ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
Available online at www.ijarbest.com
International Journal of Advanced Research in Biology Ecology Science and Technology (IJARBEST)
Vol. 2, Special Issue I, January 2016 in association with
KARUR VELALAR COLLEGE OF ARTS AND SCIENCE FOR WOMEN, KARUR
Conference on Emerging Trends and Functional Applications of Computer Technology - 12th January 2016

Knowledge Extraction from Massive Data through Meta-MapReduce Algorithm

Prof.P.GNANA SEKARAN M.Sc.,M.Phil.,B.Ed.,
Vice Principal, Jairams Arts & Science College, Karur - 3

Abstract

We have entered the big data age. Knowledge extraction from massive data is becoming more and more urgent. MapReduce provides a feasible framework for programming machine learning algorithms in Map and Reduce functions. The relatively simple programming interface has helped to solve machine learning algorithms' scalability problems. However, this framework suffers from an obvious weakness: it does not support iterations. This makes it difficult for algorithms requiring iterations to fully explore the efficiency of MapReduce. In this paper, we propose to apply Meta-learning programmed with MapReduce to avoid parallelizing machine learning algorithms while also improving their scalability to big datasets. The experiments conducted on Hadoop's fully distributed mode on Amazon EC2 demonstrate that our algorithm Meta-MapReduce (MMR) reduces the training computational complexity significantly when the number of computing nodes increases while obtaining smaller error rates than those on a single node. The comparison of MMR with the contemporary parallelized Ad a B oost algorithm,

Introduction

We have been rapidly moving from the Terabytes to the Petabytes age as a result of the explosion of data. The potential value and insights which could be derived from massive data sets have attracted tremendous interest in a wide range of business and scientific applications. 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 algorithm 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 the MapReduce infrastructure, researchers now have a simple programming interface for parallel scaling up of many data mining algorithms on larger data sets. It was shown that algorithms which fit the Statistical Query model can be written in a certain "summation form". They illustrated 10 different algorithms that can be easily parallelized on multi-core computers applying the MapReduce paradigm.

An industrial example of implementing MapReduce comes from Google. In 2009, Google proposed PLANET: a framework for large-scale tree learning using a MapReduce cluster . Their intention in building PLANET was to develop a scalable tree learner which could achieve comparable accuracy performance as the traditional in-memory algorithms and also be able to deal with bigger datasets. PLANET is used to construct scalable classification and regression trees, as well as ensembles of these models. It realizes parallelization by dividing tree learning into many distributed computations, each implemented with MapReduce.

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
Available online at www.ijarbest.com
International Journal of Advanced Research in Biology Ecology Science and Technology (IJARBEST)
Vol. 2, Special Issue I, January 2016 in association with
KARUR VELALAR COLLEGE OF ARTS AND SCIENCE FOR WOMEN, KARUR
Conference on Emerging Trends and Functional Applications of Computer Technology - 12th January 2016

Although MapReduce handles large scale computation, it doesn't support iteration. Since there are no loop steps available in Hadoop, in order to implement loops, an external driver is needed to

repeatedly submit MapReduce 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 Hadoop Distributed File System (HDFS) and the next MapReduce job which needs this information as inputs reads these messages from HDFS. Obviously, 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.

The inefficient creation of loops is a critical weakness of Hadoop. 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. However, even if the computations can be modeled by MapReduce when there are iterations in a machine learning algorithm, the execution overheads are substantial as was explained before.
Currently, there are some approaches which deal with the problem of lacking iterations in MapReduce. However, they either require substantial change of the original MapReduce framework or need designing new systems.
In contrast,we overcome this difficulty by applying the concept of Meta-learning. 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.
The method we propose in this paper is much simpler in that it eliminates the necessity of considering new models and the complexity of implementing them. Moreover, our approach requires a one time configuration and unlimited times of repetitive usage, which is much more advantageous than approaches which have to alter their design for different algorithms.

In this paper, we harness the power of meta-learning to avoid modifying individual machine learning algorithms with MapReduce. Hence, algorithms which require iterations in their model can be more easily parallelized utilizing the meta-learning schema than by altering their own internal algorithms directly with MapReduce. Although we focuse on an individual machine learning algorithm: Adaboost in this paper, the idea and our system can be easily extended to other algorithms.

This paper is organized as follows: section "Literature review" introduces work that has previously been proposed for solving the iterations problem in Hadoop MapReduce; section "Background" provides background knowledge about MapReduce and Hadoop; section "Framework" presents the meta-learning framework and our proposed algorithm: Meta-MapReduce (MMR); section "Results and discussion" demonstrates the performance of MMR in terms of error rates and speedup; section "Conclusion" concludes the paper.

Literature review

Hadoop the concept similar to MapReduce. The difference is that it provides a natural API for distributed programming framework aimed 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

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
Available online at www.ijarbest.com
International Journal of Advanced Research in Biology Ecology Science and Technology (IJARBEST)
Vol. 2, Special Issue I, January 2016 in association with
KARUR VELALAR COLLEGE OF ARTS AND SCIENCE FOR WOMEN, KARUR
Conference on Emerging Trends and Functional Applications of Computer Technology - 12th January 2016

also send information to other vertices that will be received at a next superstep. However, Pregel focuses on graph mining algorithms, while we are interested in more general applications.

As a modified version to iterative programs which is absent in the original MapReduce framework. It was mentioned that iterative computations were needed for Page Rank, recursive relational queries, clustering such as k-means, neural network analysis, social network analysis and so on. When doing these analyses, many iterations are necessary until some convergence or abort conditions are met. It was also mentioned that manually implementing iterative programs by multiple MapReduce jobs utilizing a driver program can be problematic. Therefore, proposes 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 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 a 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) is implemented with the All Reduce function (which originates in MPI) in 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) is a distributed memory abstraction intended for in-memory computations on large clusters. It is implemented in the system Spark. 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. The advantage of RDDs compared to 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 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 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 keep 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.

Background

Hadoop

Large scale data has brought both benefits and challenges to the field of machine learning. One of the benefits is that we can extract lots of useful

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
Available online at www.ijarbest.com
International Journal of Advanced Research in Biology Ecology Science and Technology (IJARBEST)
Vol. 2, Special Issue I, January 2016 in association with
KARUR VELALAR COLLEGE OF ARTS AND SCIENCE FOR WOMEN, KARUR
Conference on Emerging Trends and Functional Applications of Computer Technology - 12th January 2016

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 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.

One of the challenges of large scale computing is that storing and analyzing massive data is becoming more and more difficult. 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. 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, HDFS and MapReduce are its kernel components.

MapReduce

MapReduce simplifies many of the difficulties encountered in parallelizing data management operations across a cluster of individual machines, thus becoming 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.

MapReduce framework

As a programming model to process big data, there are two phases included in the MapReduce programs: the Map phase and the Reduce phase [30]. 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 $\langle key, value \rangle$ pairs, which are produced by the Map function. The key and value pairs are expressed as follows:

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

Here k_i represents the key for the ith input and v_i denotes the value associated with the ith input. K is the 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 for processing. The processed intermediate results are key-value pairs. Therefore, the map function can be obtained as

$$Map : K \times V \rightarrow (L \times W)^* \quad (2)$$

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
Available online at www.ijarbest.com
International Journal of Advanced Research in Biology Ecology Science and Technology (IJARBEST)
Vol. 2, Special Issue I, January 2016 in association with
KARUR VELALAR COLLEGE OF ARTS AND SCIENCE FOR WOMEN, KARUR
Conference on Emerging Trends and Functional Applications of Computer Technology - 12th January 2016

$$(k,v) \longmapsto [(l_1,x_1),\cdots,(l_1,x_r)]$$

(3)

Here L and W are key and value domains again, which represent 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: $[(l_1,x_1),\ldots,(l_1,x_r)]$ with the same key, but different values. These key-value pairs are the inputs for the reduce functions. The reduce phase is defined as

$$Reduce : L \times W^* \rightarrow W^*$$

(4)

$$(l,[y_1,\cdots,y_{sl}]) \longmapsto [w_1,\cdots,w_{ml}]$$

(5)

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 for processing. 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 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

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

(6)

As was 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.

MapReduce workflow

In order to do cloud computing, the original data is di-vided into the desired number of subsets (each subset has a fixed-size) for the MapReduce jobs to proceed. 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, if one of the machines which runs the map functions fails before it produces 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 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".

Figure 1 depicts the work flow of a generalized MapReduce job. To execute such a 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.

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
Available online at www.ijarbest.com
International Journal of Advanced Research in Biology Ecology Science and Technology (IJARBEST)
Vol. 2, Special Issue I, January 2016 in association with
KARUR VELALAR COLLEGE OF ARTS AND SCIENCE FOR WOMEN, KARUR
Conference on Emerging Trends and Functional Applications of Computer Technology - 12th January 2016

Fig. 1. Work flow of MapReduce framework

A simple word count example

In order to illustrate the work flow of MapReduce and the procedure of how the ⟨*k e y*,*v a l u e*⟩ 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. 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 ⟨*k e y*,*v a l u e*⟩ pairs. Then these pairs are split into three partitions and each partition is processed by a mapper to generate the intermediate ⟨*k e y*,*v a l u e*⟩ 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 ⟨*k e y*,*v a l u e*⟩ 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. A word count example with MapReduce

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. Preventing 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.

An HDFS cluster works following the master and slave model. The master is called the namenode while the slave is called the data node. The filesystem's namespace is managed by the namenode. The filesystem's 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. The 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 into the name nodes periodically. The reason is that the namenode doesn't store block locations persistently and the information is reconstructed from the datanodes. Figure 3 illustrates the case when the block replication is three and two files "/user/aaron/foo"&"/user/aaron/bar" are divided into pieces.

Fig.3. 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

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
Available online at www.ijarbest.com
International Journal of Advanced Research in Biology Ecology Science and Technology (IJARBEST)
Vol. 2, Special Issue I, January 2016 in association with
KARUR VELALAR COLLEGE OF ARTS AND SCIENCE FOR WOMEN, KARUR
Conference on Emerging Trends and Functional Applications of Computer Technology - 12th January 2016

interface (POSIX) is presented to the user by the client. The data flow for a file read is demonstrated in Fig. 4. 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 finishing reading the block in step 4, the datanode is closed and it continues to read the next block which also comes from the closest datanode in step 5. This continues for a number of cycles until the client has finished reading all the blocks it needs. In the final step: step 6, the FSDataInputStream is closed.

Fig. 4. 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 handle potential namenode failures, Hadoop provides two choices. The first one consists of allowing the namenode to write 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 case of namenode failure.

Research design and methodology

In this section, we first take the AdaBoost.M1 algorithm as an example to explain why it is inefficient for the original MapReduce configuration in Hadoop to do iterations for machine learning algorithms. Then we review the Meta-learning algorithm and propose our framework MMR which is implemented with the programming model of MapReduce on Hadoop. As will be introduced in the following section, the process of building and testing the base classifiers of meta-learning can be executed in parallel, which makes meta-learning easily adaptable to distributed computation.

Problems with MapReduce for Iterations

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. AdaBoost.M1 was designed to extend AdaBoost from handling the original two classes case to the multiple classes case. The detailed algorithm is shown in Algorithm 1.

In order to implement this algorithm using MapReduce, for T iterations T MapReduce jobs need to be submitted by the a driver program. This driver program also needs to determine for each iteration t whether this abortion condition $\varepsilon_t > 1/2$ is met. In this case, the number of MapReduce jobs is smaller than T.

There are many problems with this implementation. First, the training data sent to each MapReduce job is dependent on each other as each taining data subset S_t is drawn from the distribution D_t (Algorithm 1, line 4) and this distribution is updated based on the results of the previous MapReduce job. This means these

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
Available online at www.ijarbest.com
International Journal of Advanced Research in Biology Ecology Science and Technology (IJARBEST)
Vol. 2, Special Issue I, January 2016 in association with
KARUR VELALAR COLLEGE OF ARTS AND SCIENCE FOR WOMEN, KARUR
Conference on Emerging Trends and Functional Applications of Computer Technology - 12th January 2016

MapReduce jobs cannot be executed in parallel as they have to wait for the distribution D_t from the previous MapReduce job.

Second, every time a MapReduce job starts it needs to read data from the HDFS where the previous MapReduce job has stored the distribution D_t which will help select the training subset S_t. After this MapReduce job finishes, it again writes its results into the HDFS. For T iterations, the communication overhead is substaintial as data are re-loaded, re-saved and re-processed for Ttimes. Consequently, a lot of CPU resources, network bandwidth and I/O are wasted. For smaller datasets, it becomes a major factor which reduces the performances.

Third, as we mentioned before, a driver program is required for each MapReduce job to check the termination condition: $\varepsilon_t > 1/2$. This driver progam is an extra MapReduce job and causes overheads as extra tasks need to be scheduled, extra data need to read and save to HDFS, extra networks resources are demanded to move these data.

Meta-learning Algorithm

Every learning algorithm is subject to inductive bias. This means that every algorithm will search for a solution in a specific way which may or may not be appropriate for the problem at hand. In the No Free Lunch (NFL) theorems, 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 them in the first level.

Three meta-learning strategies: combiner, arbiter and hybrid are proposed in to combine the predictions generated by the first level base learners. Here we are focusing on the combiner strategy. For this strategy, depending on how we generate the meta-level training data, there are three schemes as follows:

Class-attribute-combiner:a meta-level training instance includes the training features, the predictions by each of the base classifiers and the true class for test instance;

Binary-class-combiner: for this rule, the meta-level instance consists of all the base classifiers' predictions for all the classes and the true class for the test instance;

Class-combiner: it is similar to the binary-class-combiner except that this rule only contains the predictions for the predicted class from all the base classifiers and the true class.

In the case of three base classifiers, we represent the predictions of these classifiers on a training instance X as $\{P_1(X), P_2(X), P_3(X)\}$, the attribute vector for X is $\langle x_1, x_2, \cdots, x_n \rangle$ if the number of attributes is n, the number of classes is m, the true class for X is expressed as $label(X)$. Then the meta-level training instance for X can be expressed as: (7) for the class-attribute-combiner, (8) for the binary-class-combiner and (9) for the class-combiner.

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

(7)

$$M_t = \{P_{11}(X), \cdots, P_{1m}(X), P_{21}(X), \cdots, P_{2m}(X), P_{31}(X), \cdots, P_{3m}(X), label(X)\}$$

(8)

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

(9)

In the methodology we will propose in section Meta-MapReduce (MMR), the third rule: the 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

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
Available online at www.ijarbest.com
International Journal of Advanced Research in Biology Ecology Science and Technology (IJARBEST)
Vol. 2, Special Issue I, January 2016 in association with
KARUR VELALAR COLLEGE OF ARTS AND SCIENCE FOR WOMEN, KARUR
Conference on Emerging Trends and Functional Applications of Computer Technology - 12th January 2016

based on the new test instance. The procedure for meta-learning applying the combiner rule is described as follows: for a dataset $S=\{(y_n, x_n), n=1,\cdots,N\}$ 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: $S_1,\cdots,S_j,\cdots,S_J$. 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 of the base learners to generate L models: $M_{j11},\cdots,M_{j1}l,\cdots,M_{j1}L$.

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 the 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 $M_{j1}L$ is denoted as Z_{Lg} for instance x_g, then the generated meta-level training data can be expressed as

$$S_{CV}=\left\{(Z_{1g},\cdots,Z_{Lg},y_g), g=1,\ldots,G\right\}$$

(10)

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 $\tilde{M}$. The last step in the training process is to train all the base learners on S^{-j} to generate the final level-0 models: $M_1,\cdots,M_l,\cdots,M_L$. The whole training process is also shown in Fig. 4.

In the test process, to generate the meta-level test data, S_j is used to test the level-0 base models $M_1,\cdots,M_l,\cdots,M_L$ generated in the training process. For instance, for the instance x_{test} in S_j, the meta-level test instance can be obtained as $(Z_1,\cdots,Z_l,\cdots,Z_L)$). All the meta-level test data are feed into the meta-level model $\tilde{M}$ 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 Algorithm 2.

Meta-MapReduce (MMR)

The in-memory meta-learning presented in section Meta-learning Algorithm and the distributed meta-learning which we are going to provide in this section differ as follows. 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 is 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 MMR algorithm includes three stages: Training, Validation and Test. For the base learning algorithms, we used the same machine learning algorithm. The number of base learning algorithms is equal to that of the mappers. We applied map functions without reduce functions as the map functions already generates the base classifiers we need in the training process and no further procedures are required.

It is also possible to use different base learning algorithms. But in order to compare our results with the parallel Adaboost algorithm: AdaBoost.PL in [10], we used the same base learning algorithm. MMR and AdaBoost.PL is very similar at the beginning of the training process in that they both split the original data into multiple partitions and train each part on different computing node with the AdaBoost.M1 algorithm with the same base learner. The difference is that: MMR uses the validation dataset to get the predictions from these base classifiers and then use these predictions to train a meta-learner algorithm; Adaboost.PL sorts the hypotheses generated from all the iterations from each computing node and then by merging them together a final classifier is generated.

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
Available online at www.ijarbest.com
International Journal of Advanced Research in Biology Ecology Science and Technology (IJARBEST)
Vol. 2, Special Issue I, January 2016 in association with
KARUR VELALAR COLLEGE OF ARTS AND SCIENCE FOR WOMEN, KARUR
Conference on Emerging Trends and Functional Applications of Computer Technology - 12th January 2016

MMR 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 Algorithm 3 and the detailed procedure is described as follows.

Let

$$D_{n^m}^m = \left\{\left(x_1^m, y_1^m\right), \left(x_2^m, y_2^m\right), \cdots, \left(x_{n^m}^m, y_{n^m}^m\right)\right\}$$

be the training data assigned to the mth mapper where $m \in \{1,\cdots,M\}$ 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 3). 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 4). Finally, all the trained base models are collected and stored in the output file (i.e., the HDFS) (line 6).

MMR 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 Algorithm 4 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 4). Instead of generating the predicted labels, the predicted probabilities PD^m for each of the classes from all the base classifiers are collected and put together (line 6). 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 $l(D_{kp}^p)$ (as labels) (line 8). The next step is that each map function outputs the $\langle key,value\rangle$ pairs $\langle l(D_{kp}^p), MI^p\rangle$. Here,the key is the true labels of the validation data: $l(D_{kp}^p)$ while value is the meta-instances MI^p (line 9). 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 13).

Train meta-learner: let the total number of classes be NC. For each class, new meta-instances: $filter^n c(MI)$ 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 $h^n c$ (line 16). 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 18).
MMR Test: in the test process, the meta-level test data are generated and the meta-classifiers $(h^1,\cdots,h^N C)$ are tested. The pseudo code is shown in Algorithm 5.

The following explains the processes:

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 2-14.

Test $(h^1,\cdots,h^N C)$: for each instance $MI^*[i]$ in the meta-instances MI^*, the probability $probn^c$ for each class is obtained by classifying the corresponding meta-classifier $h^n c$ on a new meta-instance $filter^n c(MI^*[i])$ (line 17). This instance is generated the same way as $filter(MI)$ in the validation process. The probabilities for all classes are then summed (line 18) and normalized (line 21). The predicted label l^i is found by seeking the class ID of the calculated normalized highest probabilities (line 23). Finally, the predicted labels for all the test instances are returned (line 25).

Results and discussion

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

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
Available online at www.ijarbest.com
International Journal of Advanced Research in Biology Ecology Science and Technology (IJARBEST)
Vol. 2, Special Issue I, January 2016 in association with
KARUR VELALAR COLLEGE OF ARTS AND SCIENCE FOR WOMEN, KARUR
Conference on Emerging Trends and Functional Applications of Computer Technology - 12th January 2016

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. 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 performance when we increase the number of computer nodes.

Experiment Settings

The experiments are deployed on Amazon EC2 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 1. These data sets can be downloaded from the UCI repository and other resources. The first 8 datasets are the same as in Datasets S1 and S2 are synthetic datasets which we generated applying the RDG1 data generator in WEKA data mining tool with default parameter settings.

Table 1. Datasets used in our experiments

Performance Results

We applied stratified sampling on each of the datasets in order to form training, validation and test partitions assuring that the number of instances for each of them, respectively, are 36/50, 9/50 and 5/50 of the original data and the classes are uniformly distributed. 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 decision stumps (decision trees with only one non-leaf node) as weak learners. This configuration is the same as that of the Adaboost.PL algorithm. The meta-learner we applied is Linear Regression.

Accuracy Performance: The error rates of PML when the number of computing nodes changes from 1 to 20 are shown in Table 2. It can be seen from this table that compared to the one mapper case our PML algorithm has lower or equivalent error rates in 9 out of 11 datasets. Although the error rates increased in datasets "yeast" and "musk", the behavior of yeast is mainly due to the fact that it has the smallest data size. When the data is split among different mappers, the more mappers there are, the smaller the partition each mapper has, which leads to inaccurate base models. This eventually produces higher error rates in the final ensemble model.

Table 2. Error rates for different number of nodes

Comparison with Adaboost.PL: We also compared the error rates obtained by our MMR and the parallelized Adaboost algorithm Ada-Boost.PL. The comparison is based on 8 of the datasets in as we could not find the datasets "swsequence" and "biogrid" which they also tested. The comparison results are shown in Table 3. 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). 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. As we don't have the source code for the AdaBoost.PL algorithm, it is difficult for us to do the statistical significance test on the performance results.

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
Available online at www.ijarbest.com
International Journal of Advanced Research in Biology Ecology Science and Technology (IJARBEST)
Vol. 2, Special Issue I, January 2016 in association with
KARUR VELALAR COLLEGE OF ARTS AND SCIENCE FOR WOMEN, KARUR
Conference on Emerging Trends and Functional Applications of Computer Technology - 12th January 2016

Table 3. Error rates comparison between AdaBoost.PL and MMR

Speedup: to demonstrate the effectiveness of the MapReduce framework of MMR, we tested the speedup 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). The detailed results of speedup for different datasets are shown in Table 4. To illustrate the speedup performance of different datasets, we plot their speedup results in Fig. 5. 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.

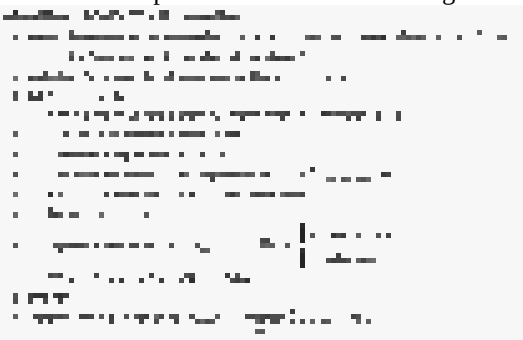

Fig. 5. Speedup results for 6 data sets

Table 4. Speedup results for different computing nodes

It should be noted that the values of speedup sometimes are higher than the number of nodes. For instance, for dataset S1, when the number of nodes is 10, the speedup value is 14.6584. This nonlinear performance can be explained in terms of computational complexity as follows.

The computation complexity of the AdaBoost algorithm f_1 depends on the number of iterations T, the number of instances N, the number of attributes D. Since we set the base learner of the AdaBoost algorithm as decision stump, f_1 can be expressed as

$$f_1 = \Theta\left(DN\left(logN + T\right)\right)$$

(11)

First sorting of the data attributes has a computationl complexity of $\Theta(D\ N\ log\ N)$, this is done before the iteration starts. After the iteration starts, in each iteration the time complexity of the decision stump is $\Theta(D\ \ N)$. As there are T iterations, the computational comlexity for the iterations part is $\Theta(D\ N\ T)$. Therefore, the total computational complexity is $\Theta(D\ N(log\ N+T))$.

For the training process of the MMR algorithm, assume the number of computing nodes is M, the computational complexity f_2 is

$$f_2 = \Theta\left(D\frac{N}{M}\left(log\frac{N}{M} + T\right)\right)$$

(12)

Based on f_1 and f_2 , speedup can be derived as

$$speedup = \frac{f_1}{f_2} = \Theta\left(M\frac{logN + T}{log\frac{N}{M} + T}\right)$$

(13)

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

Conclusion

We proposed a Meta-MapReduce algorithm MMR implemented with MapReduce. This algorithm tackles the difficulties for supporting iterations in Hadoop. The experimental results show that the error rates of MMR are smaller than the results of a single node on 9 out of 11 datasets. The comparison between MMR and the parallelized Adaboost algorithm AdaBoost.PL shows that PML has lower error rates than AdaBoost.PL on 7 out of 8 datasets.

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)
Available online at www.ijarbest.com
International Journal of Advanced Research in Biology Ecology Science and Technology (IJARBEST)
Vol. 2, Special Issue I, January 2016 in association with
KARUR VELALAR COLLEGE OF ARTS AND SCIENCE FOR WOMEN, KARUR
Conference on Emerging Trends and Functional Applications of Computer Technology - 12th January 2016

The speedup performance of MMR proves that MapReduce improves the computation complexity substantially on big datasets. In summary, our MMR algorithm has the ability to reduce computational complexity significantly, while producing smaller error rates.

For the MMR 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 increasing the computational efficiency. Moreover, we used the same algorithm: AdaBoost.M1 for all the computing nodes in this work. We intend to use different algorithms on different computing nodes to increase the accuracy further. The reason is that Meta-learning belongs to the ensemble learning paradigm of machine learning and the more diverse the base learners are, the higher accuracy could be expected.

References

1. Rajaraman A, Ullman JD (2011) Mining of Massive Datasets. Cambridge University Press, Cambridge.
2. White T (2012) Hadoop: The Definitive Guide. " O'Reilly Media, Inc.", California.
3. Venner J, Cyrus S (2009) Pro Hadoop. vol. 1. Springer, New York.
4. Lam C (2010) Hadoop in Action. Manning Publications Co., New York.
5. Panda B, Herbach JS, Basu S, Bayardo RJ (2013) Planet: massively parallel learning of tree ensembles with mapreduce. Proc. VLDB Endowment 2(2):1426-1437
6. Palit I, Reddy CK (2012) Scalable and parallel boosting with mapreduce. IEEE Trans Knowl Data Eng 24(10):1904-1916
7. Weimer M, Rao S, Zinkevich M (2010) A convenient framework for efficient parallel multipass algorithms In: LCCC: NIPS 2010 Workshop on Learning on Cores, Clusters and Clouds.
8. Agarwal A, Chapelle O, Dudík M, Langford J (2014) A reliable effective terascale linear learning system. J Mach Learn Res 15:1111-1133
9. Bancilhon F, Ramakrishnan R (1986) An Amateur's Introduction to Recursive Query Processing Strategies. vol. 15. ACM, New York, NY, USA.
    a.
10. Amazon Elastic Compute Cloud:Amazon EC2..
    a. http://aws.amazon.com/ec2/ *webcite*
    b.