Big Data Processing project report

Faras Jamil (Student id: 0551014)

January 26, 2020

1 Introduction

The goal of this project is to implement the ID3 decision tree algorithm in Spark using Scala. The project also required to configure the decision tree implementation on Isabelle and run it on Amazon Customer Reviews Dataset. To implement this problem we used two separate implementations using RDD and Dataframe and compared the results on the bases of cluster executions. To verify the implementation of the algorithms both were tested on play tennis (14 records) dataset, mentioned in the project description. Both RDD and Dataframe implementations created the same decision trees from the play tennis dataset, as explained in the project statement. That also verifies the future results predicted by our algorithms on Amazon Customer Reviews Dataset.

2 Implementation

The first step before starting implementation, we have to find the target variable for a helpful review, and other important features to classify a helpful review. Our target variable is ratio between *helpful_votes* and *total_votes*. If the ratio is greater then 0.5 the review is helpful, and lesser ratio reviews are not helpful. The other important features to classify a helpful review are *star_rating*, *vine*, *verified_purchase* and *review_body*.

After determining the feature space and target variable, the first step of implementation is to read and parse data. As the ID3 algorithm requires categorical values, we converted continuous features like *review_body* into the categorical feature. The three categories for *review_body* are divided based on text length less than 100, 100 to less than 300 and, other remaining. We also reduced *star_rating* categories from five to three by assigning the same label to 2,3 and 4 rating values. All selected attributes and their possible values are shown in table 1. Reviews

Attribute name	**Original value**	**Parsed value**
helpful_votes	Ratio of helpful and total votes	{true, false}
star_rating	{1,2,3,4,5}	{1,2,5}
vine	{T,N}	{true, false}
verified_purchase	{T,N}	{true, false}
review_body	The review text lenght	{1,2,3}

Table 1: Selected features with categories

are converted to *RDD[Review]* after parsing form raw data. Review is a class that used to contain the individual review object transformed from one row of raw data. Spark supports lazy transformations, so instead of persisting complete dataset and parse, we persisted after parsing.

Two different approaches used to implement the ID3 algorithm RDD and Dataframe because RDD is type-safe while Dataframe is untyped. For Dataframe implementation we transformed *RDD[Review]* to dataframe. As we also required to calculate the accuracy of the decision tree after training. We randomly split data into 70% train data and 30% of test data.

To build a decision tree from data, a tree data structure is required to save the nodes and their values. Figure 1 of decision tree class diagram shows that a tree has a node or a leaf. A node has a list of other nodes and an attribute name, sub-node can be added by *addSubTree* method. Leaf only has a value, and it is the end of a decision tree single branch, the value is a class label of the target variable.

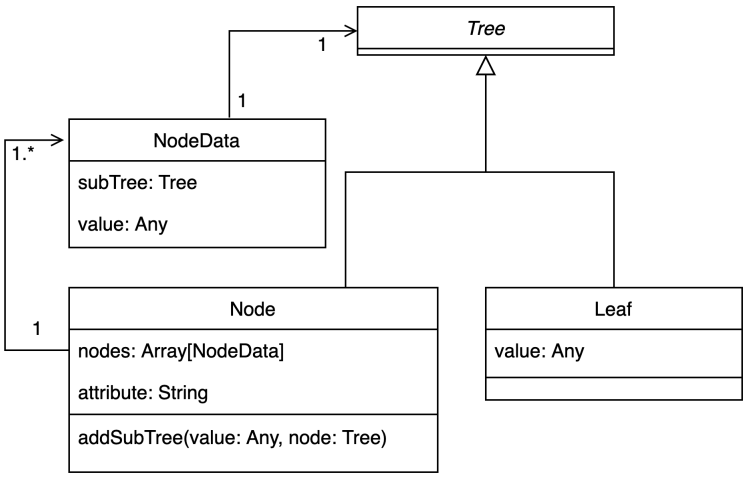

Figure 1: Decison tree class diagram.

The next step is to construct the decision tree from available training data. It is an iterative process until reached a leaf. It is implemented by following the pseudo code explained in the project statement. To enhance the performance count and entropy of whole data for each iteration is only calculated once and passed to the entropy(H) and information gain(IG) function in a parameter. Also, Dataframe and RDDs persisted after applying the filter in each iteration. The detailed architecture is shown in Figure 2 and explained in 4.1 section.

The last step after the training decision tree is to calculate the accuracy of the model on test data. This is done by mapping each row of test data on *predict* method. The predict method iterates a review on the trained tree model by matching attributes values and predict the leaf node value. The accuracy calculated by dividing correct predictions on the total count of test data.

3 Results

After completing implementations, first, we run it locally on a small dataset of 3.44 GB and later on Isabelle cluster using full HDFS data. There is some difference between each execution

time, hence to get a mean value each implementation executed five times. Results are shown in table 2, the mean values are 1.78 and 2.16 minutes for RDD and Dataframe respectively. The main goal for this project is to execute ID3 efficiently and achieve max possible performance, but accuracy also an important factor. The accuracy calculated on 30% test data for all executions is about 72.4% for both implementations. But the main focus is to improve performance without introducing anomalies in the ID3 algorithm. The performance was achieved by precisely persisting data at different levels without overdoing it. The rule we followed to persist an RDD or Dataframe only if it used more than once. Another approach applied in implementations is to avoid unnecessary duplicate calculations. Entropy and count values on the whole data used in ID3, H(Entropy) and IG(Information Gain) methods but first time calculated in the ID3 method, to avoid calculation duplication for the same values, these are passed to H and IG methods in parameters. Algorithm correctness achieved by producing the same tree from play tennis 14 records dataset as explained in the project statement.

Spark API	**1**	**2**	**3**	**4**	**5**	**min**	**max**	**mean**
RDD	1.9 m	1.6 m	1.8 m	1.8 m	1.8 m	1.6 m	1.8 m	1.78 m
Dataframe	2.1 m	2.1 m	2.2 m	2.2 m	2.2 m	2.1 m	2.2 m	2.16 m

Table 2: Results of executions on Isabelle cluster

4 Answers

4.1 persisting intermediate results

Persistence is an essential factor that helped us to improve the performance of our implementations. Figure 2 shows the Directed Acyclic Graph of RDD implementation, persisted RDDs are green while gray RDDs are simple transformations. White nodes denote values after performing some actions on RDDs. Dataframe implementation diagram is the same only change instead of RDD that has Dataframe.

Because Isabelle's allocated slots were very limited, persistent cases were only tested on the local machine during implementation and only final versions executed on Isabelle. Persisting the recursive function input RDD showed very significant improvement in performance. We measured the execution time on a small dataset of 3.44 GB and found the 15% of improvement. Without persisting ID3 function input RDD, it took 15 minutes in complete execution, and with persistence, it took about 1 min.

Persisting *RDD[Review]* after parsing improved the performance by avoiding an extra data read from file when *testRDD[Review].map(predict)*. This is not avoidable even persisting *testRDD[Review]*. The same dataset as mentioned above took 83 seconds without persisting and 56 seconds with persisting RDD[Review] after parsing. The extra 27 seconds is the reason for reading data twice.

Persisting *RDD[Perdication]* avoid mapping predict function twice on test RDD because two count actions are required to calculate the accuracy correct prediction(after the filter) and the total count of test RDD. On local test data *testRDD[Review]* one time mapping on predict function took 4 seconds.

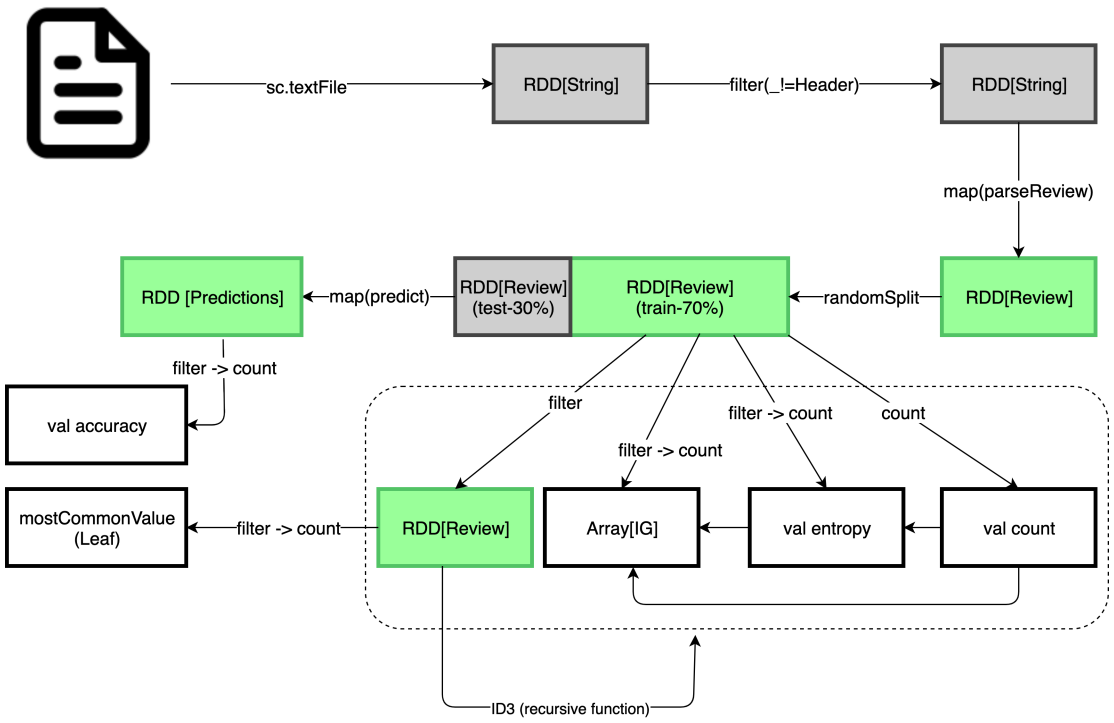

Figure 2: RDD implementation's Directed Acyclic Graph.

Persisting is important for the ID3 algorithm because this is a recursive algorithm, and at each step when an attribute is selected, it divides data into multiple parts. So it is a good idea to persist RDD after each data division we also proved this and got about 15% improvement in performance.

4.2 Partitioning

Partitions can only be customized for Pair RDDs, but you can change the number of partitions in RDD and Dataframes using *repartition(numPartitions: Int)*, it returns new RDD/Dataframe with new partitions. Partitions can be increased or decreased, Spark shuffle data to create new partitions. We examined this on a local machine by default the number of partitions is equal to the number of files reading from local storage. Reducing partitions cost down the performance while adding a few partitions doesn't show any difference, but after adding a lot of unnecessary partitions, it slows down the performance. On Isabelle, I used HDFS data already stored in partitions, and it worked well without changing the number of the partitions.

Partitions are more interesting in the case of Pair RDDs as we are using simple RDDs and Dataframe, we can't do more than just increasing and decreasing partitions. So I didn't define a specific number of partitions for this particular problem and stick with the default partitions. I tested by changing partitions on my local machine, and I didn't get any improvement, on Isabelle I used HDFS data, and it worked well with default partitions.

4.3 RDD and DataFrame

To implement the ID3 algorithm we used two different Spark APIs RDD and DataFrame, RDD is core data abstraction API in Spark, and both Dataframe and Dataset APIs run on top of it. We have semi-structured data, which gives us the freedom to use any Spark API because Dataframe and Dataset only work with structured or semi-structured data. The reason for picking RDDs and Dataframe two different APIs for implementing the task because RDDs are typesafe and don't provide and execution Optimizer while Dataframes are untyped but offer execution Optimization using Catalyst Optimizer.

RDD is a lower-level API for manipulating a distributed collection of data by allowing the in-memory computations on large clusters in a fault-tolerant manner. It helps in speeding up the task if implemented correctly by using lazy transformations execute when an action applied to them.

Datafame is a high-level Spark API for manipulating a distribution collection of structured data arranged into named columns. Like a table in the relational database, it allows executing the SQL queries over the data. Dataframe API designed to make the processing easier over large distributed datasets, and with the help of Catalyst optimizer, it rearranges the transformations and actions before executing it on RDDs to get maximum performance. Because Catalyst has the complete information of all data types with the schema details of data and also has the detail information of operation required to perform on data.

ID3 algorithm implementation doesn't require any complex operations over data this is the reason we didn't see any difference between RDD and Dataframe implementation performance. Even RDD performed better in our case because to calculate the probabilities of attributes we mostly required to filter and then count.