An Ensemble Method for Large Scale Machine Learning with Hadoop MapReduce

by

Xuan Liu

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the degree of Masters of Applied Science in
Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
School of Electrical Engineering and Computer Science
University of Ottawa

Table of Content

List of Figures

List of Tables

List of Abbreviations

API:	Application Program Interface
CUDA:	Unified Device Architecture
DSL:	Domain Specific Language
EM:	Expectation Maximization
GFS:	Google File System
GPU:	Graphics Processing Units
HDFS:	Hadoop Distributed File System
HPC:	High Performance Computing
ML:	Machine Learning
MLR:	Multi-response Linear Regression
MPI:	Message Passing Interface
NFL:	No Free Lunch
PML:	Parallelized-Meta-Learning
POSIX:	Portable Operating System Interface
PSP:	Protein Structure Prediction
RDBMS:	Relational Database Management System
RDDs:	Resilient Distributed Datasets
SMP:	Symmetric Multiprocessing

Abstract

We propose a new ensemble algorithm: the meta-boosting algorithm. This algorithm enables the original Adaboost algorithm to improve the decisions made by different WeakLearners utilizing the meta-learning approach. Better accuracy results are achieved since this algorithm reduces both bias and variance. However, higher accuracy also brings higher computational complexity, especially on big data. We then propose the parallelized meta-boosting algorithm: Parallelized-Meta-Learning (PML) using the MapReduce programming paradigm on Hadoop. The experimental results on the Amazon EC2 cloud computing infrastructure show that PML reduces the computation complexity enormously while retaining lower error rates than the results on a single computer. As we know MapReduce has its inherent weakness that it cannot directly support iterations in an algorithm, our approach is a win-win method, since it not only overcomes this weakness, but also secures good accuracy performance. The comparison between this approach and a contemporary algorithm AdaBoost.PL is also performed.

Acknowledgements

It is my pleasure to take this opportunity to thank all the people who have helped me over the past two years.

I would like to thank my supervisors: Prof. Nathalie Japkowicz and Prof. Stan Matwin who have given me the opportunity to be a member of the TAMALE group at the University of Ottawa. Their passion in research and the critical thinking guided me through my master's study. This thesis would be impossible without their constant support and discussions.

Finally, I must thank my family. Their encouragement, understanding and constant love made this work possible.

Chapter 1

Introduction

The supervised learning problem can be described as: given a set of training examples $\{(X_1,\ y_1), \ldots (X_n,\ y_n)\}$, a learning program will be trained on these training examples to generate a function $y = h(X)$. Here X_i represents a vector of features, y_i is its label and, in the case of classification, it belongs to a discrete set of classes $\{1, \ldots, K\}$. h represents the hypothesis that estimates the true function $\emptyset$ which is unknown. It is also called a classifier. In the test process, a new example without label X' will be tested on the generated function h to obtain its predicted label y'.

In this scenario, ensemble learning is interpreted as combining the individual decisions of an ensemble of classifiers to classify new examples. It is also known as a multiple classifier system, a committee of classifiers, or a mixture of experts.

Although ensemble-based systems have been put to practice only recently by the computational intelligence community, ensemble-based decision making has been part of our daily lives perhaps as long as the civilized communities existed. In matters of making important decisions about financial, medical, social, or other implications, we often seek many opinions and combine them through some thought process to reach a final decision which is regarded as the most reasonable and wisest one. There are many such examples: consulting several doctors before undergoing major surgery; reading user reviews before

purchasing a product; talking to referees before hiring a potential job applicant; voting to elect a candidate official or decide on a new law; sentencing based on a jury of peers or a panel of judges; even peer reviewing of this thesis.

On the other hand, we have entered the big data age. Here is one proof: the size of the "digital universe" was estimated to be 0.18 zettabytes in 2006 and 1.8 zettabytes in 2011 [1]. A zettabyte is 2^{70} bytes.It is becoming more and more important to organize and utilize the massive amounts of data currently being generated. Therefore, developing methods to store and analyze these new data has become a very urgent task. One common and popular approach to big data is the cloud computing environment, and its most popular open-source implementation: Hadoop [12]. At the same time, extracting knowledge from massive data sets has attracted tremendous interest in the data mining community.

This thesis develops a new ensemble classification method and adapts it to the big data age. The reminder of this chapter gives the motivations and contributions of the thesis. Section 1.1 presents our motivation for doing this research, while section 1.2 and 1.3provide this thesis's contribution and outline, respectively.

1.1 Motivation

One of the main pursuits when designing machine learning algorithms is to gain high accuracy and one of the methods to achieve this goal is to use ensemble learning. Ensemble learning is designed to improve the accuracy of an automated decision-making

system. It has been demonstrated in numerous empirical and theoretical studies that ensemble models often attain higher accuracy than its individual models. This does not necessarily mean that ensemble systems always beat the performance of the best individual classifiers in the ensemble. However, the risk of choosing a poorly behaved classifier is certainly reduced.

Studying methods to construct good ensembles of classifiers has been one of the most active and popular areas of research. To construct an ensemble system: first, a set of base classifiers needs to be generated; second, a strategy is needed to combine these base classifiers. When building base classifiers, we need to select a strategy to make them as diverse as possible. When combining them, a preferred strategy aims to amplify correct decisions and cancel incorrect ones.

The significant development of Machine Learning (ML) in the last few decades has been making this field increasingly applicable to the real world. These applications include business, social media analytics, web-search, computational advertising, recommender systems, mobile sensors, genomic sequencing and astronomy. For instance, at supermarkets, promotions can be recommended to customers through their historic purchase record; for credit card companies, anomaly use of credit cards could be quickly identified by analyzing billions of transactions; for insurance companies, possible frauds could be flagged for claims.

Subsequently, an explosion of abundant data has been generated in these applications due to the following reasons: the potential value of data-driven approaches to optimize every aspect of an organization’s operation has been recognized; the utilization

of data from hypothesis formulation to theory validation in scientific disciplines is ubiquitous; the rapid growth of the World Wide Web and the enormous amount of data it has produced; the increasing capabilities for data collection. For example, Facebook has more than one billion people active as per Oct 4, 2012, which has increased by about 155 million users as compared to Nov 6, 2011[1]; until May, 2011, YouTube had more than 48 hours of video uploaded to its site every minute, which is over twice the figures of 2010[2].

These enormous amounts of data are often out of the capacity of individual disks and too massive to be processed with a single CPU. Therefore, there is a growing need for scalable systems which enable modern ML algorithms to process large datasets. This will also make ML algorithms more applicable to real world applications. One of the options is to use low level programming frameworks such as OpenMP [2] or MPI [3] to implement ML algorithms. However, this task can be very challenging as the programmer has to address complex issues such as communication, synchronization, race conditions and deadlocks. Some previous attempts for scalable ML algorithms were realized on specialized hardware/parallel architectures through hand-tuned implementations [4] or by parallelizing individual learning algorithms on a cluster of machines [5] [6] [7].

In very recent years, a parallelization framework was provided by Google's MapReduce framework [8] and was built on top of the distributed Google File System (GFS) [9]. This framework has the advantages of ease of use, scalability and fault-tolerance. MapReduce is a generic parallel programming model, which has become very popular and made it possible to implement scalable Machine Learning algorithms

[1] https://newsroom.fb.com/Timeline

[2] http://youtube-global.blogspot.ca/2011/05/thanks-youtube-community-for-two-big.html

easily. The implementations are accomplished on a variety of MapReduce architectures [10] [11] [12] and includes multicore [13], proprietary [14] [15] [16] and open source [17] architectures.

As an open-source attempt to reproduce Google's implementation, the Hadoop project [12] was developed following the success at Google. This project is hosted as a sub-project of the Apache Software Foundation's Lucene search engine library. Corresponding to GFS which is used by Google's large scale cluster infrastructure, Hadoop also has its own distributed file system known as the Hadoop Distributed File System (HDFS).

The Map-Reduce paradigm and its implementations are therefore an interesting environment in which to perform ensemble-based Machine Learning. In addition to generating more accurate decision-making systems, ensemble systems are also preferred when the data volume is too large to be handled by a single classifier. This is realized by letting each classifier process a partition of the data and then combining their results. Our thesis is motivated by both of these aspects of ensemble learning: the generation of more accurate ensemble learning models and the handling of enormous amount of data. Even for medium sized data, the approach of parallelizing ensemble learning would be beneficial. The reason is that ensemble learning increases the classification accuracy at the expense of increased complexity.

1.2 Thesis Contributions

We propose a new ensemble learning algorithm: meta-boosting. Using the boosting method, a weak learner can be converted into a strong learner by changing the weight distribution of the training examples. It is often regarded as a method for decreasing both the bias (the accuracy of this classifier) and variance (the precision of the classifier when trained on different training sets) although it mainly reduces variance. Meta-learning has the advantage of coalescing the results of multiple learners to improve accuracy, which is a bias reduction method. By combining boosting algorithms with different weak learners using the meta-learning scheme, both of the bias and variance are reduced. Moreover, this configuration is particularly conducive for parallelizing general machine learning algorithms. Our experiments demonstrate that this meta-boosting algorithm not only displays superior performance than the best results of the base-learners, but that it also surpasses other recent algorithms.

The accuracy results of the meta-boosting algorithm are very promising. However, one of the issues facing not only this algorithm but also other ensemble learning approaches is that the computation complexity would be huge when confronted with big datasets. Therefore, we plan to apply the relatively simple programming interface of MapReduce to help solve these algorithms' scalability problems. However, this MapReduce framework suffers from an obvious weakness: it does not support iterations. This makes those algorithms requiring iterations (e.g. Adaboost) difficult to fully explore

the efficiency of MapReduce. In order to overcome this weakness and improve these algorithms's scalability to big datasets, we propose to apply Meta-learning programmed with MapReduce to realize the parallelization. This framework can also be extended to any algorithms which have iterations and needs to increase their scalability.

Our approach in this thesis is a win-win method: on one hand, we don't need to parallelize Adaboost itself directly using MapReduce, which reduces the massive difficulties caused by doing so; on the other hand, the high accuracies are secured as a result of the meta-learning and Adaboost integration. The experiments conducted on Hadoop fully distributed mode on Amazon EC2 demonstrate that our algorithm Parallelized-Meta-Learning algorithm (PML) reduces the training computational complexity significantly when the number of computing nodes increases, while generating smaller error rates than those on one single node. The comparison of PML with a parallelized AdaBoost algorithm shows that PML has lower error rates.

The methodologies proposed in this thesis are peer-reviewed and evaluated in the fields of ensemble learning and large-scale machine learning, as the results presented here are published. Our published papers are listed as follows:

- Liu, Xuan, Xiaoguang Wang, Stan Matwin, and Nathalie Japkowicz. "Meta-learning for large scale machine learning with MapReduce." In Big Data, 2013 IEEE International Conference on, pp. 105-110. IEEE, 2013.
- Liu, Xuan, Xiaoguang Wang, Nathalie Japkowicz, and Stan Matwin. "An Ensemble Method Based on AdaBoost and Meta-Learning." In Advances in Artificial Intelligence, pp. 278-285. Springer Berlin Heidelberg, 2013.

1.3 Thesis Outline

The thesis contains five chapters and is organized as follows. Chapter 2 introduces the related works and background on ensemble learning and large scale machine learning. Chapter 3 elaborates on the details of our meta-boosting algorithm to utilize the power of ensemble learning to construct more accurate machine learning algorithms. Chapter 4 presents our scalable meta-boosting algorithm using MapReduce deployed on the Hadoop platform. The experiments and comparison with other algorithms are also provided. Chapter 5 presents our conclusions and future work.

Chapter 2

Background

This chapter introduces some of the background information about ensemble learning and large-scale machine learning. Section 2.1 discusses the following aspects about ensemble learning: some of the important reasons to choose ensemble learning in practice; the theoretical reasons for constructing ensembles; why it is important from a theoretical point of view to have diverse base learners in an ensemble; available methods to build base learners and combine their outputs in order to create diversity; introductions to meta-learning and Adaboost algorithm based on the knowledge about the available methods for generating ensembles. Section 2.2 presents the following backgrounds about large-scale machine learning: how do we define large-scale problems in practice; a brief introduction about Hadoop, MapReduce and Hadoop distributed file system (HDFS) which we need to investigate in order to understand our parallelized meta-boosting algorithm in chapter 4; some recently developed systems and frameworks which realize parallel computing and try to make machine learning algorithms scalable. Section 2.3 provides some conclusions about this chapter.

2.1 Ensemble Learning

In ensemble learning, multiple machine learning algorithms are trained, and then, their outputs are combined to arrive at a decision. This process is shown in fig.2.1. The individual classifiers contained in an ensemble system are called base learners. Ensemble learning is in contrast to the more common machine learning paradigm which trains only one hypothesis on the training data. These "multiple machine learning algorithms" are regarded as a "committee" of decision makers. It is expected that by appropriately combining the individual predictions generated by these "multiple machine learning algorithms", the committee decision should produce better overall accuracy, on average, than any individual committee member.

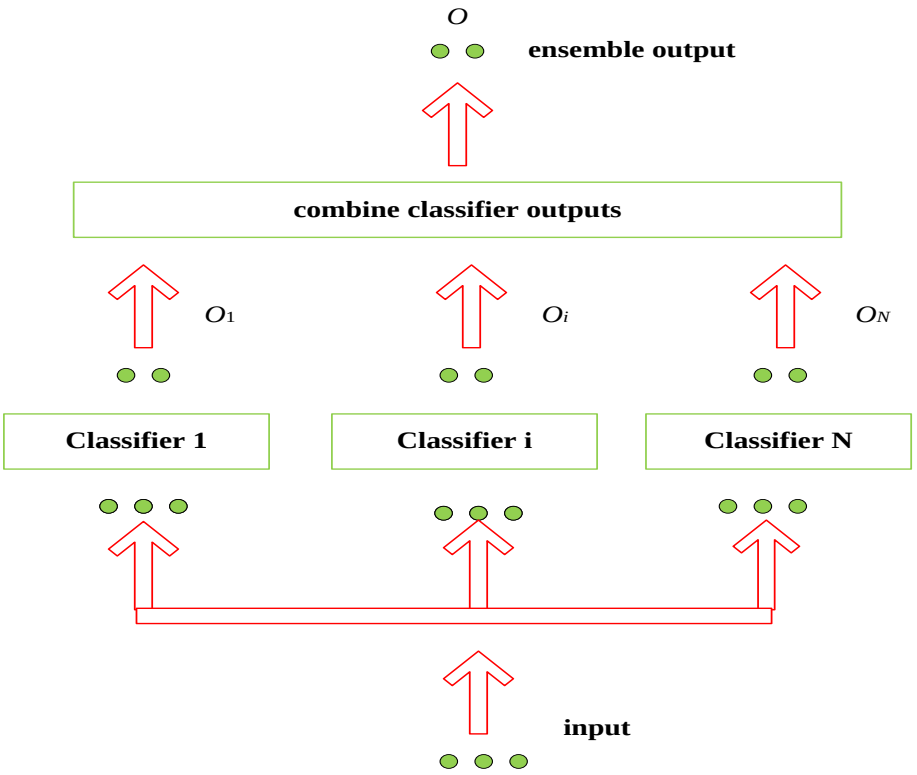

Fig.2.1. A schematic view of ensemble learning

For the committee of ensemble learning, if the errors of the committee members (total number is M) are uncorrelated, intuitively we would imagine that by averaging all these errors, the error of the ensemble can be reduced by a factor of M. Unfortunately, in practice, these individual errors are highly correlated and the reduced error for the ensemble is generally small. However, Cauchy's inequality tells us that the ensemble error will not exceed that of the base classifiers [18]. This statement is proved as follows: we compare the error rate of the ensemble to the average error of the individual models in the ensemble. The expected mean squared error of the ensemble is expressed as

$$E\left[\left(y^{(M)} - y_T\right)^2\right] = E\left[\left(\frac{1}{M}\sum_{m=1}^{M} y_m - y_T\right)^2\right] = E\left[\left(\frac{1}{M}\sum_m (y_m - y_T)\right)^2\right] = \frac{1}{M^2}E[(\sum_m \epsilon_m)^2] \quad (2.1)$$

Here $y^{(M)}(x)$ is ensemble average hypothesis, $y_T(x)$ is the true function, $y_m(x)$is the *m*th individual base hypothesis in an ensemble that has M base models, $\epsilon_m = y_m - y_T$ stands for the error of the *m*th individual base hypothesis. Applying Cauchy's inequality, we have

$$(\sum_{m=1}^{M} \epsilon_m)^2 \leq M \sum_{m=1}^{M} \epsilon_m^2 \quad (2.2)$$

Therefore, the following formula can be derived:

$$E\left[\left(y^{(M)} - y_T\right)^2\right] \leq \frac{1}{M}\sum_{m=1}^{M} E[\epsilon_m^2] \quad (2.3)$$

This means the expected error of the ensemble is smaller or equal to the average expected error of the base models in the ensemble.

The predictions generated by the "committee" could be class labels, posterior probabilities, real-valued numbers, rankings, clusterings, and so on. These predictions can be combined applying the following approaches: averaging, voting, and probabilistic

methods.

In addition to classification, ensemble learning can be applied to a wide spectrum of applications such as incremental learning from sequential data [19][20][21], feature selection and classifying with missing data [22], confidence estimation [23], error correcting output codes [24][25][26], data fusion of heterogeneous data types [27][28], class-imbalanced data [29][30][31], learning concept drift from non-stationary distributions [32][33] and so on.

2.1.1 Reasons to Choose Ensemble Methods

In the computational intelligence community, we prefer ensemble systems due to the following theoretical and practical reasons.

First, when the data volume is too large to be handled by a single classifier, an ensemble system can let each classifier process a partition of the data and then combine their results. This is one of the reasons we choose ensemble learning in our thesis.

Second, it is also beneficial to construct an ensemble system when the data available is not adequate. A machine learning algorithm usually requires sufficient and representative data to generate an effective model. If the data size is small, there is still a way to satisfy this condition by constructing an ensemble of classifiers trained on overlapping random subsets drawn from resampling the original data.

Third, when the decision boundary that separates different classes is too complex for a single classifier to cover, this decision boundary can be approximated by an appropriate

combination of different classifiers. For instance, although a linear classifier would not be capable of learning a complex non-linear boundary in a two dimensional and two-class problem, an ensemble of such linear classifiers can solve this problem by learning smaller and easier-to-learn partitions of the data space.

Finally, if there is a request for *data fusion*, an ensemble method is useful. *Data fusion* happens when we have data from various sources having features with different number and types that cannot be processed wholly by a single classifier. In this case, individual classifiers are applied on the data from the different sources and their outputs can be combined to make a final decision.

2.1.2 Reasons for Constructing Ensembles

In data analysis and automated decision making applications, there are three fundamental reasons to construct ensembles: statistical, computational and representational [34]. These reasons are derived by viewing machine learning as a search through a hypothesis space for the most accurate hypothesis.

The statistical problem happens when the data available is too small. When the training data is scarce, the effective space of hypotheses searched by the learning algorithm will be smaller than when the training data is plentiful. This case is illustrated in fig.2.2. The outer circle represents the space of all hypotheses: H. The shaded inner region denotes all hypotheses having good performances on the training data, such as h_1, h_2, h_3, h_4 and so on. Despite these hypotheses' good performances, their

generalization performances are different. Here, f represents the best hypothesis for the problem. If we randomly pick a hypothesis like h_1, this may cause the risk of selecting a bad solution for this problem. Instead of doing this, a safer option would be to "average" these hypotheses' outputs. By combining different hypotheses which all give the same accuracy on the training data, the risk of choosing the wrong classifier can be reduced. In fig 2.2, this means the resultant hypothesis by applying ensemble will be closer to f.

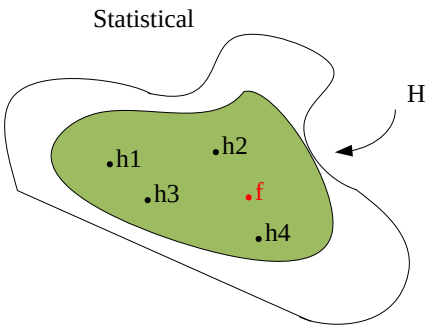

Fig.2.2.The statistical reason for combining classifiers.

As shown in fig.2.3, the computational reason pertains to learning algorithms which get stuck in local optima (h_1, h_2, h_3) when they perform a local search. For instance, neural network and decision tree algorithms both have this problem. In this case, even if the statistical problem is absent, it is still difficult for these algorithms to find the best hypothesis. Constructing an ensemble of individual classifiers generated from different starting points may produce a hypothesis which is closer to the true unknown function: f in this case.

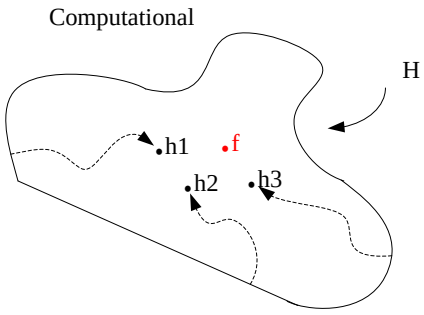

Fig.2.3.The computational reason for combining classifiers.

The representational problem appears when there is no hypothesis in the hypotheses space to represent the true function. That is to say, the true function is outside of the hypotheses space. Fig.2.4 depicts this case. In such cases, it is possible that the hypotheses space can be expanded by summing weighted hypotheses drawn from the hypotheses space. For example, if the best hypothesis for a dataset is nonlinear while the hypotheses space is restricted to linear hypotheses, then the best hypothesis would be outside this hypotheses space. However, any decision boundary with any predefined accuracy can be approximated by an ensemble of linear hypotheses.

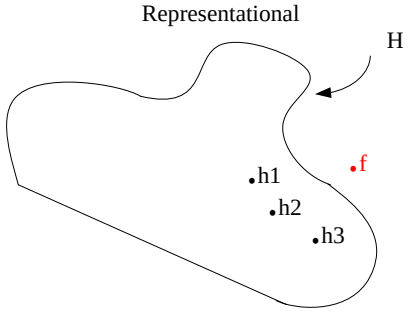

Fig.2.4. The representational reason for combining classifiers.

2.1.3 Ensemble Diversity

The error of a classifier contains two parts: the accuracy of this classifier (bias); the precision of the classifier when trained on different training sets (variance). It is well known that there is a trade-off between bias and variance, that is to say, classifiers having low bias turn out to have high variance and vice versa. It can be inferred that ensemble systems try to reduce the variance with their constituent classifiers having relatively fixed (or similar) bias.

Since combining classifiers with the same outputs generates no gains, it is important to combine classifiers which make different errors on the training data. Therefore, the diversity is desired for the errors of these classifiers. The importance of diversity for ensemble systems is discussed in [35] [36]. It is stated that the classifiers' errors should be independent or negatively correlated [37] [38]. Although the ensemble performance is inferior when there is a lack of diversity, the relationship between diversity and ensemble accuracy has not been explicitly established [39]. In practice, defining a single measure of diversity is difficult and what is even more difficult is to expressively relate a diversity measure to the ensemble performance. The ensemble methods which induce diversity intuitively are very successful, even for ensembles such as AdaBoost which weakens the ensemble members so as to gain better diversity. However, generating diversity explicitly applying the diversity measurement methods is not as successful as the aforementioned more implicit ways.

Since part of the ensemble prediction successes can be attributed to the accuracies of the individual models and the other part is due to their interactions when they are combined, the breakdown of the ensemble error has two components: "accuracy" and "diversity". Depending on the choice of the combiner rule and the type of the error function, the accuracy-diversity breakdown is not always possible.

For ensemble learning, we have the following notations:

- H $= \{h_1, \dots, h_n\}$: the set of individual classifiers in the ensemble;
- $K = \{k_1, \dots, k_l\}$: the set of class labels;
- $x \in \Re^m$: a vector with m features to be labelled in K.

Generally speaking, there are three types of base classifiers' outputs:

1) $O^i = \{h_{i,1}(x), \dots, h_{i,l}(x)\}$where the support of classifier h_i given to all class labels for x, for instance, is the estimated posterior probabilities $\hat{P}_i(k_j|x)$ with $j = 1, \dots, l$. The total outputs include all classifiers' predictions: $O = \{O^1, \dots, O^n\}$;
2) $O^i = \{h_i(x)\}$ where the predicted label from classifier h_i is returned, meaning that the total outputs will be all the predicted labels from all classifiers: $O = \{O^1, \dots, O^n\} = \{h_1(x), \dots, h_n(x)\}$;
3) $O^i = \{1\}$,if $h_i(x) = K(x)$; $O^i = \{0\}$, if $h_i(x) \neq K(x)$ where the outputs are correct/incorrect decisions or the oracle output, and $K(x)$ is the correct label for x. In other words, the outputs for classifier h_i would be equal to 1 if the predicted label is the same as x's correct label, otherwise, the output would be 0.

Depending on the types of classifier output, the diversity can be categorized as follows:

A. For the first case, where the classifier outputs are estimates of the posterior probabilities, we assume that the estimated posterior probability for class label k_j from classifier h_i is $\hat{P}_i(k_j|x)$ and the corresponding true posterior probability is expressed as $P(k_j|x)$. Therefore, the relationship between estimated and true posterior probability can be expressed as

$$\hat{P}_i(k_j|x) = P(k_j|x) + e_{i,j}(x) \tag{2.4}$$

Here, $e_{i,j}(x)$ is the error made by classifier h_i. The estimated posterior probabilities are linearly combined, through averaging or other methods which we will discuss in the later sections, to produce the final decision. The theoretical results about the correlations of the probability estimates and the ensemble classification error are derived in [40]. It is mentioned that the reducible classification error of a simple averaging ensemble: E_{add}^{ave} can be expressed as

$$E_{add}^{ave} = E_{add}\left(\frac{1+\delta(n-1)}{n}\right) \tag{2.5}$$

Where E_{add} is the classification error of an individual classifier, and δ is a correlation coefficient between the model outputs.When $\delta = 1$, the individual classifiers are identical (positively correlated), and we have $E_{add}^{ave} = E_{add}$. When $\delta = 0$, the individual classifiers are statistically independent (uncorrelated), and $E_{add}^{ave} = E_{add}/n$. When $\delta < 0$, the individual classifiers are negatively correlated, and E_{add}^{ave} is reduced even

further. To sum up the observation, the smaller the correlation, the better the ensemble. However, this equation 2.5 is derived under quite strict assumptions. One assumption is that the base classifiers generate independent estimates of the posterior probabilities $\hat{P}_i(k_j|x), j = 1, \dots, l$. We know this is not the real case since by design $\sum_j \hat{P}_i(k_j|x) = 1$. Another assumption is that the estimates of the posterior probabilities for different classes from different individual classifiers are independent. However, there is not enough information to find out whether the disagreement on this condition has an impact on the derived relation of equation 2.5.

B. For the second case when the individual classifiers' outputs are class labels, the classification error can be expressed as bias and variance (or spread) [41][42][43]. Here the diversity of the ensemble is the same as the variance (or spread).

C. For the third case when the outputs are correct/incorrect results, a list of pairwise diversity measures and non-pairwise diversity measures are discussed in [44].

2.1.4 Methods to Build Base Classifiers

There are several strategies available to achieve diversity among base classifiers. According to the analysis in section 2.1.3, these strategies can be distinguished by whether they encourage diversity **implicitly** or **explicitly**.

The majority of ensemble methods are implicit. There are three different ways:

- Some methods apply different subsets of the training data to train base learners. And different sampling methods lead to different ensemble systems. For example, *Bagging*[45] selects bootstrapped replicas of the training data as the training subsets for base learners; A variation of the *Bagging* algorithm is *Random Forests* [46], which uses decision tree as the base learner; *Random Subspace* methods [47] generate training subsets by selecting different subsets of available features of the training data.
- A less common strategy is to create different base classifiers by applying different training parameters. For instance, the base learners of *Neural Network Ensembles* can be constructed by using different initial weights, number of layers/nodes, error goals, etc. [48].
- Finally, another way is to apply different base learners. One of the examples is meta-learning [49]. In meta-learning, a set of base classifiers are created from different base learners. Their outputs on validation datasets are then combined with the actual correct classes to train a second level

meta-classifier.

The explicit alternatives apply some measurements to ensure the differences between ensemble members when constructing ensembles. By altering the distribution of training examples *Boosting* algorithms [50] select samples from the distribution so that instances which were previously misclassified have a higher probability to be selected. In this way, more accurate future predictions are encouraged on previously wrongly classified examples. By explicitly changing the distribution of class labels, the *DECORATE* algorithm [51] forces successive models to learn different answers to the same problem. By including a penalty term when learning individual ensemble members, *Negative Correlation Learning* [35] [36] manages the accuracy-diversity trade-off explicitly.

2.1.5 Methods to Combine Base Classifiers

There are two main categories of methods to combine base classifiers: one is the method of combining class labels, the other is the method of combining class-specific continuous outputs. The former requires only the classification decisions from the base classifiers while the latter needs the outputs of probabilities for the decisions. These probabilities represent the support the base classifiers give to each class.

A. Combining class labels

To combine class labels, there are four strategies: majority voting, weighted majority voting, behavior knowledge space (BKS) [52] and Borda count [53]. The first two

strategies are the most common ones and will be introduced in detail.

To do majority voting, there are three options: the final decision is the one all classifiers agree on (*unanimous voting*); the final decision is the one which is predicted by more than half the number of base classifiers (*simple majority*); the final decision is the one which receives the highest number of votes no matter whether the sum of these votes exceed 50% of the total votes (*plurality voting*). If it is not specified in the context, majority voting usually means the last one. A more detailed analysis of majority vote can be found in [54].

In weighted majority voting, a higher weight is given to the decision of the base classifier which has a higher probability of predicting a correct result. These weights will be normalized so that their sum is 1. Then the weighted votes will be combined just like in majority voting to select a decision which has the highest outcome. More information about weighted majority voting is available at [55].

B. Combining continuous outputs

Algorithms such as Multilayer Perceptron, Radial Basis function Networks, Naive Bayes and Relevance Vector Machines produce continuous outputs for each class. After normalization, these outputs can be regarded as the degree of support for each class. By satisfying certain conditions, these supports can be regarded as posterior probability for that class. For a base classifier k and instance x, if we represent this base classifier's prediction for class c as $k_c(x)$ and the normalized result (through softmax normalization [56]) as $\tilde{k}_c(x)$, then the approximated posterior probability $P(c|x)$ can be expressed as

$$P(c|x) \approx \tilde{k}_c(x) = \frac{e^{k_c(x)}}{\sum_{i=1}^{C} e^{k_i(x)}} \Rightarrow \sum_{i=1}^{C} \tilde{k}_i(x) = 1 \tag{2.6}$$

We refer to Kuncheva's decision profile matrix $DP(x)$ [57] to represent all the support results from all base classifiers. This matrix is shown in 1.2. In this matrix, each element stands for support and is in the range of [0, 1]; each row represents the support of a given base classifier to each class; each column offers the supports from all base classifiers for a given class.

$$DP(x) = \begin{bmatrix} d_{1,1}(x) & \cdots & d_{1,c}(x) & \cdots & d_{1,C}(x) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ d_{k,1}(x) & \cdots & d_{k,c}(x) & \cdots & d_{k,C}(x) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ d_{K,1}(x) & \cdots & d_{K,c}(x) & \cdots & d_{K,C}(x) \end{bmatrix} \tag{2.7}$$

To combine continuous outputs, there are two options: *class-conscious strategies* and *class-indifferent strategies*. The class-conscious strategies include *non-trainable* combiners and *trainable* combiners. The class-indifferent strategies consist of *decision templates* [57] and *Dempster-shafer based combination* [58] [59]. Here, we will focus on class-conscious strategies since the methodology we plan to propose in chapter 3 relies on them.

The class-conscious strategies are also called algebraic combiners. When applying the method of algebraic combiners, the total support for each class is calculated by using some algebraic function to combine the individual support from each base classifier for this class. If we indicate the total support for class c on instance $x$as $T_c(x)$, then we have

$$T_c(x) = f(d_{1,c}(x), \cdots, d_{k,c}(x), \cdots, d_{K,c}(x)) \tag{2.8}$$

Here f is one of the following algebraic combination functions and elaborated as follows.

1) non-trainable combiners

Once the ensemble members are trained, the outputs are ready to be combined and there is no need for extra parameters to be trained. **mean rule** takes the average of all supports from all base classifiers; **trimmed mean** discards unusually low or high supports to avoid the damage that may have done to correct decisions by these supports; **minimum / maximum / median rule** simply takes the minimum, maximum, or the median support among all base classifiers' outputs; **product rule** multiplies all the supports from each classifier and divides the result by the number of base classifiers ; in **generalized mean**, the total support received by a certain class c is expressed as

$$\mu_c(x) = \left(\frac{1}{K}\sum_{k=1}^{K}(d_{k,c}(x))^{\alpha}\right)^{1/\alpha} \tag{2.9}$$

Some already mentioned rules such as minimum rule ($\alpha \to -\infty$), product rule ($\alpha \to 0$), mean rule ($\alpha \to 1$) and maximum rule ($\alpha \to \infty$) are special cases of generalized mean.

2) Trainable combiners

It includes **weighted average** and **fuzzy integral** [60]. We choose to discuss weighted average in more details as it is most related to our methodology proposed in chapter 3. Weighted average is a combination of mean and weighted majority voting rules, which means that instead of applying the weights to class labels, the weights are applied to continuous outputs and then the result is obtained by taking an average of the weighted summation afterwards. For instance, if the weights are specific for each class, then the support for class c is given as

$$\mu_c(x) = \sum_{k=1}^{K} w_{kc}\, d_{k,c}(x) \tag{2.10}$$

Here w_{kc} is the weight specific to classifier k and class c. The individual supports for

class c are used to calculate the total support for itself. To derive such kind of weights, linear regression is the most commonly applied technique [61] [62]. A notable exception is that the weights of the algorithm *Mixture of Experts* [63] are determined through a gating network (which itself is typically trained using the *expectation-maximization* (**EM**) algorithm) and is dependent on the input values.

2.1.6 AdaBoost.M1 Algorithm

As mentioned in the previous sections, boosting algorithms are methods which encourage diversity explicitly and they combine the base classifiers' outputs using majority voting. Among the boosting algorithm variations, AdaBoost is the most well-known and frequently studied boosting algorithm. Here, we introduce the procedure of AdaBoost.M1 (one of several AdaBoost variants) step by step.

AdaBoost.M1 was designed to extend AdaBoost from handling the original two classes case to the multiple classes case. In order to let the learning algorithm deal with weighted instances, an unweighted dataset can be generated from the weighted dataset by resampling. For boosting, instances are chosen with probability proportional to their weight. For detailed implementation of AdaBoost.M1, please refer to Table 2.1. It was proven in [94] that a weak learner—an algorithm which generates classifiers that can merely do better than random guessing— can be turned into a strong learner using Boosting.

The AdaBoost algorithm generates a set of hypotheses and they are combined

through weighted majority voting of the classes predicted by the individual hypotheses. To generate the hypotheses by training a weak classifier, instances drawn from an iteratively updated distribution of the training data are used. This distribution is updated so that instances misclassified by the previous hypothesis are more likely to be included in the training data of the next classifier. Consequently, consecutive hypotheses' training data are organized toward increasingly hard-to-classify instances.

Table 2.1. AdaBoost.M1 Algorithm

Input: sequence of $m$examples $\{(x_1, y_1), \ldots, (x_m, y_m)\}$with labels$y_i \in Y = \{1, \ldots, K\}$

base learner B

number of iterations T

1. Initialize the weight for all examples so that $D_1(i) = \frac{1}{m}$

2. Do for $t = 1 \ldots T$

3. Select a training data subset S_t, drawn from the distribution D_t

4. Call the base learner$B$, train B with S_t

5. Generate a hypothesis $h_t: X \to Y$

6. Calculate the error of this hypothesis $h_t: \varepsilon_t = \sum_{i:h_t(x_i) \neq y_i} D_t(i)$

7. If$\varepsilon_t > 1/2$, then set $T = t - 1$and abort loop

8. Set $\beta_t = \varepsilon_t/(1 - \varepsilon_t)$

9. Update distribution$D_t: D_{t+1}(i) = \frac{D_i(i)}{Z_t} \times \begin{cases} \beta_t & if\ h_t(x_i) = y_i \\ 1 & otherwise \end{cases}$

Where $Z_t$is a normalization constant

10. End for

Output: the final hypothesis: $h_{fin}(x) = \underset{y \in Y}{argmax} \sum_{t:h_t(x)=y} log \frac{1}{\beta_t}$

As shown in table 2.1, all the training examples are assigned an initial weight $D_1(i) = \frac{1}{m}$ at the beginning of this algorithm. Using this distribution a training subset S_t is selected. The uniform distribution at the beginning of this algorithm is to ensure that all the examples have equal possibilities to be selected. The algorithm is set for T iterations. For each of the iteration t, the base learner B is called to perform on the training subset S_t. Then a hypothesis h_t is generated as an output of this training process. The training error is calculated as $\varepsilon_t = \sum_{i:h_t(x_i)\neq y_i} D_t(i)$. If this error is larger than 0.5 (which is higher than random guess) the loop will be aborted. Otherwise, a normalized error $\beta_t = \varepsilon_t/(1-\varepsilon_t)$ is calculated to update the distribution, which is shown in step 9 in table 2.1. Here Z_t is a normalization constant chosen so that D_{t+1} will be a distribution. The training process is also shown in fig.2.5.

To test new instances, this algorithm applies weighted majority voting. For a given unlabeled instance x, the class is selected as the one which gains the most votes from all the hypotheses generated during T iterations. In this process, each hypothesis h_t is weighted by $log\frac{1}{\beta_t}$ so that the hypotheses which have shown good performance are assigned with higher voting weights than the others.

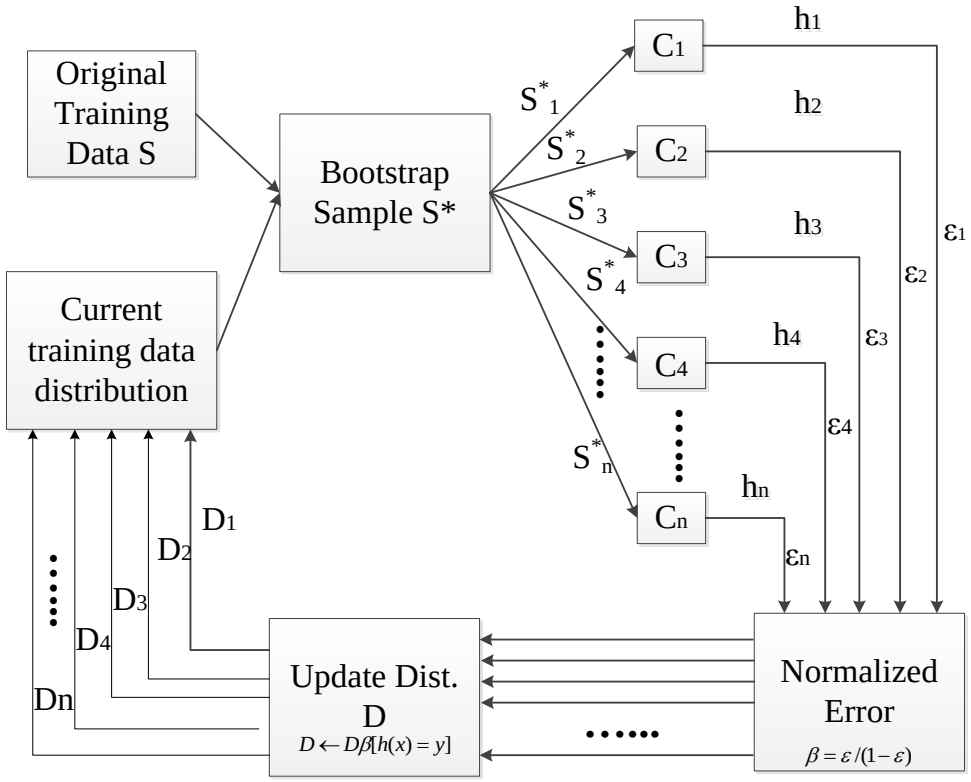

Fig.2.5. AdaBoost.M1 training process

The reason why boosting obtains good performance is that there is an upper bound of the training error which was proven in [64]:

$$E < 2^T \prod_{t=1}^{T} \sqrt{\varepsilon_t(1-\varepsilon_t)} \tag{2.11}$$

here E is the ensemble error. As ε_t is guaranteed to be less than 0.5, E decreases with each new classifier. With the iterated generation of new classifiers people would think, intuitively, that AdaBoost suffers from overfitting. However, it turns out that AdaBoost is very resistant to overfitting. A margin theory [65] was provided to explain this phenomenon. It is shown that the ensemble error is bounded with respect to the margin, but is independent of the number of classifiers.

2.1.7 Meta-learning Algorithm

According to the earlier sections, meta-learning is a method which encourages diversity implicitly. It also combines the base learners' continuous outputs using class-conscious strategies with trainable combiners. Meta-learning can be used to coalesce the results of multiple learners to improve accuracy. It can also be used to combine results from a set of parallel or distributed learning processes to improve learning speed. The methodology that we propose in chapter 3 will concentrate on the first case and the methodology in chapter 4 utilizes the second advantage of meta-learning.

The phenomenon of inductive bias [66] indicates that the outcome of running an algorithm is biased in a certain direction. In the No Free Lunch (NFL) theorems [91], it was stated that there is no universally best algorithm for a broad problem domain. Therefore, it is beneficial to build a framework to integrate different learning algorithms to be used in diverse situations. Here we present the structure of meta-learning.

Meta-learning is usually referred to as a two level learning process. The classifiers in the first level are called base classifiers and the classifier in the second level is the meta-learner. This meta-learner is a machine learning algorithm which learns the relationships between the predictions of the base classifiers and the true class. One advantage of this schema is that adding or deleting the base classifiers can be performed relatively easily since no communications are required between these base classifiers in the first level.

Three meta-learning strategies: combiner, arbiter and hybrid are proposed in [49] to combine the predictions generated by the first level base learners. Here we are focused on the combiner strategy since the methodology we will present in chapter 3 applies this strategy. Depending on how to generate the meta-level training data, there are three schemes as follows:

A. Class-attribute-combiner

In this case a meta-level training instance is composed by the training features, the predictions by each of the base classifiers and true class. In the case of three base classifiers, we represent the predictions of these classifiers on a training instance X is $\{P_1(X), P_2(X), P_3(X)\}$, the attribute vector for X is $< x_1, x_2, \dots, x_n >$ if the number of attributes is n, the true class for X is $lable(X)$. Then the meta-level training instance for X is

$$M_t = \{P_1(X), P_2(X), P_3(X), x_1, x_2, \dots, x_n, label(X)\} \quad (2.12)$$

B. Binary-class-combiner

For this rule, the meta-level training instance is made up by all the base classifiers' predictions for all the classes and the true class for instance X. The elements in this meta-level training instance basically include all the terms we have shown in equation 2.7 except that the predictions are not probabilities but binary. For example, if we have m numbers of classes and three base classifiers, then the meta-level training instance for X can be expressed as

$$M_t = \left\{P_{1_1}(X), \dots P_{1_m}(X), P_{2_1}(X), \dots P_{2_m}(X), P_{3_1}(X), \dots P_{3_m}(X), label(X)\right\} \quad (2.13)$$

Here, each prediction for each class is generated by the base classifier which is trained on

the instances which are labeled as the corresponding class or not.

C. Class-combiner

Similar with the Binary-class-combiner, the class-combiner rule also includes the predictions by all the base classifiers and the true class. The difference is that this rule only contains the predicted class from all base classifiers and the correct class. In this case, we express the meta-level training instance as

$$M_t = \{P_1(X), P_2(X), P_3(X), label(X)\} \tag{2.14}$$

In the methodology we will propose in chapter 3, the third rule: class-combiner rule is applied. This combiner rule is not only employed in the training process but also in the testing process to generate the meta-level test data based on the new test instance. The procedure for meta-learning applying the combiner rule is described as follows: for a dataset $S = \{(y_n, x_n), n = 1, \dots, N\}$ with y_n as the nth class and x_n as the nth feature values for instance n. Then for J-fold cross validation, this dataset is randomly split into J data subsets: $S_1, \dots, S_j, \dots S_J$. For the jth fold, the test dataset is denoted as S_j and the training dataset is expressed as $S^{-j} = S - S_j$. Then each of the training datasets is again split randomly into two almost equal folds. For instance, S^{-j} is split into T_{j1}, T_{j2}. We assume that there are L level-0 base learners.

Therefore, for the jth round of cross validation, in the training process, first T_{j1} is used to train each the base learners to generate L models: $M_{j11}, \dots, M_{j1l}, \dots M_{j1L}$. Then, T_{j2} is applied to test each of the generated models to generate the predictions. These predictions are used to form the first part of meta-level training data. The second part of these data is produced by generating the base models on T_{j2} and testing the models on

T_{j1}. If there are G instances in S^{-j} and the prediction generated by model M_{j1L} is denoted as Z_{Lg} for instance x_g, then the generated meta-level training data can be expressed as

$$S_{CV} = \{(Z_{1g}, \dots, Z_{Lg}, y_g), g = 1, \dots, G\} \quad (2.15)$$

This meta-level training data is the combination of the first and second parts' predictions. Based on these data, the level-1 model is generated as $\widetilde{M}$. The last step in the training process is to train all the base learners on S^{-j} to generate the final level-0 models: $M_1, \dots M_l, \dots M_L$. The whole training process is also shown in fig.2.6.

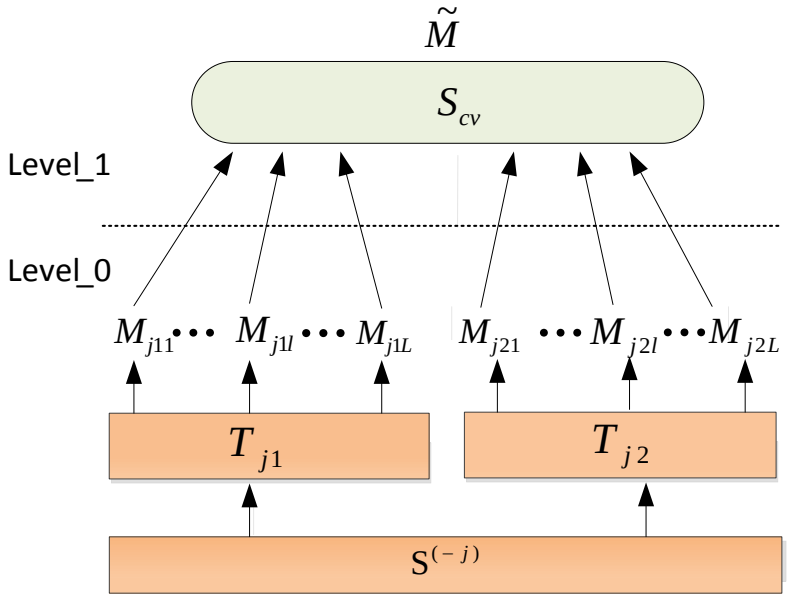

Fig.2.6. Meta-learning training process for fold j

In the test process, to generate the meta-level test data, S_j is used to test the level-0 base models $M_1, \dots M_l, \dots M_L$ generated in the training process. For instance, for the instance x_{test} in S_j, the meta-level test instance can be obtained as $(Z_1, \dots, Z_l, \dots, Z_L)$. All the meta-level test data are feed into the meta-level model $\widetilde{M}$ to generate the predictions for these test data. We also present the pseudo code for the meta-learning training algorithm for the jth cross validation training process in table 2.2.

Table 2.2. Meta-learning algorithm for training process

Input:

Training dataset: S^{-j}

Meta learning algorithms:C^m

Base learning algorithms:$\{\mathrm{C}_1, \mathrm{C}_2, \ldots, \mathrm{C}_\mathrm{L}\}$

Output:

Ensemble E

1. $E = \emptyset$

2. $S_{cv} = \emptyset$

3. $T_{j1}, T_{j2} = \mathrm{SplitData}(S^{-j}, 2)$

4. **For**$l = 1$ to L**do**

5. $M_{j1l} = C_l(T_{j1})$

6. $M_{j2l} = C_l(T_{j2})$

7. **End for**

8. $S_{cv}^1 = \{M_{j11}(x_i), M_{j12}(x_i), \ldots M_{j1L}(x_i), y_i\}$

9. $S_{cv}^2 = \{M_{j21}(x_k), M_{j22}(x_k), \ldots M_{j2L}(x_k), y_k\}$

10. $S_{cv} = \bigcup_{i=1}^{2} S_{cv}^i$

11.$\widetilde{M} = C^m(S_{cv})$

12.$\{M_1, M_2, \ldots M_L\} = \{C_1(S^{-j}), C_2(S^{-j}), \ldots, C_L(S^{-j})\}$

13. $E = (\{M_1, M_2, \ldots M_L\}, \widetilde{M})$

14.**Return** E

2.2 Large Scale Machine Learning

Although the MapReduce framework is very popular in the industry for cloud computing, its power to construct scalable machine learning algorithms is rarely recognized in the academic field. Basically, to construct scalable machine learning algorithms using MapReduce, knowledge of the following three components are indispensible: the Hadoop platform which is an open source implementation of MapReduce, the MapReduce programming framework and the Hadoop distributed file system. Being acquainted with their roles in the system is beneficial for us to construct new scalable machine learning algorithms. In addition to MapReduce and Hadoop some other low-level and high-level systems were also proposed in the very recent years. The active research in the scalable machine learning area is boosting its development to a new level in the future.

This section first discusses the datasets which can be categorized as large scale for machine learning algorithms and then proceeds to introduce Hadoop, MapReduce and Hadoop distributed file systems which are the three pillars for implementing scalable machine learning algorithms with MapReduce. Finally, some recent advances in methodologies and systems for scalable machine learning are presented and the motivation for our proposal of the methodology in chapter 4 is also presented.

2.2.1 Large Scale Datasets

Here we discuss the spectrum of 'large scale' not according to their respective domains but to the characteristics of the problem. The most relevant method to determine 'large scale' is through the number of records. For example, the genetic sequences database of the US National Institute of Health: GenBank[3] contains more than 125 million gene sequences and more than 118 billion nucleotides on October 2010 while the corresponding numbers are 82 million and 85 billion in 2008. Up to 700 megabytes of data can be generated per second by the Large Hadron Collider[4]. Moreover, up to 1 billion base pairs can be sequenced by the next-generation sequencing technologies in a single day [67]. Furthermore, the large records of data not only come from the natural sciences but also from the industry. For instance, one of the most famous data mining challenges in the latest years: Netflix prize[5] released an anonymized version of their movie ratings database of over 100 million ratings from 480,000 customers and 18,000 movies. The public was challenged to provide a recommendation algorithm which outperforms by 10% their own proprietary method.

Another challenging aspect is the number of variables of the problem. As an example, one of the most widespread technologies of molecular biology research: microarray analysis [68] usually generates data having few records with tens of thousands

[3]Genbank release notes. Available at: ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt.

[4] Physicists brace themselves for LHC 'data avalanche'. Available at: http://www.nature.com/news/2008/080722/full/news.2008.967.html.

[5]The netflix prize. Available at: http://www.netflixprize.com.

of variables as the generation of each record is very expensive. As a result, there is a very high imbalance between records and variables and many machine learning algorithms tend to overfit this kind of data. Another example in the natural sciences domain is protein structure prediction (PSP)[6] which also generates datasets which have far more number of variables than records.

Last but not least, the large number of classes is also another source of difficulty for machine learning methods. The most well-known example happens in the information retrieval/text mining area where datasets have an extremely high number of classes[7]. The other two class related difficulties are for datasets which have a hierarchical structure of the class and certain classes have very low frequency (the class imbalance problem [69]). Although this class imbalance problem cannot be considered as a difficulty specially for large scale data mining as it also happens for small data sets, it could become a recurrent problem when datasets grow and tend to become more heterogeneous.

In sum, the question we are trying to answer is when a dataset becomes large scale. This is not an easy question to address as it depends on the type of learning tasks and resources available to analyze the data. The simplest answer is when a dataset's size starts to become an issue to take into account explicitly in the data mining process[8]. If an algorithm only needs to process the training set through one single pass, it would be easy for the algorithm to process hundreds of millions of records. However, in reality an algorithm usually needs to use the training set over and over again. In this case, the

[6] The ICOS PSP benchmarks repository. Available at: http://icos.cs.nott.ac.uk/datasets/psp_benchmark.html.

[7]Reuters-21578 text categorization collection. Available at: http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html.

[8]Dumbill E. What is big data?, 2012. Available at: http://radar.oreilly.com/2012/01/what-is-big-data.html

algorithm struggles with only a few tens of thousands of instances. The same thing happens with the number of attributes (variables).

2.2.2 Hadoop

Large scale data has brought both benefits and challenges. One of the benefits is that we can extract lots of useful information by analyzing such big data. Extracting knowledge from massive data sets has attracted tremendous interest in the data mining community. In the field of natural language processing, it was concluded [70] that more data leads to better accuracy. That means, no matter how sophisticated the algorithm is, a relatively simple algorithm will beat the complicated algorithm with more data. One practical example is recommending movies or music based on past preferences [71].

One of the challenges is that the problem of storing and analyzing massive data is becoming more and more obvious. Although the storage capacities of hard drives have increased greatly over the years, the speeds of reading and writing data have not kept up the pace. Reading all the data from a single drive takes a long time and writing is even slower. Reading from multiple disks at once may reduce the total time needed, but this solution causes two problems.

The first one is hardware failure. Once many pieces of hardware are used, the probability that one of them will fail is fairly high. To overcome this disadvantage, redundant copies of the data are kept, in case of data loss. This is how Hadoop's file system: Hadoop Distributed File System (HDFS) works. The second problem is how to

combine data from different disks. Although various distributed systems have provided ways to combine data from multiples sources, it is very challenging to combine them correctly. The MapReduce framework provides a programming model that transforms the disk reads and writes into computations over sets of keys and values.

Hadoop is an open source Java implementation of Google's MapReduce algorithm along with an infrastructure to support distribution over multiple machines. This includes its own filesystem HDFS (based on the Google File System) which is specifically tailored for dealing with large files. MapReduce was first invented by engineers at Google as an abstraction to tackle the challenges brought about by large input data [8]. There are many algorithms that can be expressed in MapReduce: from image analysis, to graph-based problems, to machine learning algorithms.

In sum, Hadoop provides a reliable shared storage and analysis systems. The storage is provided by HDFS and the analysis by MapReduce. Although there are other parts of Hadoop, these two are its kernel components.

2.2.3 MapReduce

MapReduce simplifies many of the difficulties in parallelizing data management operations across a cluster of individual machines and it becomes a simple model for distributed computing. Applying MapReduce, many complexities, such as data partition, tasks scheduling across many machines, machine failures handling, and inter-machine communications are reduced.

Promoted by these properties, many technology companies apply MapReduce frameworks on their compute clusters to analyze and manage data. In some sense, MapReduce has become an industry standard, and the software Hadoop as an open source implementation of MapReduce can be run on Amazon EC2[9]. At the same time, the company Cloudera[10] offers software and services to simplify Hadoop deployment. Several universities have been granted access to Hadoop clusters by Google, IBM and Yahoo! to advance cluster computing research.

However, MapReduce's application to the machine learning tasks is poorly recognized despite of its growing popularity. It is natural to ask the question whether the MapReduce-capable compute infrastructure could be useful in the development of parallelized data mining tasks with the wide and growing availabilities of such infrastructures.

A. MapReduce Framework

As a programming model to process big data, there are two phases included in the MapReduce programs: Map phase and Reduce phase [72]. The programmers are required to program their computations into Map and Reduce functions. Each of these functions has key-value pairs at their inputs and outputs. The input is application-specific while the output is a set of <key, value> pairs, which are produced by the Map function. The key and value pairs are expressed as follows:

$$[(k_1, v_1), \dots, (k_n, v_n)]: \forall i = 1 \dots n : (k_i \in K, v_i \in V) \quad (2.16)$$

Here $k_i$represents key for the ith input and v_i denotes the value for the ith input. K

[9]http://aws.amazon.com/ec2/

[10]http://www.cloudera.com/

is a key domain and V is the domain of values. Using the Map function, these key-value pairs of the input are split into subsets and distributed to different nodes in the cluster to process. The processed intermediate results are also key-value pairs. Therefore, the map function can be obtained as

$$Map: K \times V \rightarrow (L \times W)^* \tag{2.17}$$

$$(k, v) \mapsto [(l_1, x_1), \dots, (l_1, x_r)] \tag{2.18}$$

Here L and $W$are key and value domains again, which also represents the intermediate key-value pair results.In the map process, each single key-value input pairs: (k, v) is mapped into many key-value pairs: $[(l_1, x_1), \dots, (l_1, x_r)]$ with the same key, but different values. These key-value pairs are the inputs for reduce functions. The reduce phase is defined as

$$Reduce: L \times W^* \rightarrow W^* \tag{2.19}$$

$$(l, [y_1, \dots, y_{sl}]) \mapsto [w_1, \dots, w_{ml}] \tag{2.20}$$

The first step in the reduce function is to group all intermediate results with the same key together. The reason to perform this aggregation is that although the input key-value pairs have different key values, they may generate the intermediate results with the same key values. Therefore, there is a need to sort these results and put them together to process. This process is achieved when $(L \times W)^*$ is processed to generate $L \times W^*$. And these are the inputs for the reduce functions. The Reduce process can be also parallelized like the Map process. And the result is that all the intermediate results with the same keys: $L \times W^*$ are mapped into a new result list: W^*.

In sum, the whole MapReduce process can be expressed as

$$MapReduce: (K \times V)^* \rightarrow (L \times W)^* \quad (2.21)$$

As have mentioned before, to realize parallelization many map and reduce functions are performed simultaneously on various individual machines (or nodes in the context of cluster) with each of them processing just a subset of the original dataset.

B. MapReduce Workflow

In order to do cloud computing, the original data is divided into the desired number of subsets (each subset has a fixed-size) for the MapReduce jobs to process. And these subsets are sent to the distributed file system HDFS so that each node in the cloud can access a subset of the data and do the Map and Reduce tasks. Basically, one map task processes one data subset.

Since the intermediate results output by the map tasks are to be handled by the reduce task, these results are stored in each individual machines' local disk instead of HDFS. To perform fault tolerance, another machine is automatically started by Hadoop to perform the map task again, once one of the machines which run the map functions failed before it produced the intermediate results.

The number of reduce tasks is not determined by the size of the input, but specified by the user. If there is more than one reduce task, the outputs from the map tasks are divided into pieces to feed into the reduce functions. Although there are many keys in a map tasks' output, the piece sent to the reduce task to handle contains only one key and its values. As each reduce task will have the inputs from multiple map tasks, this data flow between map tasks and reduce tasks is called "the shuffle".

Fig.2.7 depicts the work flow of a generalized MapReduce job. To execute such job the following preparation information is need: the input data, the MapReduce program

and the configuration information. As we have discussed before, there are two types of tasks involved in a MapReduce job: the map tasks and the reduce tasks. To control the job execution process, a jobtracker and some tasktrackers are configured. The tasks are scheduled by the job tracker to run on tasktrackers. And the tasktrackers report to the jobtracker about the situations of the tasks running. By doing this, if some tasks fail, the jobtracker would know it and reschedule new tasks.

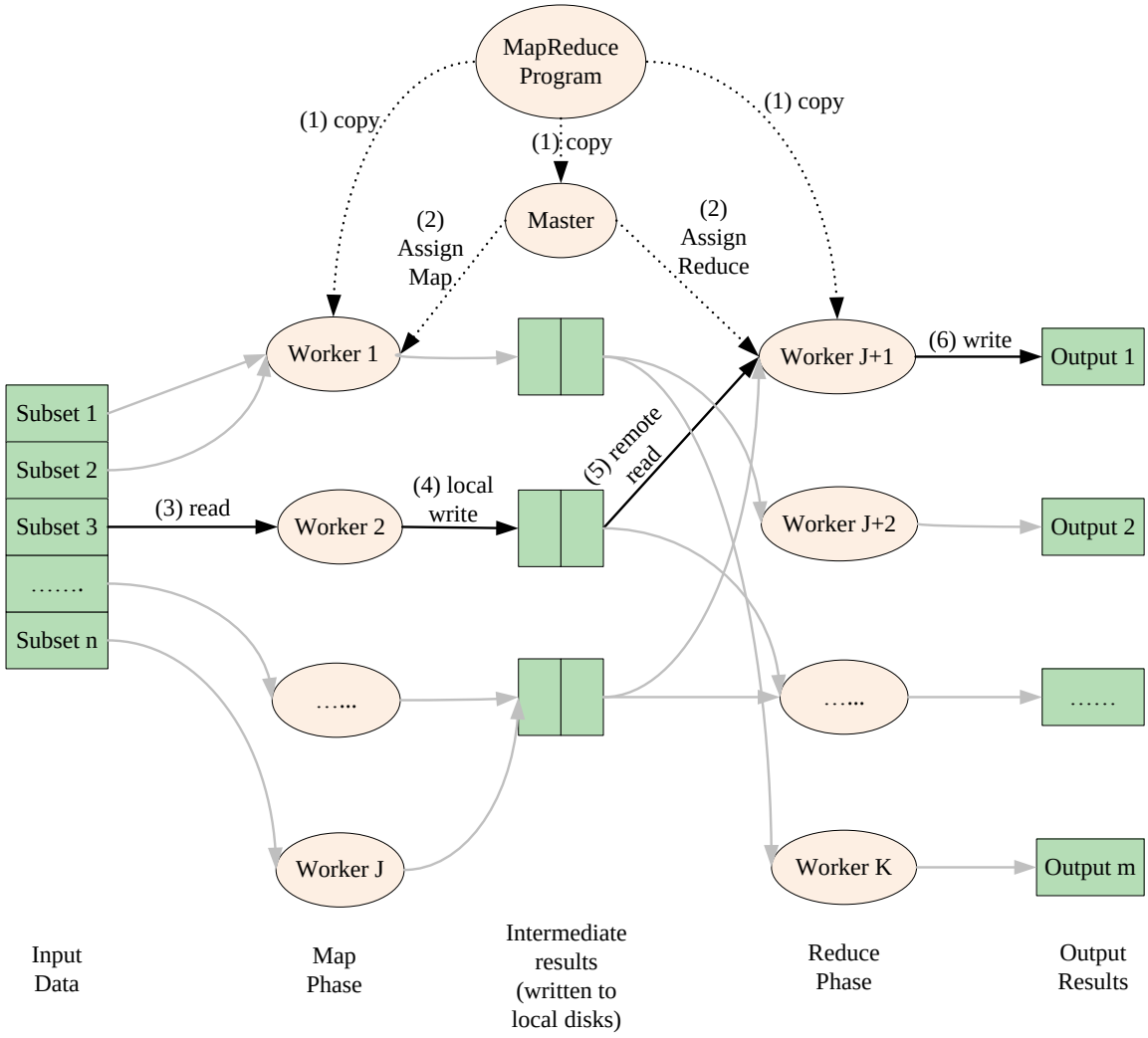

Fig.2.7. Work flow of MapReduce framework

Now we have the basic knowledge about the MapReduce jobs, let's take a closer look at the six steps shown in fig.2.7.

Step 1: the input data is divided into n fixed-size subset datasets. The size for each of these subsets is usually between 16 to 64 MB and can be controlled by the client. At the same time, the copies of the MapReduce program are sent to both the master and the workers. Here the master is also a computer node. The difference between the master and the worker (or "slave") is that the master will assign works to the workers and control their progress. Load balancing is desired in this case, as a faster worker may be able to process more data than a slower worker, and even if the workers' abilities are the same, load balancing would be affected when some processes fail or there are other jobs running simultaneously.

Step 2: There are J workers which will do the map tasks and $(K-J)$ workers which will execute the reduce tasks. In this step, the master assigns map or reduce tasks to each idle worker.

Step 3: The worker which is chosen to process map task reads a subset dataset which has already been sent to the HDFS. This subset data is parsed into key-value pairs and feeds to the map function one by one to be processed. The generated intermediate results from the map task are stored in memory.

Step 4: After a fixed period, the intermediate results in memory are written to a local disk. They are then divided into $(K-J)$ partitions based on the customized portioning function. The locations for each of the partitions are forwarded to the master, so that it can again send this information to workers which process reduce tasks.

Step 5: Using the information about the locations of the partitions, the reduce workers remotely read the partitions of the intermediate outputs. Once the reading is

complete for its own partition, the reduce worker starts to sort all the key-value pairs in the partition so that the records which have the same key are grouped together. This process is also regarded as a “sort & merge” process. The result is a list of key-value pairs with unique key domain. The necessity of this manipulations originates from the fact that each partition indicated for the reduce works has intermediate results from more than one map task.

Step 6: Based on the sorted results from step 5, the program passes all the values with the same key to the reduce function. This process iterates until all the values corresponding to each unique keys are handled. Then, the outputs of the reduce function for all the key-value pairs are appended to each other to output the final results. Each reduce worker corresponds to one output data. Here, in fig.2.7, there are $(J-K)$reduce workers and m outputs. Therefore, $(J-K)=m$.

Step 7: This step cannot be shown in fig.2.7 as it deals with the user code. When the entire map and reduce tasks are completed, the user program is wakened up by the master and returns back to the user code. That is to say, the MapReduce function can be one sub function to be called in a user’s program.

C. A Simple Word Count Example

In order to illustrate the work flow of MapReduce and the procedure of how the <key, value>pairs are processed, here we show a simple word count program which count the number of consonant and vowel letters in the string "MACHINELEARNING". This example is shown in fig.2.8. In the first step, each letter is assigned a unique identification number picked from 1 to 15. This identification number is the "key" and the

letter itself is the "value". Therefore, the input has fifteen<key, value> pairs. Then these pairs are split into three partitions and each partition is processed by a mapper to generate the intermediate <key, value> pairs. Here, the key is either "consonant" or "vowel" and the value is still the letter itself. The next very important step is that these intermediate <key, value> pairs are sorted and merged so that the values which belong to the same key are grouped together to form two categories: consonant and vowel. In the final step, the reducer calculates the number of consonant and vowel letters and output the results.

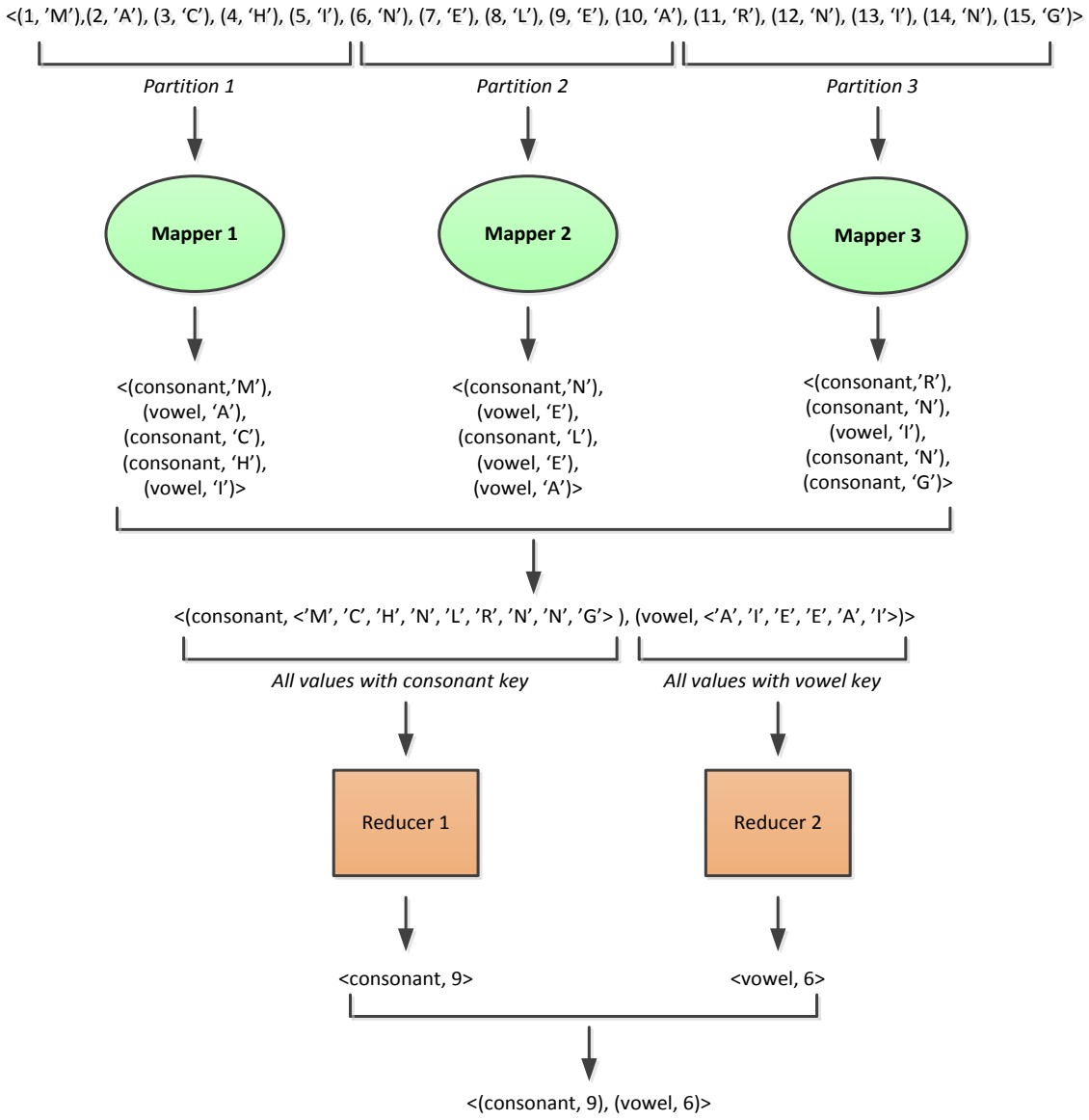

Fig.2.8. A word count example with MapReduce

D. Comparison with Other Frameworks

In comparison with the relational database management system (RDBMS), grid computing and volunteer computing, MapReduce has the following advantages. The comparison results are also shown in table 2.3.

The traditional structure applied in RDBMS is B-Tree which relies on seeks to perform data inquiries. Seeks are processes which read or write data by moving a disk's head to a specific place on a disk. As we have mentioned before, the advances in seeking rate is slower than the transfer rate which is correlated with MapReduce. As introduced in the MapReduce workflow, MapReduce builds the database through Sort/Merge which utilizes the bandwidth, and the bandwidth is related to the transfer rate. Therefore, RDBMS is less efficient than MapReduce when updating most of the databases. MapReduce is more suitable to deal with whole datasets in batch fashion, while RDBMS is specialized in point queries or updates on small amount of data.

The traditional large-scale data processing is accomplished by High Performance Computing (HPC) and grid computing. For grid computing, which uses the Message Passing Interface (MPI) as API (Application Program Interface), the problem appears when nodes in the cluster need to access larger data volumes (hundreds of gigabytes) as it is limited by the network's bandwidth. On the contrary, MapReduce executes data locality: data are collocated with the compute node; as the data is local the access rate is fast. MPI requires the programmer to handle the mechanics of the data flow explicitly, while MapReduce operates only at the higher level and the data flow is implicit. Therefore, MPI programs are more difficult to write as they have to explicitly manage

their own checkpointing and recovery, while in MapReduce the program detects failed map or reduce tasks and fixes the problem automatically without letting the programmer think about the failure.

There are many well-known volunteer computing projects such as the Great Internet Mersenne Prime Search [11], Folding@home [12] and SETI@home [13]. Among these projects,SETI@home is the most famous one.SETI represents for the“Search for Extra-Terrestrial Intelligence”. This project searches for signs of extra terrestrial intelligence by analyzing radio telescope data utilizing the CPU time donated by volunteers from their idle computers. Although it seems that the volunteer computing also breaks a big problem into smaller work units and send them to numerous individual computers to calculate the results, there are many differences between volunteer computing and MapReduce. First, volunteer computing is very CPU-intensive and the computer resources donated by volunteers are CPU cycles, not bandwidth. Second, volunteer computing runs on untrusted machines on the Internet. These machines have highly variable connection speeds and there is no data locality functionality designed. In contrast, MapReduce has data locality and runs jobs on trusted and dedicated machines with very high aggregate bandwidth for internal communications.

[11] http://www.mersenne.org/

[12] http://folding.stanford.edu/

[13] http://setiathome.berkeley.edu/

Table 2.3. Comparisons between MapReduce and other systems

	MapReduce	RDBMS	Grid computing	Volunteer Computing
Efficiency	√	×		
Data Locality	√		×	×
High Level	√		×	
Trusted machine	√			×
High bandwidth	√		×	×

2.2.4 Hadoop Distributed File System (HDFS)

As we have seen in the work flow of MapReduce, the user's MapReduce program first needs to be copied to the nodes in the cluster in order to perform computations. Here the action of copy is to move the user's program to the HDFS so that every node in the cluster can access it. In addition to this, every split data subset is also stored in HDFS. Thus, the HDFS manages the storage across a cluster, and also provides fault tolerance. To prevent the data loss caused by possible node failure is a very challenging task. Therefore, the programming of HDFS is more complex than that of regular disk filesystems.

The block size of filesystems is often an integral multiple of the disk block size. For HDFS, it has a block size of 64 MB by default. There are three advantages using this fixed block size:

- First, big files can be easily divided into chunks of blocks to be sent across a

number of nodes. There is no need to store this file on one machine. This handles the files which initially exceeded the storage capacity of one machine.

- Second, compared to storing files, using blocks simplifies the storage subsystem. The storage management can be simplified (for fixed-size blocks, it is easy to calculate how many can be stored on a given disk) and the metadata concerns are eliminated (blocks just store chunks of data, while another system can handle the metadata separately).
- Finally, applying the strategy of blocks is very helpful for providing fault tolerance and availability by replicating blocks. To prevent disk and machine failure, each block is replicated on a number of different nodes (the number is usually three).

An HDFS cluster works in the manner of master and slave. The master is called the namenode while the slave is called the data node. The filesystem namespace is managed by the namenode. The filesystem tree and the metadata for all the files and directories are maintained in it. This information is stored in the form of two files: the namespace image and the edit log. Datanodes are the ones that do the real jobs: they store and retrieve blocks when they are required to do so by clients or the namenode. They also need to transfer back the information of the lists of blocks they are storing to the name nodes periodically. The reason is that the namenode doesn't store block locations persistently and the information is reconstructed from the datanodes. Fig.2.9 illustrates the case when the block replication is three and two files "/user/aaron/foo"&"/user/aaron/bar" are

divided into pieces.

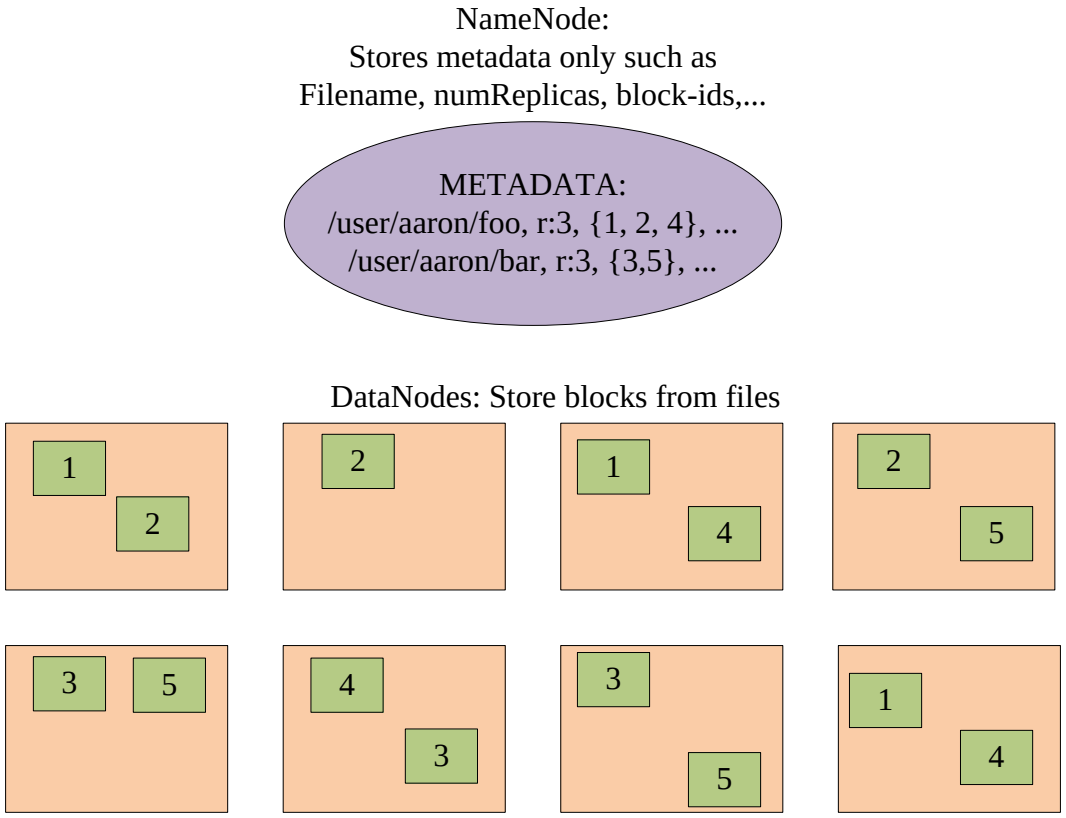

Fig.2.9. Information stored in namenode and datanodes.

By communicating with the namenode and the datanodes a client can access the filesystem representing the user. The user code doesn't need to know the details of the namenode and datanode as a filesystem interface similar to a portable operating system interface (POSIX) is presented to the user by the client. The data flow for a file read is demonstrated in fig.2.10. When the client wants to read a file in step 1, it opens an instance of DistributedFileSystem which will communicate with the namenode to obtain the locations for the blocks of files in step 2. The namenode provides the addresses for all datanodes which hold replications of the blocks. These addresses are then sorted according to their distance to the client and the closest datanode's address is chosen. In step 3, the client reads the data block through the FSDataInputStream which is returned

by the DistributedFileSystem. In step 4, it reads the first data block from the closest datanode. After finish reading the block in step 4, the datanode is closed and it continues to read the next block also from the closest datanode in step 5. This happens for some cycles until the client has finished reading all the blocks it need. In the final step: step 6, the FSDataInputStream is closed.

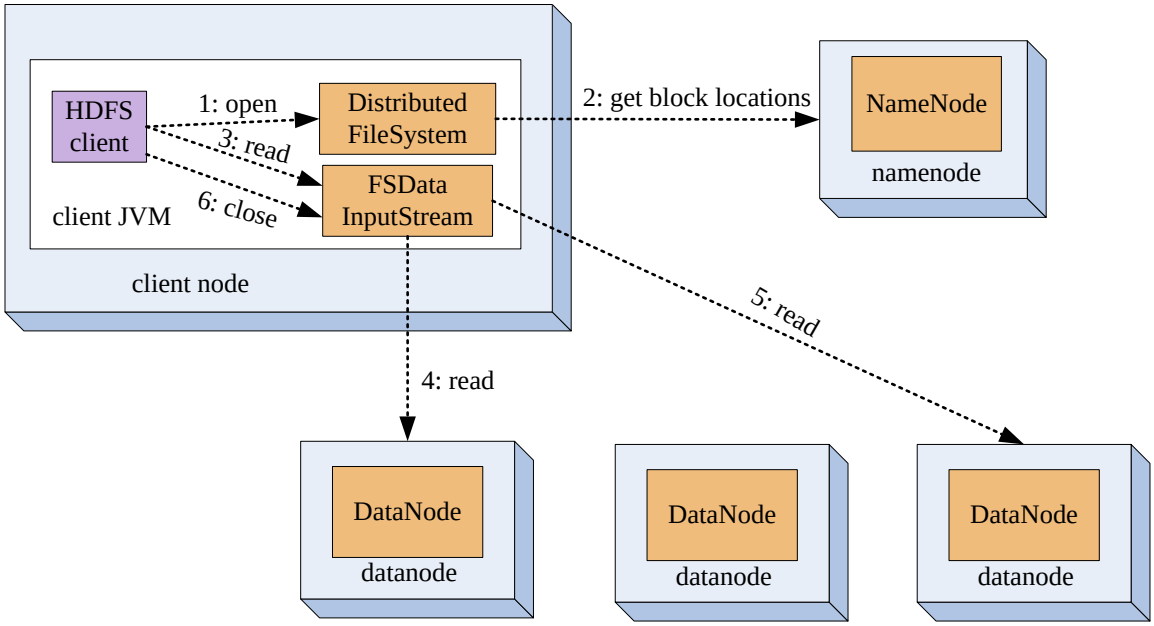

Fig.2.10. Client reads data from datanodes through HDFS

In the namenode and datanodes style, there is a risk for namenode failure. Once the namenode fails there would be no information available to retrieve the files from blocks on the datanodes and all files on the filesystem would also be lost. In order to combat against the namenode failure, Hadoop provides two choices. The first one is that the namenode writes its persistent state to multiple filesystems. Usually, this is accomplished by writing to both the local disk and a remote NFS mount. The second option is to run a secondary namenode which keeps a copy of the merged namespace image to be used in the case of namenode failure.

2.2.5 Related Work

As we mentioned in section 2.2.3, HPC has been doing parallel computing for years. The programming models it relies on are symmetric multiprocessing (SMP) and openMP for programming multicore processors, MPI for parallel clusters, and unified device architecture (CUDA) for graphics processing units (GPU).

Here we list some advances in HPC. **SNOW** [73] is built on top of MPI packages and it requires explicit distribution of data across worker nodes. So, it is still a low level system. **MLPACK** [74] is a scalable C++ machine learning library optimized for multicore architectures. It aims to bridge the gap between scalability and ease of use. However, the design and implementation of new scalable machine learning methods are not simplified. **OptiML** [75] is a domain-specific language (DSL) for machine learning and an implicitly parallel, expressive and high performance alternative to MATLAB and C++. It performs domain-specific analyses and optimizations and automatically generates CUDA code for GPUs. **SEJITS**[76]stands for selective embedded just-in-time specialization. It combines the efficiency-language (such as CUDA, Cilk, C with OpenMP) performance with productivity-language (such as Python) programmability. Both OptiML and SEJITS need to find regions of code to be mapped to efficient low-level implementations. Normally, this process is done by pattern matching. However, this task is difficult in machine learning because it is a rapidly evolving discipline.

The above mentioned parallel computing structures don’t aim to make machine

learning algorithms scalable. Here, we discuss the following systems which fulfill this job.

Matlab and R have parallel computing toolboxes which enable distributed objects to be created and message-passing functions to be defined to implement task parallel algorithms. The programmer is directly exposed to message-passing systems in R packages like **rmpi** and**rpvm**. As we have discussed in section 2.2.3, MapReduce has advantages over these systems, as these packages require parallel programming expertise.

There are some current efforts to make R scalable to massive datasets either through integrating R with Hadoop or scaling R itself directly. **Ricardo** [77] expresses large-scale computations in JAQL and it is a R-Hadoop system. JAQL is a high-level query interface on top of Hadoop. As part of the eXtreme Analytics Platform (XAP) project at the IBM Almaden Research center, Ricardo decomposes the data-analysis algorithms into two parts: one part (the smaller-scale single-node statistical tasks) is executed by the R statistical analysis system; the other part is handled by the Hadoop data management system. One disadvantage about Ricardo is that the programmer is required to find out the scalable components of an algorithm and re-express large-scale matrix operations through JAQL queries. **Rhipe** [78] is also an integrated processing environment with R and Hadoop. However, it needs the programmer to directly write Map and Reduce functions and runs R instances on Hadoop nodes. **Presto** [79] is a distributed system that extends R and addresses many of limitations, such as not being able to scale to large datasets. It is designed based on the observation that machine learning and graph algorithms are based on matrix computation. So, they apply R in this system as it is an array-based language.

MADLib [80] provides a machine learning library for relational database systems. **Google prediction API**[14] is Google's proprietary web-service for cloud-based machine learning prediction tools, but the maximum training data-size is restricted to 250 MB. It was mentioned in [81] that predictive models should be natively supported by databases and they continued to provide the first prototype: **Longview**. The platform **Hyracks** which is in roughly the same space as the open source Hadoop platform is introduced in [82]. It is a new partitioned-parallel software platform designed to run data-intensive computations on large shared-nothing clusters of computers.

A compiler like **DryadLINQ**[83] translates LINQ programs into a set of jobs that can be executed on the Microsoft Dryad platform. **Dryad** [84] belongs to the dataflow abstraction where parallel computation is represented as a directed acyclic graph with data flowing along edges between vertices. Dryad runs the applications by executing the vertices of this graph on a set of available computers and communicates through files, TCP pipes and shared-memory FIFOs. As a .NET extension, LINQ provides declarative programming for data manipulation. There is a large vector library built on top of DryadLINQ. Machine learning algorithms can be implemented in C# using the simple mathematical primitives and data types provided in this library. However, the programmer is left with the task of indentifing the data parallel components of an algorithm and expressing them in DryadLINQ expressions.

The following shows some examples for parallelizing graph mining algorithms. Being a Hadoop-based library, **Pegasus** [85] implements a class of graph mining

[14]Google Prediction API. https://developers.google.com/prediction/.

algorithms that can be expressed with repeated matrix-vector multiplications. The framework called **Graphlab** [86] aims to provide an expressive and efficient high-level abstraction to satisfy the needs of a broad range of machine learning algorithms. Although it works well on certain types of machine learning tasks, it uses low-level API and its focus on graphs makes it less applicable to more traditional machine learning problems.

The very recent developments also saw some systems try to address the shortcomings of MapReduce such as: inefficiency in handling iterative and interactive algorithms; each algorithm should be written in the Map and Reduce functions. Our thesis will concentrate on the first shortcoming of MapReduce.

Similar to DryadLINQ in the *Scala* [15] programming language, a convenient language-integrated programming interface is provided by *Spark*. **Spark** [87] focuses on the applications that reuse a working set of data across multiple parallel operations. It also retains the scalability and fault tolerance of MapReduce. *Spark* is built on a data abstraction called Resilient Distributed Datasets (**RDDs**) [88]. RDDs is proposed to let programmers perform in-memory computations on large clusters in a fault-tolerant manner. This abstraction is motivated by two types of applications: iterative algorithms and interactive data mining tools that current computing frameworks handle inefficiently.

As the cost of implementing machine learning algorithms as low-level MapReduce jobs on varying data and machine cluster sizes can be prohibitive, **SystemML** [89] is proposed to express machine learning algorithms in a higher-level language which are

[15]Scala. http://www.scala-lang.org.

compiled and executed in a MapReduce environment. **MLbase** [90] is a scalable machine learning system for both end-users and machine learning researchers. Two main functions are provided by MLbase: it enables machine learning researchers to implement scalable machine learning methods without requiring deep system knowledge; it provides a novel optimizer to select and dynamically adapt the choice of learning algorithms. Similar to SystemML, MLbase applies simple declarative way to specify machine learning tasks.

2.3 Conclusion

In this chapter, we presented some of the important reasons to choose ensembles in practice and the statistical, computational, and representational reasons to construct ensemble systems in theory. However, the success of good ensemble systems lies in the technique to design diverse base learners. Two categories of methods encouraging diversity implicitly and explicitly both work well in practice. The Adaboost algorithm encourages diversity explicitly, while the meta-learning algorithm injects diversity implicitly. Many ensemble methods were developed in the recent years, but there have been no algorithms that tried to integrate meta-learning and Adaboost. We will propose this algorithm in chapter 3.

A dataset becomes large scale when its size starts to become an issue to take into account explicitly in the data mining process. To process big datasets and extract information from them, the open source Hadoop platform was developed and became very popular in industry. This platform uses a programming prototype called MapReduce

to implement parallel machine learning algorithms. To move data in and out to different computing nodes, the HDFS is required. Understanding each component's structure and work flow is helpful for us to comprehend the parallel methodology we are going to demonstrate in chapter 4. Moreover, many parallel computing systems were also proposed in the very recent years. Some applies the traditional way for parallelization and the difficulties involved in programming limits their popularity. Others proceed to employ higher level declarative models than MapReduce and resolve the shortcoming brought by MapReduce. In chapter 4, we will also focus on the disadvantage of MapReduce and propose a new approach to eliminate it.

Chapter 3

Meta-boosting Algorithm

This chapter introduces a new ensemble learning algorithm: the Meta-boosting algorithm. Section 3.1 gives some introduction and motivations for proposing this algorithm. Section 3.2 presents the detailed models and algorithms. Section 3.3 shows the experimental setup and results. Section 3.4 offers some discussions on the computation complexity of this algorithm. Section 3.5 gives some conclusions.

3.1 Introduction

Empirical studies have shown that a given algorithm may outperform all others for a specific subset of problems, but there is no single algorithm that achieves best accuracy for all situations [91]. Therefore, there is growing research interest in combining a set of learning algorithms into one system called ensemble method. This kind of multiple learning systems try to exploit the local behaviors of different base learners to enhance the accuracy of the overall learning system [92]. Such systems are realized by combining the individual decisions of a set of classifiers. Since combining classifiers having similar decisions makes no difference, classifiers with different decisions are often chosen as the base-level classifiers.

There are two different approaches to creating multiple systems: homogeneous classifiers—which use the same algorithm over diversified data sets; heterogeneous classifiers—which use different learning algorithms over the same data [93]. Examples of homogeneous classifiers are bagging [45], random subspace method [47] and boosting [94]. Bagging creates random training sets, while the random subspace method selects random subsets of features from the feature set. In boosting, the classifiers are trained in series with training instances having different weight distributions. One of the examples for heterogeneous classifiers is Meta-learning. As we have introduced in section 2.1.7, meta-learning learns from the predictions of the base classifiers (level-0 generalizer) applying a learning algorithm (level-1 generalizer). This algorithm is adopted to learn how to integrate the learned classifiers.

To combine multiple predictions from separately learned base classifiers, meta-learning has two distinct strategies: arbitration and combining. The arbiter which is used in arbitration is another classifier. It may choose a final outcome based on its own prediction and the other classifiers' predictions. Combining refers to the use of knowledge about how classifiers behave with respect to each other. Predictions from the base classifiers are combined by learning the relationship or correlation between these predictions and the correct prediction.

Our meta-boosting algorithm is inspired by the AdaBoost Dynamic algorithm that was introduced in [95]. As the conventional boosting algorithm only boosts one weak learner, AdaBoost Dynamic tries to improve the AdaBoost.M1 algorithm by calling different weak learners inside the boosting algorithm. It is concluded that for a large

majority of the datasets, the performance of this algorithm is only comparable with AdaBoost.M1 for the best single weak learner. This may be caused by the reason that different types of weak learners build different decision boundaries, and they may have bad influences on each other for different iterations.

In our meta-boosting algorithm, instead of combining different weak learners using weighted majority voting inside the boosting algorithm, we combine the predictions of the boosting algorithm with different weak learners using a meta-learner. This is the combining strategy that is used for combining multiple predictions from different base learners in the meta-learning algorithm. Since it is an ensemble system, this algorithm's performance is expected to improve. Furthermore, it can be readily parallelized using Hadoop MapReduce.

3.2Proposed Method

3.2.1 Combining AdaBoost Using Meta-learning

In this chapter, we introduce a new algorithm by combining homogeneous and heterogeneous classifiers together, in an effort to increase accuracy.

It was proposed that by calling a different weak learner in each of the iterations of AdaBoost.M1, this algorithm could be improved [95]. Adaboost Dynamic, the method presented in [95], applied different training sets to train different algorithms while conventional methods usually use either the same algorithm trained on different training

datasets or different algorithms trained on the same training datasets.

Instead of combining the results of different weak learners inside the AdaBoost.M1 algorithm as done by Adaboost Dynamic, here, we propose to combine them outside this algorithm applying a combiner strategy. Since the AdaBoost.M1 algorithm performs classification using different training datasets and all the AdaBoost.M1 algorithms with different weak learners are combined in the second level to make a final decision, our method is also a combination of multiple datasets with multiple classifiers.

To make the final decision, we choose meta-learning[96] [97]. Fig.3.1 describes the different stages for the Meta-boosting ensemble algorithm: a combination of AdaBoost algorithm and meta-learning algorithm. First, different base classifiers (level-0 classifiers) are obtained by training the AdaBoost.M1 algorithm with different weak learners using the initial (base-level) training datasets. Second, predictions are generated by the learned classifiers on a separate validation dataset. Third, a meta-level training set is collected from the true labels of the validation dataset and the predictions generated by the base classifiers on the validation dataset. For multi-class problems, we choose the prediction for the class which has the highest probability. Lastly, the final classifier (meta-classifier) is trained from the meta-level training set.

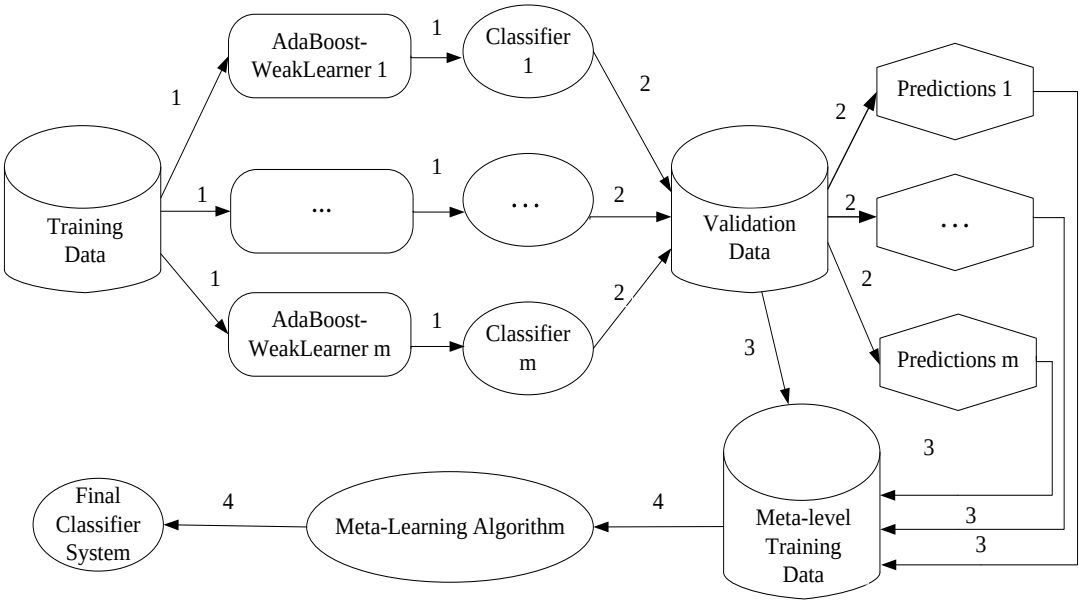

Fig.3.1. The model of Meta-boosting algorithm

One obvious difference between the meta-boosting model and the original meta-learning model introduced in section 2.1.7 is that our model uses a separate validation data to generate the predictions for the second-level learning. In contrast, the original meta-learning model doesn't have a separate validation data. It uses all the training data to generate the predictions to be used in the meta-level. The advantage for using a separate validation data in our model is improved one time training efficiency, especially for big datasets. The detailed process is discussed in the following section.

3.2.2 Detailed Procedures and Algorithm

In order to prepare the original dataset to be used in the ensemble learning process, for the instances in the dataset, we first split it into 10-fold stratified cross validation (CV) datasets. Then for each Fold 0~Fold 9 we again split it into 10-fold stratified CV data sets. The split datasets are training and validation datasets for each of the base learners. This

process is different from the original meta-learning algorithm in section 2.1.7 which split each Fold 0~Fold 9 only into two parts using one part for training the base learners and the other part for generating predictions. Details of this procedure are illustrated in Fig.3.2.

Then, how these data will be used in the whole process is presented in Fig.3.3. Here, we take the data: Train 00, Valid 00 and Test 0 from Fig.3.2 for an example. We assume that there are only three base learners in our system.

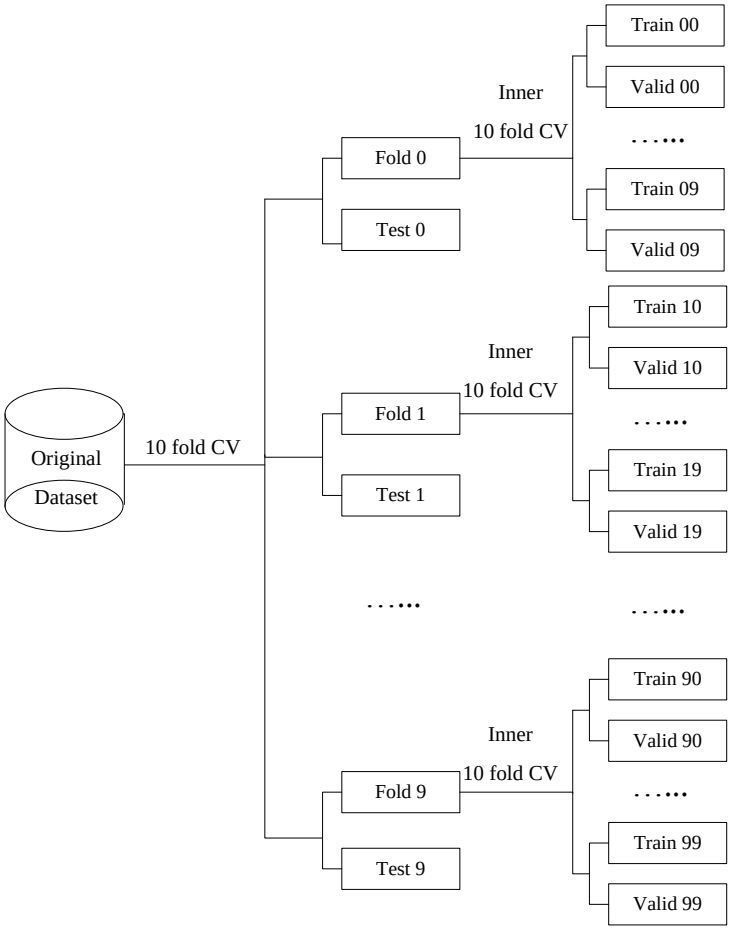

Fig.3.2. Data preparation procedure.

Training process: The meta-level training data consists of the predictions of the base classifiers on the validation dataset Valid 00 and the true labels of Valid 00. When these training data are used to train the meta-learning algorithm, compared to the

common classification problem, the predictions are considered as features while true labels of Valid 00 are considered as classes. Here, the predictions are level-0 class probabilities. Unlike the original meta-learning training process, we don't apply the whole training & validation datasets to train the base learners and obtain the final base classifiers. The base classifiers which would be applied in the test process are the ones generated using only the training datasets. This process is a special case of Cascade Generalization [98].

Test process: Once the meta-classifier is obtained, it will be tested on the meta-level test data. To get this data, each base classifier obtained in the training process is first tested on all instances from Test 0. The predictions from each of them will be gathered to become the meta-level test data. Finally, the accuracy of the whole ensemble method will be calculated by comparing the predicated labels of the meta-classifier and the true labels of Test 0. It should be noted that in order to get reliable accuracy results, the process in Fig.3.3 should be repeated 10 times for each test data since each test data corresponds to 10 pairs of training and validation datasets. After obtaining 10 times balanced accuracy of one test data, the final accuracy for the meta-boosting algorithm on the original dataset is calculated by averaging the balanced accuracy of all the 10 different test data.

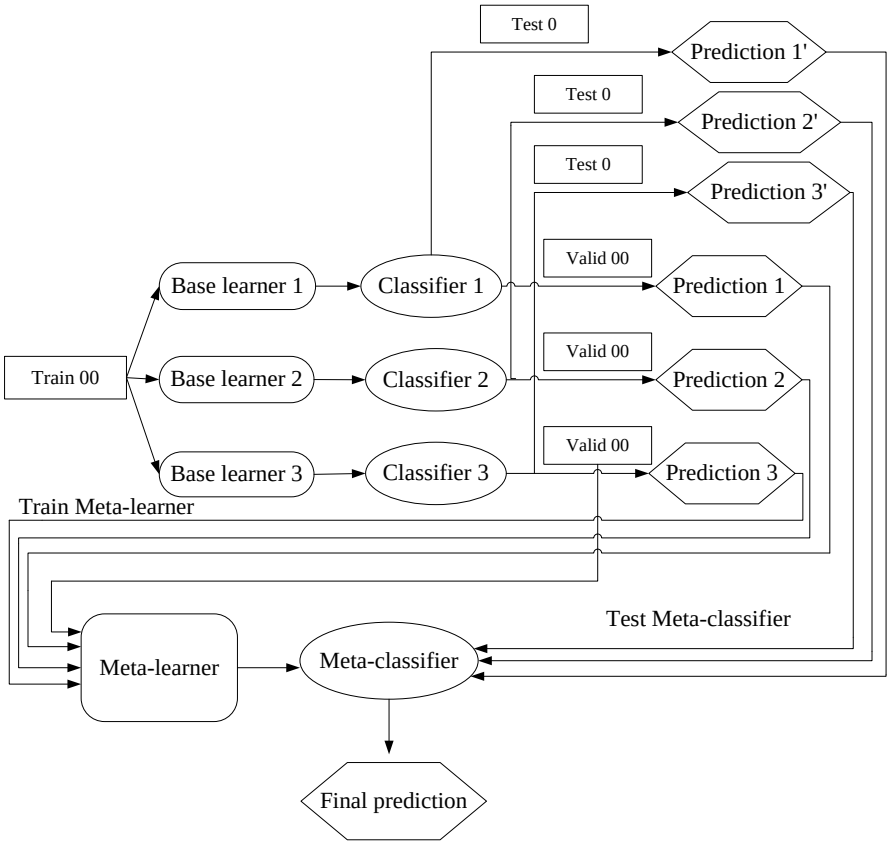

Fig.3.3. Implementation process for one time training, validation & test.

Table 3.1 shows the pseudo code for the proposed Meta-boosting algorithm. In the whole process, two functions are called: f_1 andf_2. The first one gets the predicted probabilities of each base learner and uses all the predictions and the true labels in the validation dataset to train the meta-learner. This function returns the trained model. The second function gets the predictions of all the base learners on the test data and uses these predictions to test the already trained ensemble model. The output accuracy is obtained by comparing the test results with the true labels of the test data set.

Table 3.1. Pseudo code for Meta-boosting algorithm

Input:

- S_1 , training data: sequence of k_1 examples $S_1 = \{(x_1, y_1), \dots, (x_{k_1}, y_{k_1})\}$with labels$y_i \in Y = \{y_1, \dots, y_c\}$;

- S_2, validation data: sequence of k_2 examples $S_2 = \{(x_1, y_1), \dots, (x_{k_2}, y_{k_2})\}$ with labels $y_i \in Y = \{y_1, \dots, y_c\}$;
- S_3, test data: sequence of k_3 examples $S_3 = \{(x_1, y_1), \dots, (x_{k_3}, y_{k_3})\}$ with labels $y_i \in Y = \{y_1, \dots, y_c\}$;
- n, number of base learners;
- T, number of iterations in AdaBoost.M1 algorithm.

Output:

- y, accuracy of ensemble learning

Calls:

- L, List of the base learners
- f_1, function which returns the trained meta-level model
- f_2, function which returns the test result (accuracy) on the meta-level classifier

Train ensemble model

1. Call f_1
2. For $i = 1, \dots, n$
3. Initialize $\mathrm{D}_1(\mathrm{j}) = \frac{1}{\mathrm{k}_1}$, $\mathrm{j} = 1, \dots, \mathrm{k}_1$
4. Call $L[i]$
5. Do for $t = 1,2, \dots, T$:
6. Select a training data subset S_t , drawn from D_t.
7. Train $L[i]$with S_t
8. Get back hypothesis $h_t: X \to Y$

9. Calculate the error of $h_t: \varepsilon_t = \sum_{i:h_t(x_i)\neq y_i} D_t(j)$

If $\varepsilon_t > \frac{1}{2}$, abort.

10. Set $\beta_t = \varepsilon_t/(1-\varepsilon_t)$

11. Update distribution

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} \beta_t & \text{if } h_t(x_i) = y_i \\ 1 & \text{otherwise} \end{cases}$$

where$Z_t = \sum_j D_t(j)$is a normalization constant

12. End for

13. End for

14. For $i = 1, \ldots, n$

15. For any unlabeled instance $x$in S_2, obtain total vote received by each class

$$V_j = \sum_{t:h_t(x)=y_j} \log\frac{1}{\beta_t}, j = 1, \ldots, C$$

16. Choose the class that receives the highest total vote as the final classification; get predicted probability for this class.

17. End for.

18. Use all the predicted probabilities P_1 and all true Labels of the instances in $S_2$to train meta-learner.

19. Output trained ensemble model.

Test ensemble model

1. Call f_2

2. Use S_1 to train L (repeat 2-13 in train ensemble model)

3. Use S_3 to test trained base classifiers (repeat 14-17 in train ensemble model), get all the predicted probabilities P_2

4. Use P_2 to test the trained ensemble model

5. Output the tested accuracy of the ensemble model

3.2.3 Advantages of Our Algorithm

There are three reasons why the Meta-boosting algorithm would improve the whole system's performance. First, AdaBoost is demonstrated to have the ability of boosting weak learners, while the method of combining multiple classifiers is proven to be useful in classification. As we have introduced at the beginning of section 2.1, ensemble learning produces better overall accuracy than any of its constituent classifiers. By combining multiple classifiers using AdaBoost with different weak learners as base learners, the total accuracy will be further improved on the basis of AdaBoost algorithm. Second, in AdaBoost Dynamic the combination of different weak learners is sequential. That means the updated weight for each instance made by the previous weak learner may affect the functionality of the next one as the weak learners are trained iteratively. In our Meta-boosting algorithm, each base learner will not impair each other's performance since all the base learners work in parallel. That is, they are independent from each other before their predictions on the test data are combined. This is one of the advantages we secure by applying the meta-learning schema to combine base classifiers. In meta-learning, the combination is performed on the predictions of the base classifiers not on the base classifiers themselves and all the base classifiers are treated as black boxes.

We will explain the third reason in terms of bias and variance. Ensemble errors are

often decomposed into bias and variance terms to find out the reasons why an ensemble classifier outperforms any of its components. From Kohavi and Wolpert's definition [43], bias, variance, and noise at an instance x are expressed as

$$\text{bias} = \frac{1}{2}\sum_{c_i}(P(c_i|x) - P_h(c_i|x))^2 \tag{3.1}$$

$$\text{variance} = \frac{1}{2}\left(1 - \sum_{c_i}(P_h(c_i|x))^2\right) \tag{3.2}$$

$$\text{noise} = \frac{1}{2}\left(1 - \sum_{c_i}(P(c_i|x))^2\right) \tag{3.3}$$

here$h$ is a classifier randomly chosen, $c_i \in C$ where C is the set of possible labels for an instance x. The true value of the posterior probability for the class c_i given x is $P(c_i|x)$, and the predicted probability for a given class c_i assigned by classifier h is$P_h(c_i|x)$. By definition, $\sum_i P(c_i|x) = 1$ and$\sum_i P_h(c_i|x) = 1$.

Bias is the measure of the difference between the true distribution and the guessed distribution. Variance represents the variability of the classifier's guess regardless of the true distribution. Noise demonstrates the variability of the true distribution regardless of the guess of the classifier. Bias is related with under fitting the data, while variance is associated with over fitting [99].

Using some simple derivations, the probability of error for a given x is given by

$$P(\text{error}|x) = 1 - \sum_{c_i} P(c_i|x)\, P_h(c_i|x) = \text{bias} + \text{variance} + \text{noise} \tag{3.4}$$

It was often found that very simple learners perform very well in experiments, and sometimes even better than more sophisticated ones [100] [101]. The reason for this phenomenon is that there is a trade-off between bias and variance. When more powerful learners reduce one of the bias and variance they increase the other. To overcome this shortcomings, ensembles of models are designed, which outperforms the results of a

single model. Although intensive searching for a single model is susceptible to increased variance, the ensembles of multiple models can often reduce it.

There is no general theory about the effects of boosting on bias and variance. From the experimental studies, we know that AdaBoost has much stronger reduction of variance with respect to the reduction of bias. Thus, variance reduction is the dominant effect for AdaBoost [41][102][103].

For meta-learning the goal of using the estimated probabilities of those derived from base classifiers as the input to a second learning algorithm is to produce an unbiased estimate. Although there is no detailed theoretical analysis of meta-learning's effect on bias and variance, it is often regarded as a bias reducing technique [104].

Therefore, the third advantage of our algorithm is that it has both reduced variance and bias as it is an integration of AdaBoost and meta-learning.

3.3Experimental Setup and Results

The experiments were performed on 17 datasets taken from the UCI repository [105]. We selected datasets from the repository that have different numbers of training instances and features. A summary of the datasets used is presented in Table 3.2.

As choosing all the base learners may increase the dimensionality of the meta-dataset and choosing some of the base learners yields results similar to those obtained when choosing all the base learners, here, we chose three common base learners that each use a distinct learning strategy: Ada_NB—AdaBoost.M1 with Naïve Bayes as

the learner, Ada_KNN—AdaBoost.M1 with k-nearest neighbor as the learner and Ada_tree—AdaBoost.M1 with J48 as the learner. The meta-learner is multi-response linear regression (MLR) which was proven to be the best meta-learner [106].

Table 3.2. Datasets used in the experiments

Name	# of Instances	# of Attributes	# of Classes
Contact-lenses	24	5	3
Post-operative	90	9	3
Ionosphere	351	35	2
Diabetes	768	9	2
Blood	748	5	2
Crx	690	16	2
Au1	1000	21	2
Soybean	683	36	19
Iris	150	5	3
Segment	1500	20	7
Breast tissue	106	10	6
CTG	2126	23	10
Car	1728	7	4
Cmc	1473	10	3
Glass	214	10	6
Zoo	101	18	7
Balance-scale	625	5	3

3.3.1 Comparison Between Meta-boosting and Base Learners

In Table 3.3, we present the percentage of correctly classified instances for each of the base learners and the Meta-boosting algorithm. The presented results were obtained with 10 fold cross validation. It is obvious from the results that our proposed ensemble method improves the accuracy of the base learners.

It can be seen from this table that meta-boosting is the best for 14 datasets, Ada_NB is the best for 2 datasets and Ada_KNN is the best for 1 dataset. As Friedman's Test is a non-parametric statistical test for multiple classifiers and multiple domains, we performed this test on the results in table 3.3. The details of Friedman's Test and the following Nemenyi's post-hoc test can be found in [107]. The null hypothesis for this test is that all the classifiers perform equally.

The result for the Friedman's test are Friedman chi-squared=23.3963, df=3, p-value=3.339e-05. As the critical values for the chi-square distribution for $k = 4$ and $n = 17$ is $\chi_C^2 = 7.8$for a 0.05 level of significance for a single-tailed test, while 23.3963 is larger than 7.72, we can reject the hypothesis. The summarization of these results is shown in table 3.4.

Table 3.3. The accuracy of all the base learners and meta-boosting

(method that has the best performance for a given dataset is marked in bold)

Dataset	Meta-boosting	Ada_NB	Ada_KNN	Ada_tree
Contact-lenses	**83.33**	70.83	79.17	70.83
Post-operative	**71.11**	63.33	60.00	58.88
Soybean	**94.58**	92.82	91.06	92.82
Ionosphere	**94.02**	92.02	86.32	93.16
Diabetes	**76.30**	76.17	70.18	72.39
Blood	76.07	**77.13**	69.25	76.07
Crx	**87.53**	81.59	81.15	84.20
Au1	**76.00**	72.80	66.70	70.80
Iris	**95.33**	93.33	**95.33**	93.33
Segment	**98.33**	81.06	96.20	97.46
Breast tissue	70.82	70.75	**71.69**	69.80
CTG	**90.78**	77.23	83.96	90.45
Car	**96.82**	90.16	93.52	96.12
Cmc	**51.93**	50.78	44.19	50.78
Glass	**75.56**	49.06	70.56	74.29
Zoo	96.18	**97.02**	96.04	95.04
Balance-scale	**93.28**	91.04	86.56	78.88
Average accuracy	**84.00**	78.06	78.93	80.31

Table 3.4. Friedman's test results

χ_F^2	df	p	χ_C^2
23.3963	3	3.339e-05	7.8
$\chi_F^2 > \chi_C^2$ hypothesis rejected			

Following Friedman's test, in order to determine whether Meta-boosting's result is significantly better than its base learners, we applied Nemenyi's post-hoc test on the results in table 3.3. If we let 1, 2, 3 and 4 represent the algorithms meta-boosting, Ada_NB, Ada_KNN and Ada_tree respectively, then we have $|q_{12}| = 58.71$, $|q_{13}| = 73.4$, $|q_{14}| = 62.1$. As $q_\alpha = 3.79$ for $\alpha = 0.05$, df=48, number of groups $k = 4$, $q_\alpha/\sqrt{2} = 2.68$. Since the absolute value of q_{12}, q_{13}, q_{14} are all larger than 2.68, the null hypothesis that meta-boosting performs equally with Ada_NB, Ada_KNN and Ada_tree is rejected. Therefore, we conclude that meta-boosting performs significantly better than all the base learners. These results are also presented in table 3.5.

Table 3.5. Nemenyi's post-hoc test

q_{12}	q_{13}	q_{14}	$q_\alpha/\sqrt{2}$
58.71	73.4	62.1	2.68
$q_{12} > q_\alpha/\sqrt{2}$: Meta-boosting has better performance than Ada_NB $q_{13} > q_\alpha/\sqrt{2}$: Meta-boosting has better performance than Ada_KNN $q_{14} > q_\alpha/\sqrt{2}$: Meta-boosting has better performance than Ada_tree			

3.3.2 Comparison Between Meta-boosting and AdaBoost Dynamic

To compare our results with AdaBoost Dynamic [95], we used 13 of the 18 datasets in [95]. We omitted the datasets which are not the same as the original ones at UCI repository. Table 3.6 shows the comparison between our algorithm and the AdaBoost Dynamic in [95]. It can be seen that our algorithm wins 10 out of 13 datasets.

To test the significance of these results, we then applied Wilcoxon's signed rank test [107] and the result is $p = 0.03271$. As $p < 0.05$, at 0.05 significance level, we reject the hypothesis that meta-boosting performs equally with AdaBoost Dynamic [95].

3.4 Discussion

It can be anticipated that one disadvantage of our Meta-boosting algorithm is its higher computation complexity compared to each of the base learners. According to the Meta-boosting algorithm, to obtain reliable accuracies, there are 100 passes of the training, validation and test process. For each training and validation process, three AdaBoost.M1 algorithms are involved and each of them performs 10 fold cross validation themselves. Therefore the time spent in obtaining the training model of Meta-boosting is much more than the sum of the training times of all the base learners. In order to prove this, we list the training time for Meta-boosting and its three base learners in table 3.7.

In table 3.7, it takes much more time for the Meta-boosting algorithm to construct

the training model. It may not be a big problem for the relatively small datasets which we have chosen in this chapter's experiments. However, when it comes to big datasets and also high requirements on accuracies, the Meta-boosting algorithm just like other common machine learning algorithms fails in this case. In order to settle this problem, another approach is presented in chapter 4 to parallelize this Meta-boosting algorithm.

Table 3.6**.** Comparisons between Meta-boosting and AdaBoost Dynamic

(method that has the best performance for a given dataset is marked in bold)

Dataset	Meta-boosting	AdaBoost Dynamic
Contact-lenses	**83.33**	74.17
Post-operative	**71.11**	56.11
Soybean	**94.58**	93.45
Blood	76.07	**77.93**
Crx	**87.53**	81.42
Au1	**76.00**	72.25
Iris	**95.33**	95.13
Segment	**98.33**	95.81
Breast-tissue	**70.82**	64.28
Car	96.82	**99.14**
Cmc	51.93	**54.08**
Zoo	**96.18**	95.66
Balance-scale	**93.28**	89.71
Average accuracy	**84.00**	80.70

Table 3.7. Computation complexity comparison (“ms”: milliseconds).

Dataset	Meta-boosting	Ada_NB	Ada_KNN	Ada_tree
Contact-lenses	125ms	0ms	0ms	0ms
Post-operative	421ms	0ms	31 ms	0ms
Soybean	39327 ms	531 ms	4718 ms	203 ms
Ionosphere	5734 ms	218 ms	47 ms	313 ms
Diabetes	4391 ms	110 ms	78 ms	282 ms
Blood	12593 ms	47 ms	1375 ms	31 ms
Crx	5203 ms	62 ms	296 ms	172 ms
Au1	13343 ms	281 ms	344 ms	594 ms
Iris	360 ms	16 ms	0ms	16 ms
Segment	21874 ms	860 ms	266 ms	906 ms
Breast tissue	547 ms	32 ms	0ms	31 ms
CTG	196360ms	1391 ms	21234 ms	1875 ms
Car	5563 ms	78 ms	343 ms	94 ms
Cmc	54062 ms	62 ms	5078 ms	438 ms
Glass	1171 ms	31 ms	0ms	79 ms
Zoo	188 ms	15 ms	0ms	16 ms
Balance-scale	2578 ms	63 ms	46 ms	125ms

3.5 Conclusions

In this chapter, we presented a new method for constructing ensembles of classifiers by combining AdaBoost.M1 algorithms with the meta-learning framework. The comparison of our algorithm and each of the base learners was performed on 17 datasets from the UCI domain. These experimental results demonstrate that our algorithm outperforms all its base learners. Friedman's test [107] and Nemenyi's post-hoc test [107] further confirmed that the superior performance of our algorithm over all the base learners is statistically significant, although it has higher computational complexity. As meta-boosting improved the results further on the basis of the boosting algorithm, our method actually enhances the boosting results. We also compared our results with AdaBoost Dynamic [95], and both the experimental accuracies and statistical Wilcoxon's signed rank test [107] demonstrated that our algorithm obtains better performance. To resolve the disadvantage of high computation complexity, parallel computing techniques need to be considered.

Chapter 4

Parallel Meta-boosting Algorithm with MapReduce

This chapter presents our parallel meta-learning algorithm (PML) which reduces the computation complexity encountered with in-memory meta-boosting algorithm and also provides a framework to settle the problem of lacking of iterations in the original MapReduce paradigm. This chapter is organized as follows: section 4.1 discusses the weakness of the current MapReduce framework and our motivations for doing this research; section 4.2 illustrates some of the proposed related work to tackle this shortcoming in MapReduce; section 4.3 presents the detailed procedure about our proposed algorithm: Parallel meta-learning (PML); section 4.4 demonstrates the performance of PML in terms of error rates and speedup; section 4.5 concludes the entire chapter.

4.1 Introduction

Nowadays, we are moving from the Terabytes to the Petabytes age as a result of the rapid expansion of data. The potential values and insights which could be derived from massive data sets have attracted tremendous interest in a wide range of business and scientific applications [108] [6][8]. It is becoming more and more important to organize

and utilize the massive amounts of data currently being generated. However, when it comes to massive data, it is difficult for current data mining algorithms to build classification models with serial algorithms running on single machines, not to mention accurate models. Therefore, the need for efficient and effective models of parallel computing is apparent.

Fortunately, with the help of MapReduce[8] [109] [110] [111]infrastructure, researchers now have a simple programming interface for parallel scaling up of many data mining algorithms on larger data sets. It was shown [13] that algorithms which fit the Statistical Query model [112] can be written in a certain "summation form". They illustrated 10 different algorithms that can be easily parallelized on multi-core computers applying MapReduce paradigm. In 2009, Google proposed PLANET: a framework for large-scale tree learning using a MapReduce cluster[16]. Their intention in building PLANET is to develop a scalable tree learner which could achieve comparable accuracy performance as the traditional in-memory algorithm and also able to deal with bigger datasets. PLANET is used to construct scalable classification and regression trees, even ensembles of these models. It realizes parallelization by dividing tree learning into many distributed computation with each implemented with MapReduce.

Although MapReduce handles large-scale computation, it doesn't support iteration. There are no loop steps available in Hadoop. To have loops, an external driver is needed to repeatedly submitting MapReudce jobs. Since each MapReduce job works independently, in order to reuse data between MapReduce jobs, the results generated by a former MapReduce job are written to the HDFS and the next MapReduce job which

needs these information as inputs reads these messages for HDFS. This operation doesn't have the benefits that the caching system can bring for the in-memory computation. Moreover, owing to data replication, disk I/O, and serialization, the approach for creating loops inside the original version of Hadoop causes huge overheads. The time spent in this process may sometimes occupy a major part in the total execution time. This is a critical weakness. For instance, boosting and genetic algorithms naturally fit into an iterative style and thus cannot be exactly expressed with MapReduce. Realizing these algorithms' parallelization requires special techniques [113] [114]. However, even if the computations can be modeled by MapReduce, when there are iterations in a machine learning algorithm, the execution overheads are substantial [115] [116].

As introduced in section 2.1.7, Meta-learning is loosely defined as learning from information generated by learner(s) and it is applied to coalesce the results of multiple learners to improve accuracy. One of the advantages of meta-learning is that individual classifiers can be treated as black boxes and in order to achieve a final system, little or no modifications are required on the base classifiers. The structure of meta-learning makes it easily adapted to distributed learning.

In this chapter, we harness the power of meta-learning to avoid modifying individual machine learning algorithms with MapReduce. Hence, those algorithms which require iterations in their model can be more easily parallelized utilizing the meta-learning schema than altering their own internal algorithms directly with MapReduce.

4.2 Related Work

Pregel [117] is a concept similar to MapReduce. The difference is that it provides a natural API for distributed programming framework which aims at graph algorithms. It also supports iterative computations over the graph. This is an attribute which MapReduce lacks. In Pregel computations, *supersteps*, a sequence of iterations is adopted. With *supersteps*, a vertex can receive information from the previous iteration and also send information to other vertices that will be received at a next *superstep*. However, Pregel focuses on graph mining algorithms, while our algorithm has a more general application.

As a modified version of the Hadoop MapReduce, **HaLoop** is presented in [118] to offer support to the iterative programs which are absent in the original MapReduce framework. It was mentioned that iterative computations were needed for Page Rank [119], recursive relational queries [120], clustering such as *k*-means, neural-network analysis, social network analysis and so on. When doing these analyses, there are many iterations in the algorithm until some convergence or abort conditions are met. They also mentioned that manually implementing iterative programs by multiple MapReduce jobs utilizing a driver program can be problematic. Therefore, they propose to automatically run a series of MapReduce jobs in loops by adding a loop control module to the Hadoop master node. However, HaLoop only supports specific computation patterns.

Twister [121] is also an enhanced version of MapReduce which supports iterative

MapReduce computations. In the original MapRedcue, in order to realize iterations a set of Map and Reduce tasks are called. To communicate between each round of MapReduce job, a lot of loading and accessing activities are repetitively required. This causes considerable performance overheads for many iterative applications. In *Twister*, to reduce these overheads and achieve iterative computation, a publish/subscribe messaging infrastructure is proposed for communication and data transfers. Moreover, there are long running map/reduce tasks with distributed memory caches. Basically, it is a stream-based MapReduce framework. However, this streaming architecture between map and reduce tasks suffers from failures. Besides, long running map/reduce tasks with distributed memory caches is not a good scalable approach for each node in the cluster having limited memory resources.

As a scalable machine learning system, **Vowpal Wabbit** (VW) [122] is implemented with the All Reduce function (which originates in MPI) in Hadoop for the sake of accurate prediction and short training time in an easy programming style. By eliminating re-scheduling between iterations and communicating through network connections directly, fast iterations are optimized. Moreover, the map and reduce tasks are sped up via a cache aware data format and a *binary aggregation tree* respectively.

Resilient Distributed Datasets (RDDs) [88] is a distributed memory abstraction intended for in-memory computations on large clusters. It is implemented in the system *Spark* [87]. RDDs addresses the problem of data reuse of intermediate results among multiple computations, which cannot be handled efficiently by current proposed cluster computing frameworks such as MapReduce and Dryad [84]. The advantage of RDDs

compared to *Pregel* and *HaLoop* is that an abstraction is proposed for more general use such as the application of running ad-hoc queries across several datasets which are loaded into memory. RDDs also absorb the merits from other frameworks. These include in-memory storage of specific data, control of data partitions to reduce communications and recovery from failures efficiently. However, the optimization of RDDs is specialized for in-memory computation only.

Iterative MapReduce [123] is an extension of the MapReduce programming paradigm, which claims to be the most advanced framework for supporting iterative computations. In *Spark*, the programmer has to make systems-level decisions like to recognize what data to cache in the distributed main memory. However, sometimes the programmer may lose track of performance-related parameters in large public clusters where these parameters keeps changing. To tackle this problem which happens in *Spark*, *Iterative MapReduce* applies the ideas from database systems to eliminate the low-level systems considerations via the abstraction brought by the relational model. Furthermore, it also provides a way for DBMS-driven optimization.

However, the introduced approaches for dealing with the problem of lacking iterations in MapReduce either require substantial change of the original MapReduce framework or needs designing new systems. In contrast, our method proposed in this chapter is much simpler in that it relieves the necessity of considering new models and the complexity of implementing them. Moreover, our approach is one time configuration and unlimited times of repetitive usage, which is much more advantageous than those approaches which have to alter the design for different algorithms.

4.3 Framework

In this section, we propose our framework called parallelized meta-learning(PML) which is implemented with the programming model of MapReduce on Hadoop. As has introduced in section 2.1.7, the process of building and testing base classifiers of meta-learning can be executed in parallel, which makes meta-learning easily adapted to distributed computation.

A. Parallel Meta-learning with MapReduce

The in-memory meta-learning presented in section 2.1.7 and the distributed meta-learning which we are going to provide in this section have the following differences. In in-memory meta-learning, the last step in the training process is to generate the final base classifiers by training all the base classifiers on the same whole training data. In contrast, the distributed meta-learning has training and validation datasets. Since each training data are very big and split across a set of computing nodes, there is no way that the final base classifiers can be trained on the whole training datasets. In the end, each computing node retains their own base classifiers obtained through training on their share of training data.

Our parallel Meta-learning algorithm (PML) includes three stages: Training, Validation and Test. For the base learning algorithms which are shown in fig.3.1, we used the same machine learning algorithm. The number of base learning algorithms is equal to that of the mappers. It is also feasible to apply different machine learning algorithms for

different mappers. Here we focus on using the same algorithm and we applied map functions without reduce functions.

PML Training: in the training process, a number of base classifiers are trained on the training data. This task requires a number of mappers. The pseudo code is given in table 4.1 and the detailed procedure is described as follows.

Let $D_{n^m}^m = \{(x_1^m, y_1^m), (x_2^m, y_2^m), \dots, (x_{n^m}^m, y_{n^m}^m)\}$ be the training data assigned to the mth mapper where $m \in \{1, \dots, M\}$ and n^m is the number of instances in the mth mapper's training data. The map function trains the base classifier on their respective split data sets (line 2). Each base classifier is built in parallel on idle computing nodes selected by the system. The trained model is then written to the output path (line 3). Finally, all the trained base models are collected and stored in the output file (i.e., the HDFS) (line 5).

Table 4.1. Algorithm 1: PML Training.

Algorithm 1. PML Training $(D_{n^1}^1, D_{n^2}^2, \dots, D_{n^M}^M, BL)$

Input: The training sets of M mappers $(D_{n^1}^1, D_{n^2}^2, \dots, D_{n^M}^M)$

The selected base learning algorithm (BL)

Output: The trained base classifiers $(B^1, B^2, \dots, B^M)$

Procedure:

1: **For** $m \leftarrow 1$ to M **do**

2: $B^m \leftarrow BL.build(D_{n^m}^m)$

3: Output(m, B^m) // output key and value pairs

4: **End for**

5: Return $(B^1, B^2, \dots, B^M)$

PML Validation: in the validation process, the meta-level training data is obtained and the meta-learner is trained based on these data. The pseudo code is shown in Table

4.2 and the steps are described as follows:

- Generate meta-level training data: the validation data is split across P mappers. Each mapper will execute the process of testing all the base classifiers (line 3). Instead of generating the predicted labels, the predicted probabilities PD^m for each of the class from all the base classifiers are collected and put together (line 5). To get better understanding, these predictions are just the same as all the elements shown in equation 2.7. Then, the meta-level training instances MI^p are made up by all the base classifiers' prediction distributions PD (as attributes) and true labels of the validation data $l(D_{k^p}^p)$ (as labels) (line 7). The next step is that each map function outputs the <key, value>pairs $\langle l(D_{k^p}^p), MI^p \rangle$. Here,the key is the true labels of the validation data: $l(D_{k^p}^p)$while value is the meta-instances MI^p(line 8). At this point, the map tasks are finished and the total meta-instances MI are organized together from all the outputs of the mappers (line 12).
- Train meta-learner: let the total number of classes beNC. For each class, new meta-instances: $filter^{nc}(MI)$ are generated by selecting the prediction probabilities corresponding to this class from MI (as attributes) and setting the class label to be 1 if it belongs to this class, or 0 if not. The meta-classifier is then trained on these new instances and the trained model for each class is set as h^{nc} (line 15). As a result, each class will have its own built meta-classifier. The number of meta-classifiers equals to the number of classes. In the end, all the trained meta-classifiers are sorted

together according to the classes and saved (line 17).

Table 4.2. Algorithm 2: PML Validation

Algorithm 2. PML Validation $\left(D_{k^1}^{1}, D_{k^2}^{2}, \dots, D_{k^P}^{P}, B^1, \dots, B^M\right)$

Input: The validation sets of P mappers

$$\left(D_{k^1}^{1}, D_{k^2}^{2}, \dots, D_{k^P}^{P}\right)$$

The base classifiers $(B^1, \dots, B^M)$

The meta-learner (h)

Output: The trained meta-classifiers $(h^1, \dots, h^{NC})$

Procedure:

1: **for**$p \leftarrow 1$ to P**do**

2:$PD \in \emptyset$

3: **for**$B^m \leftarrow B^1$toB^M**do**

4: $PD^m \leftarrow B^m.classify\left(D_{k^p}^{p}\right)$

5: $PD = PD \cup PD^m$

6: **end for**

7: $MI^p \leftarrow \left(PD \cup l\left(D_{k^p}^{p}\right)\right)$

8: Output$\left(l\left(D_{k^p}^{p}\right), MI^p\right)$// output key and value pairs

9: **end for** // end of map functions

10: $MI \in \emptyset$

11: **for**$p \leftarrow 1$ to P**do**

12: $MI = MI \cup MI^p$

13: **end for**

14: **for**$nc \leftarrow 1\ to\ NC$**do**

15: $h^{nc} \leftarrow h.build\left(filter^{nc}(MI)\right)$

16: **end for**

17: return $(h^1, \dots, h^{NC})$

PML Test: in the test process, the meta-level test data are generated and the meta-classifiers $(h^1, \dots, h^{NC})$ are tested. The pseudo code is shown in table 4.3.The following shows the processes:

- Generate meta-level test data: this part is similar to the part of generating meta-level training data in the validation process. The only difference is that the input data are the test data. This part is shown through lines 1-13.
- Test $(h^1, \dots, h^{NC})$: for each instance $MI^*[i]$ in the meta-instances MI^*, the probability $prob^{nc}$ for each class is obtained by classifying the corresponding meta-classifier h^{nc} on a new meta-instance $filter^{nc}(MI^*[i])$ (line 16). This instance is generated the same way as $filter(MI)$ in the validation process. The probabilities for all classes are then summed (line 17) and normalized (line 20). The predicted label l^i is found by seeking the class ID of the calculated normalized highest probabilities (line 22). Finally, the predicted labels for all the test instances are returned (line 24).

Table 4.3. Algorithm 3: PML Test

Algorithm 3. PML Test $\left(D_{j^1}^1, D_{j^2}^2, \dots, D_{j^Q}^Q, B^1, \dots, B^M\right)$

Input: The test sets of Q mappers $\left(D_{j^1}^1, D_{j^2}^2, \dots, D_{j^Q}^Q\right)$

The base classifiers $(B^1, \dots, B^M)$

The trained meta-classifiers $(h^1, \dots, h^{NC})$

Output: predicted labels $(l^1, \dots, l^T)$

Procedure:

1: **for** $q \leftarrow 1\ to\ Q$ **do**

2:$PD^* \in \emptyset$

3: **for**$B^m \leftarrow B^1\ to\ B^M$**do**

4: $(PD^*)^m \leftarrow B^m.classify\left(D_{j^q}^q\right)$

5: $PD^* = PD^* \cup (PD^*)^m$

6: **end for**

7: $(MI^*)^q \leftarrow \left(PD^* \cup l\left(D_{j^q}^q\right)\right)$

8: Output$\left(l\left(D_{j^q}^q\right), (MI^*)^q\right)$// output key and value

9: **end for** // end of map functions

10: $MI^* \in \emptyset$

11: **for**$q \leftarrow 1\ to\ Q$**do**

12: $MI^* = MI^* \cup (MI^*)^q$

13: **end for**

14: **for**$i \leftarrow 1\ to\ T$**do** //T is the number of MI^*

15: **for**$nc \leftarrow 1\ to\ NC$**do**

16: $prob^{nc} \leftarrow h^{nc}.classify\left(filter^{nc}(MI^*[i])\right)$

17: $sum = sum + prob^{nc}$

18: **end for**

19: **for**$nc \leftarrow 1\ to\ NC$**do**

20: $prob^{nc} \leftarrow prob^{nc}/sum$

21: **end for**

22: $l^i \leftarrow l\left(max(prob^1, \dots, prob^{NC})\right)$

23: **end for**

24: return $(l^1, \dots, l^T)$

4.4 Experiments

In this section, we compare the error rates of our PML algorithm with the meta-learning method on one single node to figure out if there would be an impact on the accuracy performance when we increase the number of computing nodes. We also compare our algorithm's performance with the parallel Adaboost algorithm proposed in [113]. The comparison is worthwhile as this parallel Adaboost algorithm directly makes Adaboost scalable using MapReduce with complicated procedures which we will introduce in the following sub-section. Finally, to demonstrate the efficiency of this MapReduce framework: PML, we also show the speedup [124] performance when we increase the number of computer nodes.

4.4.1 Parallel Adaboost Algorithm (Adaboost.PL)

Two parallel algorithms are proposed in [113]: parallel Adaboost (Adaboost.PL) and parallel Logitboost (Logitboost.PL). Since we adopt Adaboost as base learners in our experiments, here we discuss mainly their approach for parallel Adaboost. Adaboost.PL realizes parallelization inside the Adaboost algorithm first through building hypotheses with the weak learner on each of the computing nodes with partitioned data subsets, sorting these hypotheses based on the weights with ascending order, and finally merging each level's hypotheses together to obtain the corresponding iteration's hypothesis. The

detailed pseudo code is presented in Table 4.4.

Table 4.4. Pseudo code for Adaboost.PL

Algorithm 4. Adaboost.PL ($D_{n^1}^1, \dots, D_{n^M}^M, T$)

Input: training datasets dispatched to M nodes ($D_{n^1}^1, \dots, D_{n^M}^M$)

The iteration number for the outside Adaboost algorithm (T)

Output: the constructed final classifier (H)

Procedure:

1: **For** $p \leftarrow 1$ *to* M **do**

2: $Adaboost\left(D_{n^p}^p, T\right) \rightarrow H^p = \left\{\left(h^{p(1)}, \beta^{p(1)}\right), \left(h^{p(2)}, \beta^{p(2)}\right), \dots, \left(h^{p(T)}, \beta^{p(T)}\right)\right\}$

3: $Sort \rightarrow H^{p^*} = \left\{\left(h^{p^*(1)}, \beta^{p^*(1)}\right), \left(h^{p^*(2)}, \beta^{p^*(2)}\right), \dots, \left(h^{p^*(T)}, \beta^{p^*(T)}\right)\right\}$

4: **End for**

5: **For** $t \leftarrow 1$ *to* T **do**

6: MERGE$\left(h^{1^*(t)}, \dots, h^{M^*(t)}\right) \rightarrow h^{(t)}$

7: $\frac{1}{M}\sum_{p=1}^{M} \beta^{p^*(t)} \rightarrow \alpha^t$

8: **End for**

In order to understand this algorithm, one needs to refer the original Adaboost algorithm discussed in section 2.1.6. The Adaboost.PL algorithm is described as follows. Assume there are M mappers involved and each is assigned $1/M$ partition of the original datasets. At the beginning, perform the original Adaboost algorithm for each mapper, which generates T number of hypotheses and weights for T iterations (line 2). Then these sets of hypotheses and weights are sorted with ascending order based on the values of the weights (line 3). To generate each iteration's hypothesis of the final

classifier, the corresponding iteration's hypotheses from all the computing nodes are merged (line 6). Each iteration's merged classifier is a ternary classifier [125] and can be expressed as

$$h^{(t)}(x) = \begin{cases} sign\left(\sum_{p=1}^{M} h^{p^*(t)}(x)\right) & if \ \sum_{p=1}^{M} h^{p^*(t)}(x) \neq 0 \\ 0 & otherwise. \end{cases} \quad (4.1)$$

It actually obtains the result through simple majority voting among all the hypotheses generated by the mappers for each iteration. Finally, the weights for each final hypothesis $h^{(t)}$at each iteration are obtained by averaging all mappers' weights (line 7). It can be inferred from the procedure of this Adaboost.PL that the process of parallelizing Adaboost algorithm itself is complicated.

4.4.2 Experiment Settings

The experiments are deployed on Amazon EC2[16] with the instance type: m1.small. Each instance's configuration is as follows: 1.7 GHz 2006 Intel Xeon processor, 1.7 GB memory, 160 GB storage. We employed 20 instances and each of them is configured with a maximum of 2 map tasks and 1 reduce task (we don't have reduce tasks for PML) for task trackers.

For the accuracy experiments, we used 11 real-world data sets with different disk sizes. For the speed up performance, we applied 2 synthetic datasets and 4 real-world datasets. The details of the data sets can be found in Table 4.5. These data sets can be downloaded from the UCI repository [105] and other resources[17,18]. The first 8 datasets

[16]http://aws.amazon.com/ec2/

[17]http://nsl.cs.unb.ca/NSL-KDD/

are the same as in [113]. Datasets S1 and S2 are synthetic datasets which we generated applying the RDG1 data generator in WEKA [126] data mining tool with default parameter settings.

Table 4.5. Datasets used in our experiments

Data	No. of Instances	No. of Attributes	Size on Disk
yeast	892	8	34 KB
wineRed	1599	12	84 KB
wineWhite	4898	12	263 KB
pendigits	7494	16	360 KB
spambase	4601	57	687 KB
musk	6598	167	4.2 MB
telescope	19020	11	1.4 MB
kdd	148517	42	21.2 MB
isolet	7797	618	30.7 MB
org	2059	9731	38.4 MB
census	299285	42	129 MB
S1	100000	400	210 MB
S2	200000	400	420 MB

4.4.3 Performance Results

We make stratified sampling on each of the data into training, validation and test partitions assuring the number of instances for each of them are 36/50, 9/50 and 5/50 of the original data and the classes are uniformly distributed in each data set. The accuracy results we obtained are 10-fold cross validation results. In the experiments, the number of mappers in the training process is determined by the number of splits of the training data. To evenly distribute the classes, the training data is split equally among the mappers using the stratification technique. The base learning algorithm we used is Adaboost with

[18]http://www.findthatzipfile.com/search-35850240-hZIP/winrar-winzip-download-icdm08feature-code.zip.htm

decision stumps (decision trees with only one non-leaf node) as weak learner. This configuration is the same as the Adaboost.PL algorithm. The meta-learner we applied is Linear Regression.

A. Accuracy performance

The error rates of PML when the number of computing nodes changes from 1 to 20 are shown in Table 4.6. It can be seen from this table that compared to one mapper case our PML algorithm has lower or equivalent error rates in 9 out of 11 datasets. Although the error rates are increased in datasets "yeast" and "musk", the behavior of yeast is mainly due to the fact that it has the smallest data size. When it is split among different mappers, the more mappers the smaller partition each mapper has, which leads to inaccurate base models. This eventually produces the final ensemble model end up with higher error rates.

Table 4.6. Error rates for different number of nodes

(Lower error rates are marked in bold)

Data	**Number of Computing Nodes**				
	1	**5**	**10**	**15**	**20**
yeast	0.3071	0.3150	0.3520	0.3565	0.3497
wineRed	0.2351	**0.2270**	**0.2313**	**0.2332**	0.2457
wineWhite	0.2276	**0.2274**	**0.2213**	**0.2274**	**0.2274**
pendigits	0.0737	**0.0693**	**0.0644**	**0.0648**	**0.0625**
spambase	0.0680	**0.0575**	**0.0554**	**0.0575**	**0.0547**
musk	0.0180	0.0270	0.0265	0.0359	0.0380
telescope	0.1608	**0.1565**	**0.1534**	**0.1507**	**0.1487**
kdd	0.0342	0.0360	**0.0300**	**0.0279**	**0.0290**
isolet	0.1530	**0.1367**	**0.1394**	**0.1450**	0.1535
org	0.1578	**0.1457**	**0.1432**	0.1636	0.1690
census	0.0496	**0.0492**	**0.0491**	**0.0490**	**0.0482**

B. Comparison with Adaboost.PL

We also compared the error rates between our PML and the parallelized Adaboost algorithm AdaBoost.PL [113]. The comparison is based on the same 8 datasets in [113] as we could not find the datasets "swsequence" and "biogrid" which exist in [113]. The comparison results are shown in table 4.7. It can be seen that our PML algorithm has the lowest error rates in 7 out 8 datasets. The reduction of error rates compared to AdaBoost.PL is due to the fact that meta-learning is an ensemble scheme which improves the performance of base learners (here the base learner is Adaboost). This has been demonstrated in chapter 3 in one single machine. The reason why we got higher error rates on the "yeast" dataset is because the dataset's size is too small to build accurate base models with large split numbers (number of mappers), which is the same reason as we have explained in the accuracy performance experiments.

Table 4.7. Comparison of error rates with Adaboost.PL

(lowest error rates are marked in bold)

Data	**10 nodes**		**20 nodes**	
	AdaBoost.PL	**PML**	**AdaBoost.PL**	**PML**
yeast	**0.3464**	0.3520	**0.3375**	0.3497
wineRed	0.2464	**0.2313**	0.2564	**0.2457**
wineWhite	0.2313	**0.2213**	0.2309	**0.2274**
pendigits	0.0699	**0.0644**	0.0642	**0.0625**
spambase	0.0576	**0.0554**	0.0617	**0.0547**
musk	0.0565	**0.0265**	0.0612	**0.0380**
telescope	0.1599	**0.1534**	0.1595	**0.1487**
isolet	0.1601	**0.1394**	0.1661	**0.1535**

C. Speedup

To demonstrate the effectiveness of MapReduce framework of PML, we tested the speedup [124] performance of the datasets: "kdd", "isolet", "org", "census", "S1" and

"S2". We calculate speedup as the ratio of the training time for a single computing node over that of a number of computing nodes processing in parallel (we vary this number from 5, 10, 15 to 20). As mentioned in [113], the computational complexity f_1 for the AdaBoost algorithm can be expressed as

$$f_1 = \Theta(dn(T + log\ n)) \tag{4.2}$$

here n is the number of instances to be processed and T is the number of iterations. Therefore, for the training process of the PML algorithm, its computational complexity can be derived as

$$f_2 = \Theta\left(\frac{dn}{M}\left(T + log\ \frac{n}{M}\right)\right) \tag{4.3}$$

here M represents for the number of computing nodes processing in parallel. Using $f_1$and f_2, the speed up is given as

$$speedup = \frac{f_1}{f_2} = \Theta\left(\mathrm{M}\frac{\log n + T}{log\frac{n}{M} + T}\right) \tag{4.4}$$

Thus, theoretically it can be inferred that the value of speedup is expected to be larger than M.

The detailed results of speedup for different datasets are shown in table 4.8. To illustrate the speedup performance of different datasets, we plot their speedup results in fig.4.1. As can be seen from this figure, the larger the dataset size, the higher speedup the program achieves. The reason is that the communication cost between different computing nodes dominates small datasets, while the computation cost dominates large datasets.

Table 4.8. Speedup results for different computing nodes

Data	Number of Computing Nodes			
	5	**10**	**15**	**20**
kdd	5.5277	9.6036	10.8576	11.2606
isolet	2.6697	3.4059	3.5796	3.9684
org	4.0975	6.3585	7.9139	8.7913
census	4.9508	8.4430	11.8850	13.2451
S1	8.3170	14.6584	16.0796	18.1314
S2	4.4755	12.5927	18.0082	18.9423

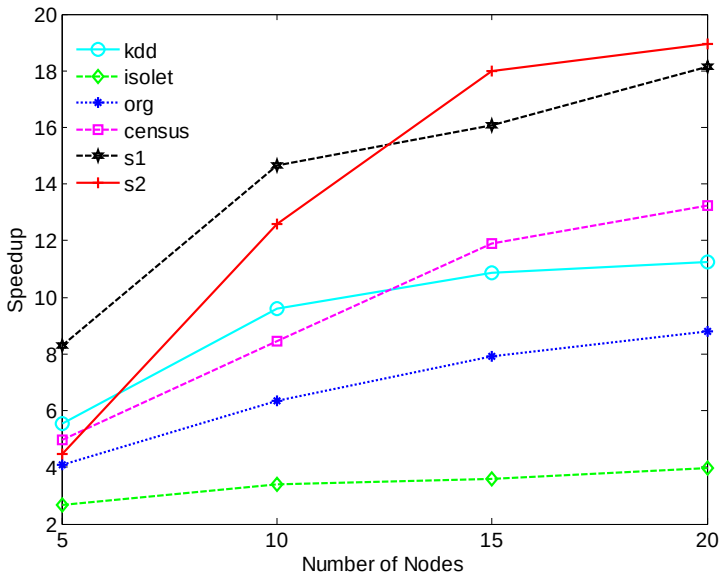

Fig.4.1. Speedup results for 6 data sets.

4.5 Conclusions

We proposed a parallel meta-learning algorithm PML implemented with MapReduce. This algorithm tackles the difficulties for supporting iterations in Hadoop. The experimental results showed that the error rates of PML are smaller than the results of a single node on 9 out of 11 datasets. The comparison between PML and the parallelized Adaboost algorithm AdaBoost.PL shows that PML has lower error rates than AdaBoost.PL on 7 out of 8 datasets. The speedup performance of PML proves that MapReduce improves the computation complexity substantially on big datasets. In summary, our PML algorithm has the ability to reduce computational complexity significantly, while producing smaller error rates.

Chapter 5

Conclusions and Future Work

5.1 Thesis Outcomes

This thesis aims to provide a new ensemble learning algorithm based on Adaboost and Meta-learning to improve the classification performance which all machine learning algorithms strive to increase. Although the accuracy performance is enhanced compared to the base learners and other contemporary similar approaches, the computation complexity is inevitably increased according to the multiple layer design of this algorithm. In order to tackle this problem, we propose a parallelized version of this algorithm utilizing the MapReduce framework. It turned out that this framework can also deal with the intrinsic weakness of the original MapReduce framework and could be applied as a generalized framework for parallelization of machine learning algorithms which have iterations in them.

5.2 Conclusions

By introducing the Meta-boosting algorithm, we are able to combine the Adaboost algorithm with different weak learns into one algorithm. This is different from the

original Adaboost algorithm as it can only boost one weak learner. Although there is a similar approach to combine different weak learners: the Adaboost Dynamic algorithm [95], the combination is done within the Adaboost algorithm via calling different weak learners in each of the iterations. This approach may cause confusion on the decision boundaries the weak learner is going to create in the next iteration as the training data's weight has been influenced by the previous weak learner's results. In contrast, our Meta-boosting algorithm combines the results of the weak learners through a second level learner: the meta-learner. In this way, instead of working in a sequential fashion, which is the way Adaboost Dynamic works, each base learner of our meta-boosting algorithm works in parallel. The experimental results show that meta-boosting algorithm has superior accuracy performance than its base learners and Adaboost Dynamic. The statistical tests also prove the results are significantly different. Since the meta-boosting algorithm is based on both Adaboost and Meta-learning, it has both reduced bias and variance. Moreover, the meta-learning approach we adopted is different than the original meta-learning algorithm in that we have a separate validation dataset and the processes of splitting the data and generating the final base classifiers are also different. These differences make this approach readily applicable for parallelization.

When we were parallelizing the meta-boosting algorithm, we found that Hadoop has no prototype for supporting iterations in an algorithm and that implementing iterations manually not only causes huge complexity, but also brings substantial communication overheads as the information has to be written and read repeatedly through HDFS. This is a disadvantage compared to the caching system for in-memory computation. To eliminate

these problems, we proposed to let each Adaboost algorithm work independently on a partition of the original datasets across a number of computing nodes. Then, their results are combined through a meta-learner, utilizing the meta-learning framework. Unlike our approach, the Adaboost.PL algorithm proposed in [113] parallelizes the Adaboost algorithm directly. They accomplish this task by: (1) running the original Adaboost algorithm on each computing node, which generates the hypotheses for all the iterations; (2) sorting these hypotheses according to their weights with ascending order; (3) merging the hypotheses within corresponding iterations to produce the final hypotheses for the final classifier. The number of iterations for the final classifier is the same as that of the Adaboost algorithm on each computing node. The entire process is very expensive and the validity of adopting this approach is problematic. In contrary, by adopting our framework: the PML, we can add or delete any algorithms without having to worry about the modifications needed. That is, our algorithm has a onetime configuration framework and can be used unlimited times afterwards. The experiments have proved that PML has better accuracy performance than Adaboost.PL and in-memory machine learning algorithm. Hence, PML has the following advantages: (1) able to classify big datasets with improved accuracy performances; (2) capable of parallelizing machine learning algorithms which require iterations in their computation; (3) no need to modify these algorithms themselves, which save a lot of efforts.

5.3 Future Work

In the Meta-boosting algorithm, although there are no existing theories available to analyze the change of bias and variance in Adaboost and meta-learning algorithm, there have been some empirical studies. We plan to do some separate experimental studies to prove the theoretical conjecture (both reduced bias and variance) we have mentioned for why the meta-boosting algorithm performs well. Moreover, the research on ensemble diversity is a very active area. We are also interested to see if the ensemble diversity is increased when we construct the algorithm. These results should also be supported by experiments. There have been lots of applications for ensemble methods on imbalanced datasets and we also plan to apply our algorithm on imbalanced datasets. Finally, one uncertainty about the meta-learning schema is to determine which kinds of base learners and meta-learns are suitable for generating good performances. We also plan to do some research on this issue, and, hopefully, we can provide some perspective on this.

For the parallel meta-learning algorithm, since the base learners which process part of the original datasets work in one single machine sequentially, in the next step, we plan to parallelize this step and distribute the computation to more computing nodes for the sake of increase the computation efficiency. Lacking of iteration is just one of the limitations of MapReduce. We are interested in discovering other disadvantages and propose methodologies to solve them. In fact, there are currently more and more new systems and frameworks proposed (which we have discussed in chapter 2) either to make

MapReduce more efficient or design brand new approaches such as applying declarative languages to deal with the relatively low level programming interface MapReduce provides. In sum, although MapReduce is ubiquitous and powerful to use, it is a relatively new framework and its shortcomings need to be addressed before we can fully explore its efficiency.

Reference

[1] John F. Gantz et al., “The Diverse and Exploding Digital Universe,” March 2008 [http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf]

[2] Dagum, Leonardo, and Ramesh Menon.“OpenMP: an industry standard API for shared-memory programming.” Computational Science & Engineering, IEEE 5, no. 1 (1998): 46-55.

[3] Gropp, William, Ewing Lusk, Nathan Doss, and Anthony Skjellum.“A high-performance, portable implementation of the MPI message passing interface standard.” Parallel computing 22, no. 6 (1996): 789-828.

[4] Catanzaro, Bryan, Narayanan Sundaram, and Kurt Keutzer.“Fast support vector machine training and classification on graphics processors.”In Proceedings of the 25th international conference on Machine learning, pp. 104-111.ACM, 2008.

[5] Graf, Hans P., Eric Cosatto, Leon Bottou, Igor Dourdanovic, and Vladimir Vapnik. “Parallel support vector machines: The cascade svm.”In Advances in neural information processing systems, pp. 521-528. 2004.

[6] Chang, Edward Y., Hongjie Bai, and Kaihua Zhu.“Parallel algorithms for mining large-scale rich-media data.” In Proceedings of the 17th ACM international conference on Multimedia, pp. 917-918.ACM, 2009.

[7] Robila, Stefan A., and Lukasz G. Maciak. “A parallel unmixing algorithm for hyperspectral images.” In Optics East 2006, pp. 63840F-63840F.International Society for Optics and Photonics, 2006.

[8] Dean, Jeffrey, and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters.” Communications of the ACM 51, no. 1 (2008): 107-113.

[9] Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung.“The Google file system.” In ACM SIGOPS Operating Systems Review, vol. 37, no. 5, pp. 29-43. ACM, 2003.

[10] Ranger, Colby, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. “Evaluating mapreduce for multi-core and multiprocessor systems.” In High Performance Computer Architecture, 2007.HPCA 2007. IEEE 13th International Symposium on, pp. 13-24. IEEE, 2007.

[11] Chaiken, Ronnie, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou.“SCOPE: easy and efficient parallel processing of massive data sets.” Proceedings of the VLDB Endowment 1, no. 2 (2008): 1265-1276.

[12] Apache, “Apache hadoop,” [http://hadoop.apache.org/].

[13] Chu, Cheng, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. “Map-reduce for machine learning on multicore.”Advances in neural information processing systems 19 (2007): 281.

[14] Liu, Chao, Hung-chih Yang, Jinliang Fan, Li-Wei He, and Yi-Min Wang.“Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce.” In Proceedings of the 19th international conference on World wide web, pp. 681-690.

ACM, 2010.

[15] Das, Abhinandan S., Mayur Datar, Ashutosh Garg, and Shyam Rajaram.“Google news personalization: scalable online collaborative filtering.”In Proceedings of the 16th international conference on World Wide Web, pp. 271-280.ACM, 2007.

[16] Panda, Biswanath, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo.“Planet: massively parallel learning of tree ensembles with mapreduce.” Proceedings of the VLDB Endowment 2, no. 2 (2009): 1426-1437.

[17] Apache, “Apache mahout,” [http://mahout.apache.org/].

[18] Hsieh, William W. Machine learning methods in the environmental sciences: Neural networks and kernels. Cambridge university press, 2009.

[19] Polikar, Robi, L. Upda, S. S. Upda, and Vasant Honavar.“Learn++: An incremental learning algorithm for supervised neural networks.” Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on 31, no. 4 (2001): 497-508.

[20] Mohammed, Hussein Syed, James Leander, Matthew Marbach, and Robi Polikar. “Can AdaBoost.M1 Learn Incrementally? A comparison to learn++ under different combination rules.”In Artificial Neural Networks–ICANN 2006, pp. 254-263.Springer Berlin Heidelberg, 2006.

[21] M. D. Muhlbaier, A. Topalis, and R. Polikar, “Learn++.NC: combining ensemble of classifiers with dynamically weighted consult-and-vote for efficient incremental learning of new classes,” IEEE Transactions on Neural Networks, vol. 20, no. 1, pp. 152–168, 2009.

[22] R. Polikar, J. DePasquale, H. Syed Mohammed, G. Brown, and L. I. Kuncheva, “Learn++.MF: A random subspace approach for the missing feature problem,” Pattern Recognition, vol. 43, no. 11, pp. 3817–3832, 2010.

[23] Muhlbaier, Michael, Apostolos Topalis, and Robi Polikar.“Ensemble confidence estimates posterior probability.”In Multiple Classifier Systems, pp. 326-335.Springer Berlin Heidelberg, 2005.

[24] T.G. Dietterich and G. Bakiri, “Solving Multiclass Learning Problems via Error-Correcting Output Codes,” Journal of Artificial Intel. Research, vol. 2, pp. 263–286, 1995.

[25] Rajan, Suju, and Joydeep Ghosh.“An empirical comparison of hierarchical vs. two-level approaches to multiclass problems.”In Multiple Classifier Systems, pp. 283-292.Springer Berlin Heidelberg, 2004.

[26] Smith, Raymond S., and Terry Windeatt.“Decoding rules for error correcting output code ensembles.”In Multiple Classifier Systems, pp. 53-63.Springer Berlin Heidelberg, 2005.

[27] Parikh, Devi, and Robi Polikar.“An ensemble-based incremental learning approach to data fusion.” Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on 37, no. 2 (2007): 437-450.

[28] Leonard, David, David Lillis, Lusheng Zhang, Fergus Toolan, Rem W. Collier, and John Dunnion.“Applying machine learning diversity metrics to data fusion in information retrieval.”In Advances in Information Retrieval, pp. 695-698.Springer Berlin Heidelberg,

2011.

[29] Wang, Shuo, Huanhuan Chen, and Xin Yao.“Negative correlation learning for classification ensembles.” In Neural Networks (IJCNN), The 2010 International Joint Conference on, pp. 1-8. IEEE, 2010.

[30] Wang, Shuo, and Xin Yao.“Diversity analysis on imbalanced data sets by using ensemble models.”In Computational Intelligence and Data Mining, 2009.CIDM'09. IEEE Symposium on, pp. 324-331. IEEE, 2009.

[31] Wang, Shuo, Ke Tang, and Xin Yao.“Diversity exploration and negative correlation learning on imbalanced data sets.”In Neural Networks, 2009.IJCNN 2009. International Joint Conference on, pp. 3259-3266. IEEE, 2009.

[32] Kuncheva, Ludmila I. “Classifier ensembles for changing environments.”In Multiple classifier systems, pp. 1-15.Springer Berlin Heidelberg, 2004.

[33] Chen, Sheng, and Haibo He.“Towards incremental learning of non-stationary imbalanced data stream: a multiple selectively recursive approach.” Evolving Systems 2, no. 1 (2011): 35-50.

[34] Dietterich, Thomas G. “Ensemble methods in machine learning.”In Multiple classifier systems, pp. 1-15.Springer Berlin Heidelberg, 2000.

[35] G. Brown, “Diversity in neural network ensembles.”PhD thesis, University of Birmingham, UK, 2004.

[36] Brown, Gavin, Jeremy Wyatt, Rachel Harris, and Xin Yao.“Diversity creation

methods: a survey and categorisation.” Information Fusion 6, no. 1 (2005): 5-20.

[37] Chandra, Arjun, and Xin Yao.“Evolving hybrid ensembles of learning machines for better generalisation.” Neuro-computing 69, no. 7 (2006): 686-700.

[38] Liu, Yong, and Xin Yao.“Ensemble learning via negative correlation.” Neural Networks 12, no. 10 (1999): 1399-1404.

[39] Kuncheva, Ludmila I. “That elusive diversity in classifier ensembles.” In Pattern Recognition and Image Analysis, pp. 1126-1138. Springer Berlin Heidelberg, 2003.

[40] Tumer, Kagan, and Joydeep Ghosh.“Error correlation and error reduction in ensemble classifiers.” Connection science 8, no. 3-4 (1996): 385-404.

[41] Bauer, Eric, and Ron Kohavi. “An empirical comparison of voting classification algorithms: Bagging, boosting, and variants.” Machine learning 36, no. 1-2 (1999): 105-139.

[42] Sharkey, Amanda JC.“Combining predictors.” In Combining artificial neural nets, pp. 31-50. Springer London, 1999.

[43] Kohavi, Ron, and David H. Wolpert.“Bias plus variance decomposition for zero-one loss functions.”In ICML, pp. 275-283. 1996.

[44] L. I. Kuncheva. Combining Pattern Classifiers: Methods and Algorithms. Wiley Press, 2004.

[45] L. Breiman, “Bagging predictors,” Machine Learning, vol. 24, no. 2, pp. 123–140,

1996.

[46] L. Breiman, “Random Forests,” Machine Learning, vol. 45, no. 1, pp. 5–32, Oct. 2001.

[47] T. K. Ho, “Random subspace method for constructing decision forests,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 20, no. 8, pp. 832–844, 1998.

[48] L.K. Hansen and P. Salamon, “Neural network ensembles,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 12, no. 10, pp. 993–1001, 1990.

[49] Chan, Philip K., and Salvatore J. Stolfo.“Experiments on multi-strategy learning by meta-learning.”In Proceedings of the second international conference on Information and knowledge management, pp. 314-323.ACM, 1993.

[50] Y. Freund and R.E. Schapire, “Decision-theoretic generalization of on-line learning and an application to boosting,” Journal of Computer and System Sciences, vol. 55, no. 1, pp. 119–139, 1997.

[51] Melville, Prem, and Raymond J. Mooney.“Creating diversity in ensembles using artificial data.” Information Fusion 6, no. 1 (2005): 99-111.

[52] Y.S. Huang and C.Y. Suen, “Behavior-knowledge space method for combination of multiple classifiers,” Proc. of IEEE Computer Vision and Pattern Recog., pp. 347–352, 1993.

[53] J. C. de Borda, Memoire sur les elections au scrutin. Paris: Historie de l'Academie

Royale des Sciences, 1781.

[54] L. I. Kuncheva, “Combining pattern classifiers, methods and algorithms,” New York, NY: Wiley Inter science, 2005.

[55] N. Littlestone and M. Warmuth, “Weighted majority algorithm,” Information and Computation, vol. 108, pp. 212–261, 1994.

[56] R. O. Duda, P. E. Hart, and D. Stork, “Algorithm independent techniques,” in Pattern classification,2 edn New York: Wiley, 2001, pp. 453–516.

[57] L. I. Kuncheva, J. C. Bezdek, and R. P. W. Duin, “Decision templates for multiple classifier fusion: an experimental comparison,” Pattern Recognition, vol. 34, no. 2, pp. 299–314, 2001.

[58] A.P. Dempster, “Upper and lower probabilities induced by multi-valued mappings,” Annals of Mathematical Statistics, vol. 38, no. 2, pp. 325–339, 1967.

[59] G. Shafer, A Mathematical Theory of Evidence. Princeton, NJ: Princeton Univ. Press, 1976.

[60] Kuncheva, Ludmila I. “ “Fuzzy” Versus “Non-fuzzy” in Combining Classifiers Designed by Boosting.” IEEE Transactions on fuzzy systems 11, no. 6 (2003): 729-741.

[61] K. M. Ting and I. H. Witten.“Issues in stacked generalization.”Journal of Artificial Intelligence Research, 10:271–289, 1999.

[62] Sharkey, Amanda JC.“Treating harmful collinearity in neural network

ensembles.”In Combining artificial neural nets, pp. 101-125. Springer London, 1999.

[63] Jordan, Michael I., and Robert A. Jacobs.“Hierarchical mixtures of experts and the EM algorithm.” Neural computation 6, no. 2 (1994): 181-214.

[64] Freund, Yoav, and Robert E. Schapire.“A decision-theoretic generalization of on-line learning and an application to boosting.”In Computational learning theory, pp. 23-37.Springer Berlin Heidelberg, 1995.

[65] Schapire, Robert E., Yoav Freund, Peter Bartlett, and Wee Sun Lee.“Boosting the margin: A new explanation for the effectiveness of voting methods.” The annals of statistics 26, no. 5 (1998): 1651-1686.

[66] Tom, M, “The need for biases in learning generalizaions.” Technical Report (1980).

[67] Pop, Mihai, and Steven L. Salzberg. “Bioinformatics challenges of new sequencing technology.” Trends in Genetics 24, no. 3 (2008): 142-149.

[68] Allison, David B., Grier P. Page, T. Mark Beasley, and Jode W. Edwards, eds.DNA microarrays and related genomics techniques: design, analysis, and interpretation of experiments. CRC Press, 2005.

[69] Japkowicz, Nathalie, and Shaju Stephen. “The class imbalance problem: A systematic study.” Intelligent data analysis 6, no. 5 (2002): 429-449.

[70] Banko, Michele, and Eric Brill.“Scaling to very very large corpora for natural language disambiguation.”In Proceedings of the 39th Annual Meeting on Association for Computational Linguistics, pp. 26-33.Association for Computational Linguistics, 2001.

[71] Halevy, Alon, Peter Norvig, and Fernando Pereira.“The unreasonable effectiveness of data.” Intelligent Systems, IEEE 24, no. 2 (2009): 8-12.

[72] Anand Rajaraman and Jerrey Ullman. Mining of Massive Datasets. [http://i.stanford.edu/ ullman/mmds.html].

[73] L. Tierney, A. J. Rossini, N. Li, and H. Sevcikova, “Snow: Simple network of workstations,” [http://cran.r-project.org/web/packages/snow].

[74] Curtin, Ryan R., James R. Cline, Neil P. Slagle, Matthew L. Amidon, and Alexander G. Gray.“MLPACK: A scalable C++ machine learning library.” Journal of Machine Learning Research 14 (2013): 801-805.

[75] Sujeeth, Arvind, Hyouk Joong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. “OptiML: an implicitly parallel domain-specific language for machine learning.” In Proceedings of the 28th International Conference on Machine Learning (ICML-11), pp. 609-616. 2011.

[76] Catanzaro, Bryan, Shoaib Kamil, Yunsup Lee, Krste Asanovic, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. “SEJITS: Getting productivity and performance with selective embedded JIT specialization.” Programming Models for Emerging Architectures (2009).

[77] Das, Sudipto, Yannis Sismanis, Kevin S. Beyer, Rainer Gemulla, Peter J. Haas, and John McPherson.“Ricardo: integrating R and Hadoop.” In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 987-998.ACM, 2010.

[78] S. Guha, “Rhipe: R and hadoop integrated processing environment,”

[http://ml.stat.purdue.edu/rhipe].

[79] Venkataraman, Shivaram, Erik Bodzsar, Indrajit Roy, Alvin Au Young, and Robert S. Schreiber.“Presto: distributed machine learning and graph processing with sparse matrices.”In Proceedings of the 8th ACM European Conference on Computer Systems, pp. 197-210.ACM, 2013.

[80] Hellerstein, Joseph M., ChristoperRé, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, KeeSiong Ng et al. “The MADlib analytics library: or MAD skills, the SQL.” Proceedings of the VLDB Endowment 5, no. 12 (2012): 1700-1711.

[81] Akdere, Mert, Ugur Cetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. “The Case for Predictive Database Systems: Opportunities and Challenges.” In CIDR, pp. 167-174. 2011.

[82] Borkar, Vinayak, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. “Hyracks: A flexible and extensible foundation for data-intensive computing.” In Data Engineering (ICDE), 2011 IEEE 27th International Conference on, pp. 1151-1162. IEEE, 2011.

[83] Yu, Yuan, Michael Isard, Dennis Fetterly, Mihai Budiu, ÚlfarErlingsson, Pradeep Kumar Gunda, and Jon Currey. “DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language.” In OSDI, vol. 8, pp. 1-14. 2008.

[84] Isard, Michael, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly.“Dryad:

distributed data-parallel programs from sequential building blocks.” ACM SIGOPS Operating Systems Review 41, no. 3 (2007): 59-72.

[85] Kang, U., Charalampos E. Tsourakakis, and Christos Faloutsos. “Pegasus: A peta-scale graph mining system implementation and observations.”In Data Mining, 2009.ICDM'09. Ninth IEEE International Conference on, pp. 229-238. IEEE, 2009.

[86] Gonzalez, Joseph E., Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. “PowerGraph: Distributed graph-parallel computation on natural graphs.”In Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 17-30. 2012.

[87] Zaharia, Matei, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. “Spark: cluster computing with working sets.” In Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, pp. 10-10. 2010.

[88] Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing.” In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, pp. 2-2.USENIX Association, 2012.

[89] Ghoting, Amol, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. “SystemML: Declarative machine learning on MapReduce.” In Data Engineering (ICDE), 2011 IEEE 27th International Conference on, pp. 231-242. IEEE, 2011.

[90] Kraska, Tim, Ameet Talwalkar, John C. Duchi, Rean Griffith, Michael J. Franklin, and Michael I. Jordan. “MLbase: A Distributed Machine-learning System.”In CIDR. 2013.

[91] Wolpert, David H., and William G. Macready.“No free lunch theorems for optimization.” Evolutionary Computation, IEEE Transactions on 1, no. 1 (1997): 67-82.

[92] Valentini, Giorgio, and Francesco Masulli. “Ensembles of learning machines.”In Neural Nets, pp. 3-20.Springer Berlin Heidelberg, 2002.

[93] Gama, Joao. “Combining classification algorithms.” PhD Thesis, University of Porto (1999)

[94] Schapire, Robert E. “The strength of weak learnability.” Machine learning 5, no. 2 (1990): 197-227.

[95] de Souza, Érico N., and Stan Matwin.“Extending adaBoost to iteratively vary its base classifiers.” In Advances in Artificial Intelligence, pp. 384-389.Springer Berlin Heidelberg, 2011.

[96] Sammut, Claude, and Geoffrey I. Webb, eds. Encyclopedia of machine learning. Springer-Verlag New York Incorporated, 2011.

[97] Vilalta, Ricardo, and Youssef Drissi. “A perspective view and survey of meta-learning.” Artificial Intelligence Review 18, no. 2 (2002): 77-95.

[98] Gama, Jo ão, and Pavel Brazdil.“Cascade generalization.” Machine Learning41, no. 3 (2000): 315-343.

[99] Dietterich, T. G. “Bias-variance analysis of ensemble learning.” In: 7th Course of the International School on Neural Networks, Ensemble Methods for Learning Machines (2002)

[100] Holte, Robert C. “Very simple classification rules perform well on most commonly used datasets.” Machine learning 11, no. 1 (1993): 63-90.

[101] Domingos, Pedro, and Michael Pazzani. “On the optimality of the simple Bayesian classifier under zero-one loss.” Machine learning 29, no. 2-3 (1997): 103-130.

[102] Domingos, Pedro.“A unified bias-variance decomposition.”In Proceedings of 17th International Conference on Machine Learning. Stanford CA Morgan Kaufmann, pp. 231-238. 2000.

[103] Webb, Geoffrey I., and Zijian Zheng. “Multistrategy ensemble learning: Reducing error by combining ensemble learning techniques.” Knowledge and Data Engineering, IEEE Transactions on 16, no. 8 (2004): 980-991.

[104] Chan, Philip, and Salvatore Stolfo. “Scaling learning by meta-learning over disjoint and partially replicated data.” In Proc. Ninth Florida AI Research Symposium, pp. 151-155. 1996.

[105] Frank, A., Asuncion, A. UCI machine learning repository , 2010.

[106] Kotsiantis, Sotiris B., Ioannis D. Zaharakis, and Panayiotis E. Pintelas. “Machine learning: a review of classification and combining techniques.” Artificial Intelligence Review 26, no. 3 (2006): 159-190.

[107] Japkowicz, Nathalie, and Mohak Shah. Evaluating learning algorithms: a classification perspective. Cambridge University Press, 2011.

[108] J. Bacardit and X. Llora, “Large scale data mining using genetics-based machine learning,” Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference (GECCO '09), NY, USA, pp. 3381-3412, 2009.

[109] T. White, Hadoop: The definitive guide. Yahoo Press, 2010.

[110] J. Venner, Pro Hadoop. Springer, 2009.

[111] C. Lam, J.Warren, Hadoop in action. Manning Publications Co., 2010.

[112] M. Kearns, “Efficient noise-tolerant learning from statistical queries,” Journal of the ACM (JACM), vol 45, pp. 392–401, 1999.

[113] P. Indranil and C. K. Reddy, “Scalable and parallel boosting with MapReduce,” Knowledge and Data Engineering, IEEE Transactions on vol 24, pp.1904-1916, 2012.

[114] J. Chao, C. Vecchiola, and R. Buyya, “MRPGA: An extension of MapReduce for parallelizing genetic algorithms,” eScience, pp.214-221, 2008.

[115] J. Ye, J. H. Chow, J. Chen, and Z. Zheng, “Stochastic gradient boosted distributed decision trees,” Proceedings of the 18th ACM conference on Information and knowledge management (CIKM '09), pp.2061-2064, NY, USA, 2009.

[116] M. Weimer, S. Rao, and M. Zinkevich, “A convenient framework for efficient

parallel multipass algorithms," NIPS 2010 Workshop on Learning on Cores, Clusters and Clouds, December 2010.

[117] Malewicz, Grzegorz, Matthew H. Austern, Aart JC Bik, James C. Dehnert, Ilan Horn, NatyLeiser, and Grzegorz Czajkowski. "Pregel: a system for large-scale graph processing." In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 135-146.ACM, 2010.

[118] Bu, Yingyi, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. "HaLoop: Efficient iterative data processing on large clusters." Proceedings of the VLDB Endowment 3, no. 1-2 (2010): 285-296.

[119] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. "The PageRank citation ranking: Bringing order to the web." Technical Report 1999-66, Stanford InfoLab, 1999.

[120] Bancilhon, Francois, and Raghu Ramakrishnan. "An amateur's introduction to recursive query processing strategies."Vol. 15, no. 2.ACM, 1986.

[121] Ekanayake, Jaliya, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-HeeBae, Judy Qiu, and Geoffrey Fox. "Twister: a runtime for iterative mapreduce." In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, pp. 810-818. ACM, 2010.

[122] Agarwal, Alekh, Olivier Chapelle, Miroslav Dud ĸ, and John Langford."A reliable effective tera-scale linear learning system." arXiv preprint arXiv:1110.4198(2011).

[123] Rosen, Joshua, Neoklis Polyzotis, Vinayak Borkar, Yingyi Bu, Michael J. Carey,

Markus Weimer, Tyson Condie, and Raghu Ramakrishnan. “Iterative MapReduce for Large Scale Machine Learning.” arXiv preprint arXiv:1303.3517 (2013).

[124] A. Grama, A. Gupta, G. Karypis, and V. Kumar, Introduction to Parallel Computing. Addison-Wesley, 2003.

[125] R.E. Schapire and Y. Singer, “Improved Boosting Algorithms Using Confidence-Rated Predictions,” Machine Learning, vol. 37, no. 3, pp. 297-336, 1999.

[126] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten, “The Weka Data Mining Software: An Update,” SIGKDD Explorations, vol. 11, no. 1, pp. 10-18, 2009.

Publications

1. Xiaoguang Wang, Xuan Liu, Nathalie Japkowicz, and Stan Matwin. "Ensemble of Multiple Kernel SVM Classifiers." Canadian AI, 2014.
2. Xiaoguang Wang, Xuan Liu, Nathalie Japkowicz, and Stan Matwin. "Automatic Target Recognition Using Multiple-Aspect Sonar Images." IEEE Congress on Evolutionary Computation, 2014.
3. Liu, Xuan, Xiaoguang Wang, Stan Matwin, and Nathalie Japkowicz. “Meta-learning for large scale machine learning with MapReduce.” In Big Data, 2013 IEEE International Conference on, pp. 105-110.IEEE, 2013.
4. Liu, Xuan, Xiaoguang Wang, Nathalie Japkowicz, and Stan Matwin. “An Ensemble Method Based on AdaBoost and Meta-Learning.” In Advances in Artificial Intelligence, pp. 278-285.Springer Berlin Heidelberg, 2013.
5. Xiaoguang Wang, Xuan Liu, Nathalie Japkowicz, and Stan Matwin. “Resampling and Cost-Sensitive Methods for Imbalanced Multi-instance Learning.” International Conference on Data Mining (ICDM), 2013.
6. Wang, Xiaoguang, Stan Matwin, Nathalie Japkowicz, and Xuan Liu.“Cost-Sensitive Boosting Algorithms for Imbalanced Multi-instance Datasets.”In Advances in Artificial Intelligence, pp. 174-186.Springer Berlin Heidelberg, 2013.

7. Wang, Xiaoguang, Hang Shao, Nathalie Japkowicz, Stan Matwin, Xuan Liu, Alex Bourque, and Bao Nguyen. “Using SVM with Adaptively Asymmetric Misclassification Costs for Mine-Like Objects Detection.” In Machine Learning and Applications (ICMLA), 2012 11th International Conference on, vol. 2, pp. 78-82. IEEE, 2012.
8. Liu, Xuan, and XiaoyiBao.“Brillouin spectrum in LEAF and simultaneous temperature and strain measurement.” Journal of Light wave Technology 30, no. 8 (2012): 1053-1059.
9. Bao, Xiaoyi, Shangran Xie, Xuan Liu, and Liang Chen.“The non-uniformity and dispersion in SBS-based fiber sensors.”In Asia Pacific Optical Sensors Conference, pp. 83512Y-83512Y.International Society for Optics and Photonics, 2012.
10. Liu, Xuan, and Xiaoyi Bao.“Simultaneous temperature and strain measurement with bandwidth and peak of the Brillouin spectrum in LEAF fiber.”In 21st International Conference on Optical Fibre Sensors (OFS21), pp. 775328-775328.International Society for Optics and Photonics, 2011.