Implementing Decision Tree Learning in Spark*

LiangLiang Zheng[1 [0548966]]

Vrije Universiteit Brussel, Department of Computer Science, Belgium

Abstract. Spark is a unified analytics engine for large-scale data processing. [1] In this report, an ID3 decision tree algorithm has been implemented in Spark scala to classify the star rating of particular products on Amazon. This report consists of the following parts: Topic introduction, Implementation, result and conclusion parts. After running the scala file on Isabelle, it has been found that the best performing running time is 2.7 mins using HDFS data with an accuracy of 87.14%.

Keywords: Spark · ID3 · Decision Tree · Dataframe · Scala

1 Implementation

1.1 Task 1: Feature Selection Categorical Attributes

The Decision tree is a classical supervised learning algorithm, which means it has the training set and ground truth in the training phase. Given the data set of Amazon product review information data set like review text, it is interesting and meaningful to predict star rating.

As mentioned in the abstract, the prediction target is the star rating. The features that will be used will be *review body*, *vine*, *verified purchase*, *total votes*, *helpful votes*.

Since decision trees are constructed from discrete or categorical values, it is natural that the next step is to transfer features with continuous value to categorical values. Detail encoding way has been illustrated in the list below.

- **review body** is transfer to character length and encoded with 0, 1 and 2 according to the length interval of [0, 100], [100, 201], [201, ∞].
- **helpful ratio** is take the ratio of helpful votes divided by total votes and encoded to 0 if its in the interval of [0, 0.5], otherwise 1.
- **vine** is encoded to 0 if it is N, otherwise 1.
- **verified purchase** is encoded to 0 if it is N, otherwise 1.

* CCBD: Big Data Project

1.2 Task 2 and 3: Decision Tree Learning (ID3)

The first thing of this project is to build up the ID3 algorithms. In the ID3 algorithm, find an attribute A with the highest information gain and then construct a new decision tree with attribute A, for all the possible value of this colum, filter out specific value and recursively build trees by calling the ID3 itself, the end condition of ID3 has 3 situations: The set of examples that need to be classified is smaller than some threshold, The list of predictive attributes is empty and All examples in data have the same value for the target attribute (zero entropy).

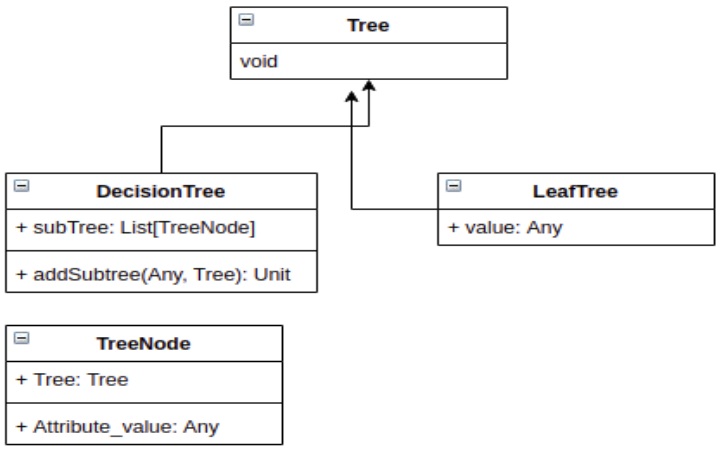

Fig. 1: UML of the decision tree classes

1.3 Strategy 1: Dataframe and pruning

RDD – RDD API is slower to perform simple grouping and aggregation operations. The data given for this project is apparently structured dataset, then dataframe is used here. Moreover, the syntax error of dataframe could be detected in the compile time while that of RDD could only be detected in running time. Spark SQL uses an optimizer called catalyst to optimize all the queries written both in spark sql and dataframe dsl. In addition, the pruning threshold in the ID3 end condition could also somewhat boost the computational efficiency when training.

1.4 Strategy 2: Persist Strategy

I used several persist in this project, the first two being used are after splitting training set and test set, these 2 data set will be used more than once in the training and test part. In the recursive ID3, after every filtering, it is also necessary to add persist, the final persist is at *IG* function, since the filtered data set

in information gain function will be used for contructing Decision Tree. In addition, when testing on cluster, I found out that specified the persist parameter with *StorageLevel.MEMORY_AND_DISK* boost the running time.

1.5 Strategy 3: Partition Strategy

When running on the cluster Isabelle, the reading file will cause partitioning. However, I used data from HDFS link, which contains more than 75GB uncompressed file stored on HDFS, then with the default HDFS block size setting (128MB) [2] it would be stored in 600 blocks, which means that the RDD which reads data from this file would have 600 partitions. So I set *spark.default.parallelism* to 600.

In my project, GroupBy will cause shuffle, when running locally, in order to reach a good performance when training on cluster, the *spark.sql.shuffle.partitions* has been set to 300.

1.6 Justification of using collect in this project

Inside the *BigDatProject.scala* file, collect has been used, one in IG function, one in H function. Since this operation is right after getting the count of the corresponding value in each attribute, the number of the value of an attribute will not exceed 5 in this project, so it will not affect the overall performance at all.

2 Result

Firstly, split the training set and test set with the ratio of 70% and 30% and then train it with the ID3 decision tree on Isabelle, the training accuracy on could vary from 86% to 90.1%, the result of running time with different parameters has been shown in the table below.

RDD or Dataframe	persist traing	persist test	persist IG	persist ID3	default.parallelism	shuffle.partitions	Running Time
RDD	Y	Y	Y	Y	600	300	more than 15 min
Dataframe	N	N	Y	Y	600	300	more than 15 min
Dataframe	Y	Y	Y	Y	600	300	3.2 min
Dataframe	Y	Y	N	N	600	300	3.1 min
Dataframe	Y(m,d)	Y(m,d)	Y(m,d)	Y(m,d)	600	300	2.7 min

It is noticeably to note that RDD is much slower than most of the dataframe when other parameters are controlled to be the same, which proves again that rdd is much slower than dataframe. Another thing to note is that persist removal of the training and test will largely affect the running time, because of the limited time provided in test slot, I killed the running node after 15 mins, given the size of the dataset, the running time of this case could reach up to more than 50 mins.

The best parameter combination is when setting all the persist to the stage level *MEMORY_AND_DISK* (m,d shown in the table), the running time has been optimized to the shortest 2.7 mins. While the performance when removing persist in Information gain and ID3 function is slightly better than that with persist in both functions, the reason of the result still remains unknown.

2.1 Weakness

There are 2 weaknesses in this project, firstly, there is no baseline for the experimenter to compare to, when getting result within 4 min in the first slot attempt, it took some time for the author to . Since *groupBy* creates shuffle, a more detailed partitioning like repartition approach should be implemented to boost the computational time.

3 Conclusions

This report show techniques when implementing classical machine learning techniques on spark and compare running time based on different strategies. Since the report focuses more on the strategy side, the feature selection and pre-processing part is not fully optimized , but it did bring some advantages, for example, the advantage of transferring features only to binary categorical or ternary categorical largely reduce the time of feature engineering part which saves time for further implementation of the dataframe optimization part. When running on cluster with large data set, it is wise to persist with memory and disk option in case of memory limitation. The reason why the removal of persist in the IG and ID3 part slightly boost the running time still remains unknown.

References

1. Shoro, A. G., Soomro, T. R. (2015). Big data analysis: Apache spark perspective. Global Journal of Computer Science and Technology.
2. 0x0FFF, Mar 12, 2015 (14:16), comment on Degget, "How does Spark partition(ing) work on files in HDFS?," Mar 12, 2015 (13:48), https://stackoverflow.com/questions/29011574/how-does-spark-partitioning-work-on-files-in-hdfs