predictive analysis. [5] [9] Dealing with a two-class problem as the tweets sentiment, we apply the binary logistic regression to assign observations onto two classes.

Three components are of the most significance in the machine learning solutions: the data, the features extracted from the data, and the model. Among them, the size of the dataset is dominant given the accumulated real-world experience over the last decades. [10] [13] Simple models on massive data perform better than sophisticated modes on small data. [2] [4]

The traditional machine learning assumed sequential algorithms on data fit in memory, which is no more realistic in the information bang era. Multi-core [6] and cluster-based solutions [1] offer new opportunities. Techniques occurs for example learning decision trees and the ensembles [20], MaxEnt models [16], structured perceptrons [15] and so on. These approaches work well when 'data is king' for their ability to process massive amount of data. Despite the gaining popularization of large-scale learning, few published studies focus on machine learning *workflows* and how such tools integrate with data management platforms. Google detects adversarial advertisements on Sculley et al. [18] Facebook builds its data platform on Hive. [21] Cohen et al. applies the integration of predictive analysis into traditional RDBMSes [7].

3 BASE ALGORITHM

3.1 Logistic Regression

Logistic regression [11] transforms its output using the logistic sigmoid function to return a probability value which can then be mapped to two or more discrete classes. Assuming a two-class problem with a training set $D = \{(x_1, y_1), (x_2, y_2)...(x_n, y_n)\}$, where x_i are feature vectors and $y_i \in \{-1, +1\}$, a class of linear discriminative functions can be defined of the form:

$$F(x) : \mathbf{R}^N \to \{-1, +1\} \tag{1}$$

$$F(x) = \begin{cases} +1 & if\ w \cdot x \geq t \\ -1 & if\ w \cdot x < t \end{cases} \tag{2}$$

where t represents a decision threshold and w is the weight vector. The modeling process usually optimizes the weight vector based on the training data D, and the decision threshold is subsequently tuned by various operational constraints. To learn w, lots of methods have been proposed to optimize different forms of loss functions or objective functions defined over the training data, where logistic regression is one particularly well-established technique interpreting the linear function $w \cdot x$ as the logarithmic odds of x belonging to class +1 over -1, i.e.,

$$log[\frac{p(y = +1|x)}{p(y = -1|x)}] = w \cdot x \tag{3}$$

The objective of regularized logistic regression (using Gaussian smoothing) is to identify the parameter vector w that maximizes the conditional posterior of the data.

$$L = exp(-\lambda w^2/2) \cdot \prod_i p(y_i|x_i) \tag{4}$$

where

$$p(y = +1|x) = \frac{1}{(1 + exp(-w \cdot x))} \tag{5}$$

and

$$\begin{aligned} p(y = -1|x) &= 1 - p(y = +1|x) \\ &= \frac{1}{(1 + exp(w \cdot x))}. \end{aligned} \tag{6}$$

In this project, three type of the variants are applied accordingly: Gradient Descent (GD) based, Stochastic Gradient Descent (SGD) based and Mini-batch Stochastic Gradient Descent (MBSGD) based.

3.2 Gradient Descent

Recall the basic Gradient Descent method, it is accomplished by adjusting the weight vector in the direction opposite to the gradient of log(L)ï¼š

$$\begin{aligned} -\nabla\, log(L) &= \lambda w + \sum_i \frac{1}{p(y_i|x_i)} \frac{\vartheta}{\vartheta w} p(y_i|x_i) \\ &= \lambda w + \sum_i y_i p(y_i|x_i)(1 - p(y_i|x_i)) \end{aligned} \tag{7}$$

3.3 Stochastic Gradient Descent

While in GD the whole training set is considered before taking one model parameters update step, in SGD only one data point is considered for each model parameters update step, cycling over the Training Set. [3] In SGD update, the gradient is computed based on a single training instance, the update to the weight vector upon seeing the ith training example is given by

$$w \leftarrow w + \eta \cdot [-\lambda w + y_i \cdot p(y_i|x_i)(1 - p(y_i|x_i))] \tag{8}$$

Noted that each element in the weight vector is decayed at each iteration. However, when the feature vectors of training instances are very sparse (as is true for our project), we can simply delay the updates for features until they are actually seen.

3.4 Mini-Batch Stochastic Gradient Descent

Mini-Batch Stochastic Gradient Descent (MBSGD) [12] is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model coefficients. It chooses to sum the gradient over the mini-batch or take the average of the gradient which further reduces the variance of the gradient. Mini-batch gradient descent seeks to find a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. It is the most common implementation of gradient descent used in the field of deep learning.

3.5 Regularization

Regularization is a concrete method for add a *"penalty term"* to the optimization problem, such that more complex models includes a larger penalty. For the case of linear regression, the new optimization problem is to minimize

$$MSE(w) + penalty(w),$$

where penalty w is increasing with the 'complexity' of w. Therefore, a complex solution can be chosen over simple one only if it leads to a big decrease in the mean-squared error.

Many methods define the penalty term like *ridge regression*, *L2 regularization* and *Tikhonov regularization*. In this project, we'll consider the *L2 regularization.*

In this method, we define

$$penalty(w) = \lambda \cdot ||w||_2^2$$

where λ is a positive *'hyperparameter'*, a knob that allows you to trade-off smaller MSE. For more details, we refer the reader to the authoritative paper [17].

4 DISTRIBUTED GRADIENT DECENT VIA APACHE SPARK

In this section, we describe our implementation details of three variants of gradient decent algorithm under the distributed setup.

With reading feature vectors from text file, we push these text lines to mapper to parse and generate training and testing instances which is composed of document id (docid), label and feature vector. These instances are represented as a RDD and used for further processing.

We describe three variants for the training processes including Gradient Descent (GD), Stochastic Gradient Descent (SGD) and Mini-Batch Stochastic Gradient Descent (MBSGD).

4.1 Gradient Descent

With GD, the basic idea is to iterate all the training data and compute the averaged gradient based on global view (all training data). In general, the method to compute the averaged gradient is directly sum up all the gradient that is based on each instance and divided by the number of instances. However, because of the sparsity of the feature vectors (most of the slots of the feature vectors are 0 and only small fraction of them are 1), the denominator will have negative effects on the magnitude of the gradient. More specifically, those feature only appear once or twice will be greatly affected because these values are divided by a large number (the total number of the instances. To tackle this problem, we compute the frequency of occurrence for each feature and use these frequencies as the regulators for the gradient computed in each iteration.

Algorithm 1 GD

```
    procedure GradientDescent
        input ← Read from file
        inputFeatures ← input.map(
                process input line and
                generate instance with
                docid, label, and features)
        globalWeight ← Map[Int, Double]
        FeatureCounter ←
              Count Feature Frequency of Occurrence
    top:
        w ← broadcast(globalWeight)
        inputFeatures.mapPartition(
         $\Delta w$ ←
              compute gradient based on
                    this partition and sum them up
    ).collect
        globalWeight ← sum over parital weights
        goto top.
```

In each iteration, the global weight is broadcast to all mappers. Here the *mapPatition* is used instead of *map*, because in the *mapPartion*, the partially sum of the gradient can be computed and the global sum of the gradient can be computed on single reducer and update the global weight. This decision can greatly lesson the load for the single reducer.

Algorithm 1 shows the details of the implementation.

4.2 Stochastic Gradient Descent

For Stochastic Gradient Descent (SGD), we add "dumpy key" for each instances to ensure all the training instances are collected to the reducer via *groupByKey*. After receiving all the training instance, the weight is updated per instance.

Algorithm 2 shows more details.

Algorithm 2 SGD

```
    procedure MiniBatchSGD
        input ← Read from file
        inputFeatures ← input.map(
                process input line and
                generate instance with
                docid, label, and features)
        globalWeight ← Map[Int, Double]
    top:
        w ← broadcast(globalWeight)
        samples ← inputFeatures.sample(fraction)
        FeatureCounter ←
              Count Feature Frequency of Occurrence
         $\Delta w$ ←
              compute gradient based on
                    this sample and sum them up
                    and regularized
        globalWeight ← sum over parital weights
        goto top.
```

4.3 Mini-Batch Stochastic Gradient Descent

The Mini-Batch Stochastic Gradient Descent (MBSGD) is the middle ground for GD and SGD. For the implementation, we use *sample* method to generate a small batch of training instances and apply the same method as the GD in the following steps. More specifically, the averaged gradient is computed based on the small batch training instances and apply the update rules to the global weights. We need to notice that the denominator for each feature is counted based on this small batch size.

Algorithm 3 shows implementation details.

Algorithm 3 MBSGD

```
    procedure StochasticGradientDescent
        input ← Read from file
        inputFeatures ← input.map(
                process input line and
                generate instance with
                docid, label, and features)
        globalWeight ← Map[Int, Double]
    top:
        for each instance
              $\Delta w$ ←
                   compute gradient based on
                    this instance
              globalWeight ← update with $\Delta w$
        goto top.
```

4.4 Regularization

We apply the *L*2 regularization during the training according to the update rules of the Apache Spark version.

5 EXPERIMENT

In this section, we experiment our logistic regression implementations on Tweets Sentiment Analysis task. More specifically, the task

Table 1: Dataset Statistics

	TRAIN_ALL	TEST_ALL	TRAIN_SMALL	TEST_SMALL
File Size	25 GB	6.1 GB	361 MB	73 MB
# of Positive Tweets	33,870,264	8,467,134	500,206	99,918
# of Negative Tweets	33,869,640	8,467,758	499,794	100,082

is a binary polarity classification task. That is, given a tweet, the classifier is to predict the sentiment of the tweet y_i which belongs to $\{Negative, Positive\}$. We follow the [14] to use the popular "emoticon trick" to generate the training and testing data. The intuition behind this is that tweets with positive emoticons or emoji, e.g., :-) and variants, express positive sentiment, and those with negative emoticons or emoji, e.g., :-(and variants express negative sentiment. However, we can not ensure that the data is in high quality because people can choose to use some positive emoji to express some negative feelings, e.g., irony. Nevertheless, in practice, "emoticon trick" is a good mechanism for generating large training instances.

5.1 Dataset Building

We generate training and testing data in following procedures: first, we extract the tweets from February 2013 to April 2018 whose language fields are English and classify them into two categories regarding their sentiment, which is determined by "emoticon trick". More specifically, if the tweet contains a emoji or emoticons which is in our positive list, then it will be put into positive pool, and in similar way we can generate negative pool. Those tweets tweets which do not contain any emoticons or emoji, or only contain those emoticons and emoji which are not appearing in our list, will not be considered in our experiments. Following Lin and Kolcz [14], we remove all preprocessed tweets with less than 20 characters. Secondly, we regard each tweet as a byte array and move a four-byte sliding window, with stride equals to one, along the array, and hash the contents of the bytes. The hashed value is taken as the feature id.

After preprocessing, the data is split into training and test set by a ratio of 8:2. We truncate the tweets and try to ensure that the numbers of positive and negative samples in both training and test set are almost equal to avoid the skewed label distribution problem. The statistics of the clean dataset can be viewed in Table 1.

5.2 Results

The best performance of three variants of gradient decent is shown in Table 2. For simplicity, we only report results from one setting (e.g. epoch, learning rate, batch size). More details analysis on effect of hyper-parameter will be discussed in 5.3. From Table 2 we can see that MBSGD performs the best while GD gets the lowest accuracy among them in *TEST_SMALL* setting. We argue that this is because MBSGD updates the weight parameter for each mini batch, which combines the advantage of both SGD and GD: it cares about both the global trend of the gradient but also pay attention to some tricky sample clusters. Under *TEST_ALL* setting, we only can successfully experiment on MBSGD and encounter *OutOfMemory* error under other two implementations. There are several reasons behind this.

Table 2: Results

	TEST_ALL	TEST_SMALL
GD	OOM	0.7268
SGD	OOM	0.7457
MBSGD	0.7529 (f = 0.01) / OOM (f = 0.1)	0.7496

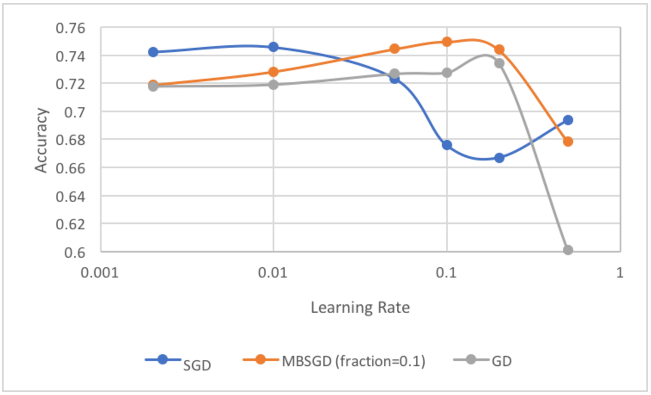

Figure 1: Parameter Analysis on Learning Rate

For example single machine might not have enough resource to host the training process under SGD setting.

5.3 Parameter Analysis

The parameter analysis of learning rate η is shown in Figure 1. We compare three GD variants with different learning rates ($\eta = 0.002, 0.01, 0.05$) on the small version of dataset. Other parameters are kept fixed ($\lambda = 0$, $N = 3$). From the result we can see that tuning the learning rate does help achieve better performance and different GD variants has different optimal learning rates.

The parameter analysis of regularization coefficient λ is shown in Figure 2. We run SGD with different regularization coefficients ($\lambda = 0.00001, 0.0001, 0.001, 0.01, 0.05, 0.5$) on the small version of dataset. Other parameters are kept fixed (η=0, N=1). From the result we can see that tuning the regularization coefficient helps little for the final result and large λ will harm the performance.

The parameter analysis of epoch N is shown in Figure 2. We compare three GD variants with different numbers of epochs ($N = 1, 2, 3, 4$) on the small version of dataset. Other parameters are kept fixed (η=0, λ=0.002). From the result we can see that after one epoch, on the other words, after the model goes through the whole dataset for one pass, the performance tends to converge.

The parameter analysis of batch fraction f is shown in Figure 4. We run MBSGD with different fractions ($f = 0.001, 0.01, 0.1, 1$)

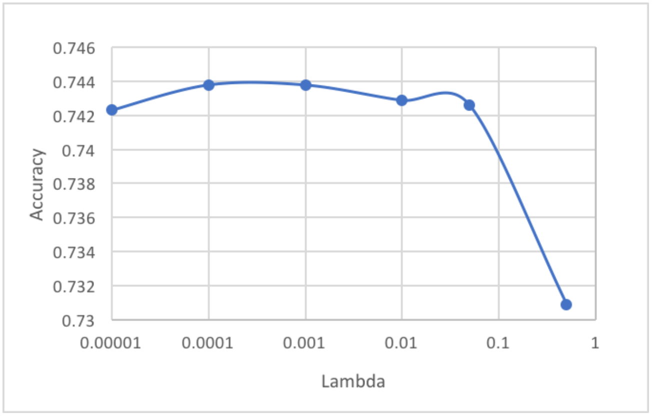

Figure 2: Parameter Analysis on Regularization Coefficient

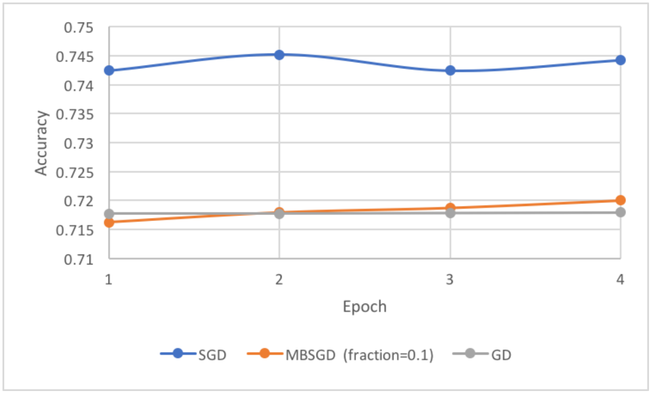

Figure 3: Parameter Analysis on Epoch

on the small version of dataset. Other parameters are kept fixed ($\eta = 0$, $\lambda = 0.002$, $N = 3$). From the result we can see that smaller fraction will bring better performance. Note that when $f = 1$, all tweets are sampled for each iteration, which means now MBSGD is essentially the same with GD. Under large fraction setting, the algorithm uses less iterations in one epoch and to achieve same accuracy, epoch number N needs to be larger.

6 CONCLUSION AND FUTURE WORK

In this project, we summarize our contributions in three points:

(1) We collect and clean the English stream of tweets text from February 2013 to April 2018 and create the training and test dataset for sentiment analysis with the label infromation from emojis and emoticons.
(2) We extract the four byte character feature for each tweet and implement the logistics regression with GD, SGD amd mini-batch SGD for the sentiment analysis on the large scale social media data.
(3) We provide the parameter analysis for the three variants of gradient decent on a subset of the whole data.

In the future, we plan to implement more fancy optimization tricks such as momentum, nesterov, learning rate decay and so on.

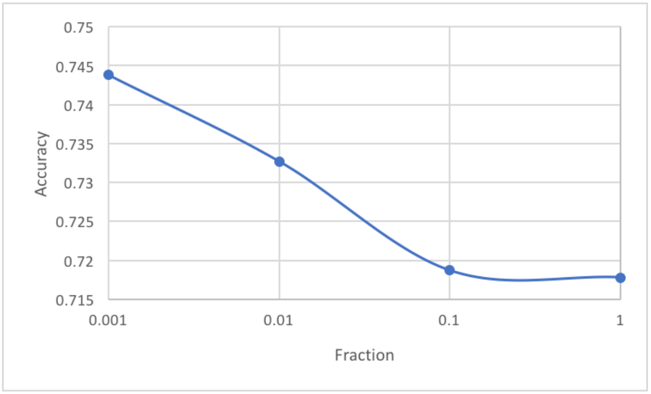

Figure 4: Parameter Analysis on Batch Fraction

We are also interested in comparing the time efficiency of different variants of gradient decent under the setup of large-scale training data.

REFERENCES

[1] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. 2014. A reliable effective terascale linear learning system. *The Journal of Machine Learning Research* 15, 1 (2014), 1111–1133.
[2] Michele Banko and Eric Brill. 2001. Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 26–33.
[3] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer, 177–186.
[4] Thorsten Brants, Ashok C Popat, Peng Xu, Franz J Och, and Jeffrey Dean. 2007. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*.
[5] M Bishop Christopher. 2016. *PATTERN RECOGNITION AND MACHINE LEARNING*. Springer-Verlag New York.
[6] Cheng-Tao Chu, Sang K Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Kunle Olukotun, and Andrew Y Ng. 2007. Map-reduce for machine learning on multicore. In *Advances in neural information processing systems*. 281–288.
[7] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M Hellerstein, and Caleb Welton. 2009. MAD skills: new analysis practices for big data. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1481–1492.
[8] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[9] James Franklin. 2005. The elements of statistical learning: data mining, inference and prediction. *The Mathematical Intelligencer* 27, 2 (2005), 83–85.
[10] Alon Halevy, Peter Norvig, and Fernando Pereira. 2009. The unreasonable effectiveness of data. *IEEE Intelligent Systems* 24, 2 (2009), 8–12.
[11] Frank E Harrell. 2001. Ordinal logistic regression. In *Regression modeling strategies*. Springer, 331–343.
[12] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).
[13] Jimmy Lin and Chris Dyer. 2010. Data-intensive text processing with MapReduce. *Synthesis Lectures on Human Language Technologies* 3, 1 (2010), 1–177.
[14] Jimmy Lin and Alek Kolcz. 2012. Large-scale machine learning at twitter. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 793–804.
[15] Ryan McDonald, Keith Hall, and Gideon Mann. 2010. Distributed training strategies for the structured perceptron. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, 456–464.
[16] Ryan Mcdonald, Mehryar Mohri, Nathan Silberman, Dan Walker, and Gideon S Mann. 2009. Efficient large-scale distributed training of conditional maximum entropy models. In *Advances in Neural Information Processing Systems*. 1231–1239.
[17] Nasser M Nasrabadi. 2007. Pattern recognition and machine learning. *Journal of electronic imaging* 16, 4 (2007), 049901.

[18] D Sculley, Matthew Eric Otey, Michael Pohl, Bridget Spitznagel, John Hainsworth, and Yunkai Zhou. 2011. Detecting adversarial advertisements in the wild. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 274–282.

[19] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee, 1–10.

[20] Krysta M Svore and CJ Burges. 2011. Large-scale learning to rank using boosted decision trees. *Scaling Up Machine Learning: Parallel and Distributed Approaches* 2 (2011).

[21] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. 2010. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE, 996–1005.

[22] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

[23] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.