Random Forest using Harp: Decision Tree + Ensemble

Rohan Ingale
Department of Computer Science
Indiana University, Bloomington
(812) 369 - 5661
ringale@indiana.edu

Anand Karandikar
Department of Computer Science
Indiana University, Bloomington
(812) 269 - 6299
askarand@indiana.edu

Abstract

This paper presents an approach to construct a Random forest[1] in a distributed environment using Harp to support parallel computation. Random forest is a classifier which constructs multiple decision trees from the training data set, and predicts the class labels of samples in test data set. In this paper we will describe in detail the technique used to construct such a classifier in distributed environment and compare it's performance based on several metrics.

Keywords

Random Forest, Decision Tree, Model, Classifier, Learning, Prediction, Features. Harp, HDFS, Random Trees, Sampling, Distributed Environment, Entropy

1. Introduction

There is always a trade-off concerned when selecting the learning model for the prediction task. The trade-off is between whether to construct a single classifier or an ensemble of classifiers. Random forest is an ensemble technique for constructing the classifier. It consists of multiple decision trees, with each decision tree constructed by sub sampling the feature set and the training samples[2]. The random forest construction in distributed environment will be discussed in detail in the later sections. We have used the airline data set provided by DataExpo[4] for implementing the distributed random forests algorithm. The task at hand was to predict whether the flight will be delayed or will arrive on time.

1.1 Motivation

The pseudo code for sequential algorithm of Random forest that builds an ensemble of random decision trees is as follows:

```
RandomTrees =[]
for i = 0 to numtrees:
// bootstrap technique to sample size N data
sample_data = sample(data, N)
// m dictates the number of features to consider (randomly) at every node in the random tree
RandomTrees[i] = build_random_tree(data, m)
```

It is evident from this pseudo-code that the ensemble of decision trees is generated sequentially in that the decision trees are generated one after the other. The important observation that can be made here is that the random trees generated are independent from each and can be generated in parallel to improve the performance.

2. Problem Definition

In this ensemble method, Random Trees are constructed on the training data. The training data is usually huge and complex with large set of features. Thus, random selection of samples with replacement and selecting the sub set of features requires a significant amount of time. Also, constructing decision trees on large amount of data requires significant amount of time. Constructing these decision trees sequentially would further degrade the performance.

We propose a method to implement the Random Forest algorithm in distributed environment by distributing the data on multiple nodes and constructing the decision trees on these nodes in parallel to reduce the total time required and thus, improve the performance. Further, we will also discuss the tradeoffs between sequential and parallel execution and challenges in distributed implementation.

3. Proposed Method

We have referred to an already existing sequential algorithm implementation of random forests[3] and ported it to implement the scalable version using harp. Harp is a map-only model. The map tasks are used to generate an ensemble of random trees. The map tasks are implemented using multiple threads.

Each map task creates as many threads as the number of trees to be constructed by the map task. Each thread samples the training data and features and then constructs the decision tree. The size of the data considered for training is same as the total size of the training data set however, may contain duplicate samples since we perform sampling with replacement.

The training data is read from local memory and split into several chunks since the data is too large to store on a single node. The distribution of the training data into chunks is done in a way such that the percentage of data belonging to a particular class remains uniform over all chunks. These chunks are then written into separate files on HDFS. The data nodes are distributed over multiple physical nodes. Each map task reads a file from HDFS and uses the data in that file to build the classifier. Each map task gets only a chunk of data. However, since the distribution of class labels over the chunks is uniform, these chunks can be assumed to be a representative portion of entire data.

The map task creates multiple threads for parallel construction of multiple decision trees. Each thread performs sampling with replacement over the training data so that each thread has partially unique data. Each thread also takes a random subset of features.

A feature is used to split the records on a decision tree node. The value of feature with least entropy[1] is used as the split value. In order to obtain this value, the attribute values are sorted and the value where the class label changes is used asa candidate for

possible split. If the number of values where the split is possible is greater than some threshold then candidate values after a fixed interval are considered.

Once the entire decision tree is constructed, the decision tree is tested over a fraction of train data which is not used for training. The accuracy measure over this data gives the OOB (Out of Box) error rate. This error rate is used to decide the weight of this tree in the random forest. Thus the trees with low OOB error are given higher weights and contribute more in predicting the labels of test samples.

The test data set is read from files on the local memory and stored in a test data set file on HDFS. Since the test data is very small in size compared to the training data, we assume that the test data can be stored on a single node. Each map task will read the test data from this file on HDFS.

Each decision tree in the Random Forest predicts a label for a test sample given as input. A majority vote is taken based on the label predicted by each tree and it's corresponding weight to determine the label of sample predicted by the random forest as a whole.

We have used Harp 3.0 for the implementation. The mapCollective API[5] is used by each Map task as the start point which reads all the training file names and calls the random forest generation API The test data set file path, total number of trees to be constructed by each map task and the number of attributes to be considered for the construction of each tree is set as the job configuration and used by each map task. The result of prediction task performed on the test data is stored in a Table data structure specific to Harp. A table is created by the master node and broadcasted to all the other nodes. The predicted value of a sample is stored in this table under a partition with partition id same as the row id of the sample. Thus the predictions done by different map tasks for same test sample are stored in the table with the same partition id. Along with the prediction the OOB error for that tree is also stored which is used to take the majority vote. Once class labels are assigned to all the test data samples the Reduce API[5] is used to combine results from all map task. The combiner function takes product of the class label with the accuracy of the tree and takes summation of all such values belonging to a single partition. Assuming that it is a binary classification we have mapped class 0 to value -1. Thus, if the summation is negative then the sample is assigned class label 0 else it is assigned class label 1. The accuracy of the random forest is computed based on the confusion matrix and is displayed as the output.

4. Experiments

We have built and tested our classifier on the Airline dataset[4]. The classifier is built using samples from January 2008 data set and is tested on samples from February 2008 data set. The ratio between train and test data set is 10:1.

The experiment was done for different number of attributes used, threads created and map tasks created to observe the change in accuracy and time required to execute. The following table gives the variation in accuracy for different number of attributes. Since the performance is best for attributes size of 5 and 6, we have used them to experiment for different number of threads and map tasks. The following graph shows the results for the change in accuracy with increase in the number of trees/threads.

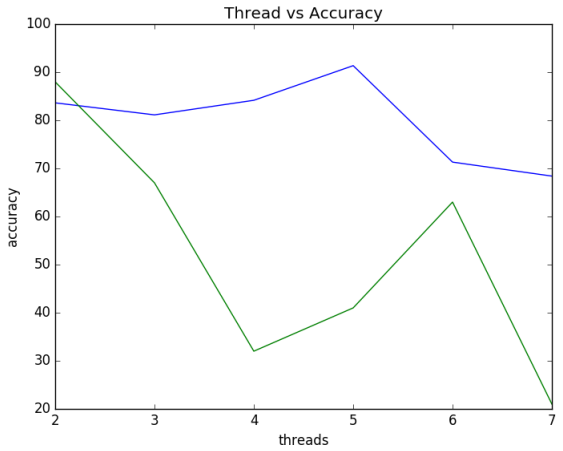

Figure 1. Accuracy vs Number of Threads.

The graph shows performance for attribute size of 5 and 6. For attribute size of 5, the accuracy increases with increase in the number of threads till total of 5 threads and then decreases with further increase in number of threads. The accuracy with 6 attributes is also better for 6 threads but degrades further. The accuracy increases with increase in the number of trees since the incorrect prediction made by a single decision tree is cancelled out in the ensemble.

The following graph shows the change in accuracy with the number of map tasks.

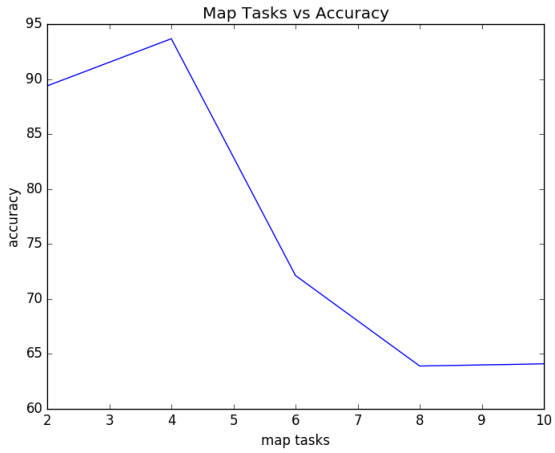

Figure 2. Accuracy vs Map Tasks.

The accuracy initially increase with an increase in the number of map tasks but decreases with further increase in the number of map tasks. The training data file is divided into number of chunks equal to number of map tasks. Thus, as the number of map tasks goes on increasing, the chunk size given to each map task decreases and thus the classifier gives low performance as it overfits to the small data given as an input

The following graph shows the change in total time required for execution with the number of map tasks.

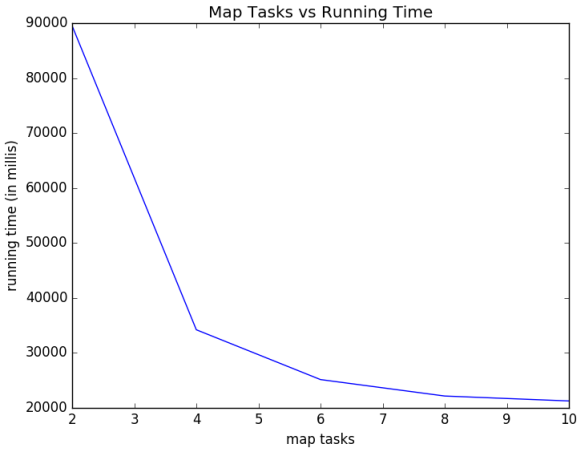

Figure 3. Running Time vs Map Tasks.

As can be seen from the graph, the time required for execution decreases with increase in the number of map tasks.

Following is the output obtained after running the Random Forest algorithm for configuration: 4 map tasks, 5 features to be selected during feature sampling and 2 trees generated by each map task:

Accuracy: 95.67

Total execution time: 34214 (ms)

Confusion matrix:

	Predicted		
		1	0
Actual	1	4168	649
	0	0	10178

Table 1. Confusion Matrix.

5. Conclusion

Parallel computation can help in faster computing of Random Forest and prediction of label, thus better utilizing the available resources.

It can be observed that the accuracy of predictions initially improves with increase in the number of trees but then decreases after a certain point.

The amount of time required for computation increases with increase in the number of trees since reduce function is the bottleneck.

The accuracy of predictions initially remains constant but later decreases with an increase in the number of map tasks since the training data is split into smaller chunks.

The amount of time required for computation decreases with an increase in number of map tasks.

Thus, there is a trade-off between the degree of parallelism and the accuracy of predictions which needs to be considered.

6. Acknowledgement

We extend our sincere thanks and deep gratitude to Prof. Judy Qiu for her valuable advice and guidance. We would like to thank Prof. Peng Bo for guiding us with the critical design and implementation decisions.

7. References

[1] Michael Steinbach, Pang-Ning Tan and Vipin Kumar "Introduction to Data Mining"

[2] Robin Genuer, Jean-Michel Poggi, Christine Tuleau-Malot, Nathalie Villa-Vialaneix "Random Forests for Big Data"

[3] Mohammad Arafath, "Random Forest Implementation in JAVA" [https://github.com/ironmanMA/Random-Forest]

[4] DataExpo,"http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236. "

[5] Bingjing Zhang, "Harp Tutorial"