Realize a Decision Tree Using Spark

YICHANG XU, University of Science and Technology of China(USTC), China

Spark is a high-speed computing engine specifying in dealing with a large scale of data.It supplies a large amount of functions, such as sql querying, filtering, mapreduce, etc. Because of these diversified properties, spark is widely used in today's applications. In this article, I first realize a decision tree algorithm only using basic pyspark manipulations, then I write a corresponding pandas version and run both of them on LOL dataset to verify their correctness. After that, I compare the speed of the two models, analyze the reason of there difference and finally discuss the suitable occasions for using each model.

Additional Key Words and Phrases: decision tree, spark, machine learning, distributed dataflow

1 INTRODUCTION

When dealing with a huge amount of data, some clustering methods like MapReduce[1] and Dryad[2] has been widely adopted. These methods use parallel computing and hence accelerate the computation. However, none of them perform well in leveraging distributed memory[5], so Spark with resilient distributed dataset[5] is proposed, which is good at reusing intermediate results[5]. Moreover, Spark is equipped with various of tools, like SQL querying, MLlib, etc. It also supports dataframe, which is even faster than RDD.In conclusion, spark has following three features[3]:

(1) High level API helps to get rid of the attention of tackling clusters. So one only needs to focus on the computation when developing an application.
(2) Spark has a high speed and supports interactive computing and complex algorithms.
(3) Spark is a general engine which can deal with all kinds of computations conveniently, like SQL search.

A typical application of Spark is logistic regression, which performs 10 times faster than Hadoop[6]. But in some scenarios, Spark is not as useful as traditional platforms, such as applications that make asynchronous fine-grained updates to shared state[5]. To further explore the property of Spark and thus use it in correct occasions, in this paper, I realize a decision tree only using basic spark operations, i.e. select, filter, persist, and count. To illustrate its performance, I also wrote a decision tree using Pandas under the same code structure. Then I validated the algorithm and compared the running speed of these two models under different conditions, i.e. different sizes of the dataset and different constraints of CPU usage.

This article will be organized as follows: in section 2, I will explain some preliminaries like the basic usage of pyspark and the algorithm of decision tree. In section 3, I will introduce the core of programs and some skills to make them faster(especially pyspark program). In section 4, I made some experiments on pyspark and pandas version, and compared them in many aspects. I also fitted the results in curves using MLP and then predicted the best occasion to use spark to realize a decision tree.

2 PRELIMINARIES

2.1 Basic knowledge about pyspark

pyspark is a version of Spark that can be compiled on a Python interpreter. To start a spark session, we can use getOrCreate():

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.
    getOrCreate()
```

There are two typical structures in pyspark: RDD and dataframe. Since dataframe is faster, in this paper, I mainly use dataframe. And we don't

Author's address: Yichang Xu, xuyichang@mail.ustc.edu.cn, University of Science and Technology of China(USTC), Hefei, China, 230000.

need to create dataframes straightforward, instead, the program reads a csv file and then pyspark will generate a dataframe automatically:

```
origin=spark.read.csv(DATASET_NAME)
```

Then "origin" is a dataframe. To select certain rows from the dataframe, we can use filter() to filter out the rows that don't match the condition.

```
origin=origin.filter(origin._c0 != '
    index')
```

This will filter the row containing column names.

Note that pyspark dataframe cannot be indexed by numbers as pandas dataframe unless there is a specified row or column that is used to index. However, in this article, we don't need such a row or column.

To select certain columns, we can use select(). Just as select statement in SQL, you can apply some operations on the selected column. For example, select(df['col']+1) will select the column named 'col' and add 1 to each element.

count() is used to count the number of elements in a column. However, it is an action, which means that every time you call count(), the lazy functions you wrote before have to be called to calculate the dataframe, which may be really time-consuming, especially in a for loop dealing with dataframes with many lazy functions applied before. To speed up, we can use persist() to cache the data before entering the loop, and unpersist() after the loop. This technique will be discussed in detail in the next section.

2.2 Decision Tree

To better illustrate the algorithm of decision tree, first take a review at entropy.

Definition 1. *In a dataset S, denote the probability that a specimen belongs to class i as p_i, then the entropy of S is given by*

$$H(S) = -\Sigma_{i=1}^{C} p_i \log(p_i)$$

where C is the total number of classes.

Entropy represents the purity of a dataset.If it has a high entropy, we can conclude that the dataset is not pure, that is, many classes of data are mixed up.And vice versa.

As the word suggests, decision tree is a tree structure with nodes representing partitioning properties. There are two types of partitioning properties: classes and threshold. When a set of data flows into a node, it will be splitted into many parts, and each part enters its corresponding child node. When it attain the leaf of the tree, the corresponding data will be classified as the class of that leaf node. Usually, the leaf of a tree will be marked as the most common class of the training data it once contained.

But in fact, information gain is not the best critical for selecting a partitioner[7].Just think about a dataset with an index column. In this case, index will be chosen as the best partitioner since there is exactly one element belong to a certain index. C4.5 algorithm[4] does not use information, instead, it uses the gain ratio to choose the best partitioner.

Definition 2.

$$Gain\_ratio(D, a) = \frac{Gain(D, a)}{IV(a)}$$

where

$$IV(a) = -\Sigma_{v=1}^{V} \frac{|D^v|}{|D|} \log_2 \frac{|D^v|}{|D|}$$

a is the property used to partition, v is the values of a, and D^v denotes dataset with value v on property a.

C4.5 algorithm uses a heuristic search[4]:it first selects out the properties whose information gain are higher than the average, then it chooses one with the highest gain ratio.

For continuous properties, we can sort them in monotonic order, then compute the medium point between each two neighboring points, so that we can use them as thresholds to partition the dataset into two parts. Other procedures are similar.

Algorithm 1 Training Algorithm

```
For the training node P, suppose the training dataset is D, and the property set is A.
if P is the leaf then
    return
end if
if the specimens in D belong to the same category C then
    Make the node as category C
    return
end if
if $A = \emptyset$ or D have the same value in A then
    Denote the node as a leaf, and mark its category as the most popular one in A
    return
end if
a ⟵ the best partition attribute in A
if a is not continuous then
    for $a* \in a.values$ do
        Generate a child of the node
        D* := the subset which has a value a* on a
        if $D* = \emptyset$ then
            Mark the child as a leaf, and mark its category as the most popular one in D
            return
        else
            Let D* be the training data of this branch
        end if
    end for
else
    Generate two branches of the node:child[0] and child[1]
    for each sub dataset D' with attribute above or below the threshold do
        if $D' = \emptyset$ then
            mark the branch as a leaf, and mark its category as the most popular class in D
            return
        end if
    end for
end if
Train each generated node respectively
```

3 PROGRAM

The program consists of three parts:file-utils, decision tree and predicting program. Code can be seen on https://github.com/Kobe972/decision-tree. In this section, I only describe the structure of the program and some optimization I have done to accelerate the program.

3.1 Code Structure

file_utils.py mainly consists of a function generate_cross_verify(train,test), whose parameters are the proportion of training and testing dataset. For example, if we want to generate a cross validation dataset with training:testing=4:1, we just need to call generate_cross_verify(4,1). the function returns three parts of data: types, training data and testing data, where "types" is a numpy array with binary elements, and each element

Algorithm 2 prune(node)

```
if node is the leaf then
    return
end if
for child in node.child do
    prune(child)
end for
Denote node as leaf, and mark its category as the category with the most popular category among the training data
Calculate the accuracy
if the accuracy is higher then
    Decide to prune the node
else
    change the node back to the intermediate node
end if
```

indicates whether the corresponding property is continuous. For example, if the first property is continuous, then types[0]=1, otherwise types[0]=0. This function first shuffles the data and uses randomSplit() to randomly split the dataset into training data and testing data. It maintains the data of the first team and then aggregates the similar properties, i.e. the program adds up data of each player in the team such as player kills.After that, columns are renamed as the their property names, and continuous properties are turned into integer type. (when the file is loaded, all of the data is set to be string type by default).

decision_tree.py is the very core of the program. I defined two main classes here, Node and DTree. Node represents the nodes on the decision tree. It has three interfaces that are useful externally: predict(), compute_accuracy() and train(). I set a max depth of the tree. If the depth of one node exceeds the limit, that node will thus be asserted to be leaf and we should stop recurring on that node. Other component functions are commented in detail in the source code. To best represent the running speed of the program, I added a timer in the main loop of compute_classifier(), which computes the information gain and gain ratio of each classifier iteratively and takes most of the training time.

DTree is the only class that a user needs to use. It contains all of the data information and node information. In other words, we don't need to operate on Node directly. We can instead call APIs of DTree, i.e. train(), compute_accuracy() and predict(). In train(), you can choose whether the tree should make a pruning or not. The tree manipulates a post-prune strategy, i.e. the program first generate a full tree and then the prune function trys to delete each node from bottom to the top and see whether the decision tree works better on testing data. If so, the node will be deleted in practice.

Other two classes are not so important, but they are the basic structure of data and classifying properties. ClsProperty is the class containing information of classifying properties. It contains four members: name, type, values and thresh. name represents the name of the property, such as player_assists. type is a boolean denoting whether the property is continuous. values is an ndarray of the values that the data may have if the property is discrete and thresh is the threshold for classifying if the property is continuous. Data contains all of the information in a dataset. It only consists of three members: properties(dataframe(in spark version) or dict(in pandas version), containing the raw data), type and count(the number of samples).

predict.py is a program designed make a cross validation and compute its accuracy. Details can be seen in the source code.

I have written two versions of decision tree: pandas and spark version and tried my best to keep the structures of the two programs consistent. I just used different APIs in corresponding places. For example, when filtering certain data, I would use mask in pandas version while I use .filter() in spark version.

3.2 Optimizations

When the program was initially made, the speed of spark version was much slower than that of the pandas version. After inserting a timer to debug, I have found that the most time-consuming part is a for loop in the function that computes the best classifier of a node. I went into the loop and found that the delay could be attributed to the entropy calculating function. In the function, the count of a dataframe will be called for many times. However, when generating the dataset, there're some operations that are really time-consuming, like the summand of similar attributes. And since operators like filter, select are lazy, i.e. the program won't really compute them until an action like count() occurs, in this program, when counting the number of the sub dataset, the program has to repeat all of the preceeding procedures like select, summand, etc. So I made a small change, that is, before loading data to the node, I first use persist() to cache the selected data. And in order to release the burden of the memory, after training a node, unpersist() will be called to release the cached data and persist the corresponding in child nodes. By doing this, I found that the speed per iteration became almost three times faster than before. And the data processing procedure even sped up in a larger scale, since method gen_clsproperties() in class Node also calls count() in a for loop.

I also optimized IV() method in class Node. When calculating the IV value of a threshold of a continuous attribute, the function used to compute the ratios that the attribute is beyond the threshold and above respectively, which calls count() twice. I then changed to calculate the second ratio by just computing $1 - ratio1$, so that I could avoid one call of count().

And I revised the entropy function. I no longer used numpy to record each item in the summand. instead, I summed the values up just after calculating plogp. By making such Optimizations, the time usage per loop reduced from 6 seconds to 0.6 seconds on a dataset with 80000 items, almost 10 times faster.

I have tried to make other adjustments, such as adding persist() to each loop and rewrite entropy function in MapReduce form, but they had no effect of the running speed or even worse, the program became much slower than before. The reason is that persist() consumes more time than count() and reduce costs much more. In a loop that needs to iterate for many times, we should avoid using such operations.

4 EXPERIMENTS

The experiment mainly focuses on two things: change the capacity of the dataset and explore how the speed of pandas and spark version changes; limit the workers(CPUs) and observe how Spark version changes its speed of the loop mentioned before.

4.1 Datasets and Settings

The dataset is extracted of 80000 LOL games. Each row contains whether team1 wins and the performance of both teams and their members. The dataset is larger than 5MB, so it is uploaded as a release. In the experiment, I do not certainly use the whole data. In some cases, I only extract the first 100 items.To enlarge the dataset to GBs, I simply use concat.py that copys the dataset and append it over and over again. The main reason why I do not use the 80000 items directly is that some continuous attributes may have many candidate thresholds(for example, player_GoldEarned has about 10000 thresholds), which takes too much time to choose the best one.

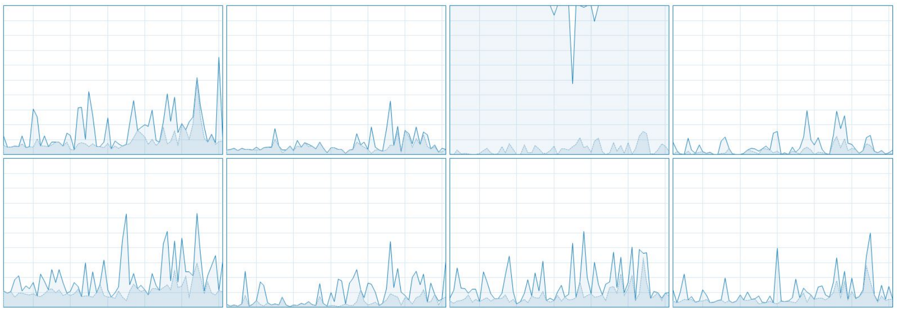
When testing accuracy of the model, I applied a cross validation where train:test=4:1. I also added some print() and a timer into the program so that it prints the main step that it is now working on and how long it spends on some key operations. The max depth of the decision tree is set to 20 and I applied pruning after training.

4.2 Test the program

In this section, I test the pandas version with the whole dataset and test the Spark version with the first 100 items, since Spark is slower when dealing with a small amount of data. I tested each program 5 times and then computed the mean accuracy. The mean accuracy of both version were all 0.96.

4.3 Alternate the Size of the Dataset

In the preceeding experiment, I found that the Spark version was much slower than pandas version. This is because Spark is a distributed data system. It will spend some time distributing works for each worker. To verify that the workers in Spark program cooperates well, I monitored the 8 physical CPU cores on the laptop.


(a) Spark Version

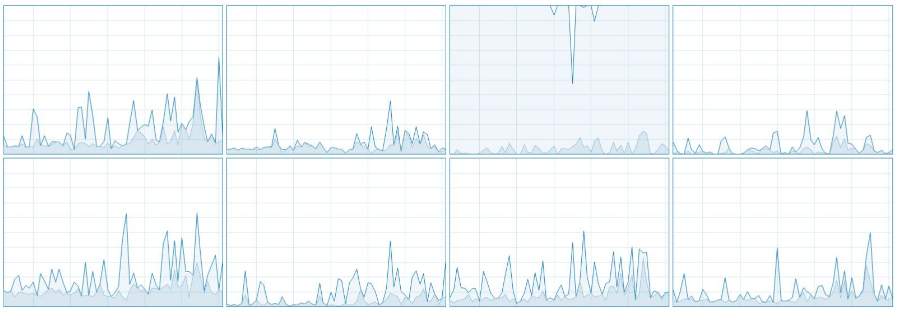

(b) Pandas Version

Fig. 1. CPU utility of each version

From figure 1 (a) and (b), we can see that the CPU utility of the Spark version is 100% and that of pandas version is only about 25%(in fact, the monitor shows that the accurate utility ranges from 23% to 26%), which is much less efficiency than Spark version.

Although Spark takes some time to make an

Table 1. Recorded Data of Time Consuming(s)

Size	Spark Version	Pandas Version
34.7K	0.35	almost 0
3.29M	0.35	almost 0
32.9M	0.47	0.01
66.0M	0.50	0.02
132M	0.55	0.03
264M	0.59	0.06
529M	0.68	0.12
1.03G	0.79	0.25
1.45G	0.97	0.35
1.94G	1.10	1.46
2.42G	1.31	0.57
2.91G	1.44	0.68
3.23G	1.56	0.75

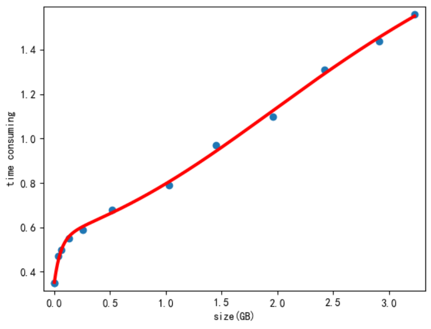
ideal distribution, it may be more efficient when dealing with large amount of data than pandas. So I enlarged the dataset and tested the time usage per iteration. When the dataset grows to 4GB or more, the pandas version can't properly run because the data is out of memory. However, the Spark version can run even if the size surpasses 10GB. The reason is simple: pandas stores the entire data after loading, while Spark only stores the filtered data after persist(), which is just a small fraction of the original data. This is an additional benefit of Spark. To complete the test, I only varied the size of data from 34.7K to 3.23G, and recorded the speed of these two versions respectively. See Table 1 for detailed data.

To sense it visually, I also fitted the data points in a curve using MLP (code can be seen on GitHub), as seen in figure 2.

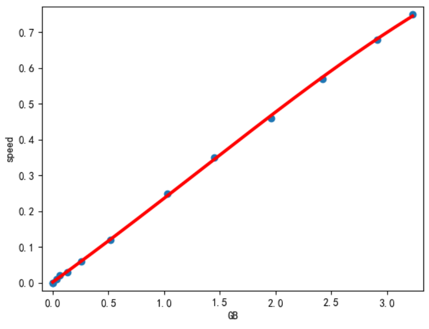
To our disappointment, it seems that The speed of Spark version can't catch that of pandas version even if we expand the dataset. Indeed, after computing the increasing rate of the two methods, we found that they are quite similar. I guess the reason of this phenomenon is that there are still no enough CPU cores so that Spark can't make full use of its strength. Then I went on the next experiment to explore the relationship between the number of CPU cores and the executing speed.

4.4 Limit the Workers

I set the master when creating the session to limit the CPU cores that the program is available to use. Similar to the last experiment, I made a table and plotted a graph(Table 2 and figure 3). The size of dataset is set to 2.94GB. Notice that the speed when 8 CPUs available is slower than before. This is because there are redundant persist() in the program and had not been deleted yet when doing this experiment. However, it has almost no influence on the regularity that we're going to find.



(a) Time Consuming of Spark Version

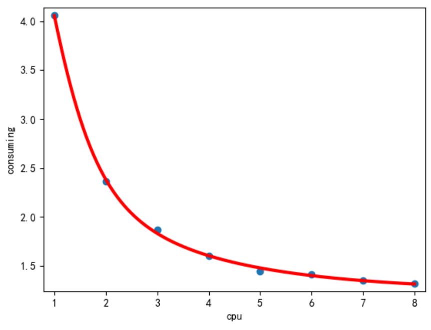


(b) Time Consuming of Pandas Version

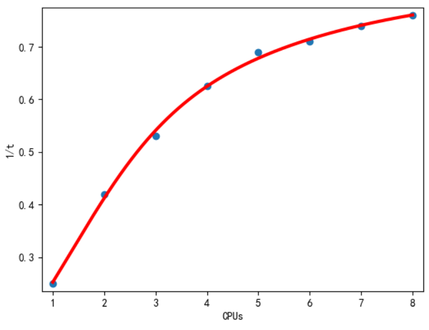
Fig. 2. Time consuming of each program when changing the size of data

Table 2. Time Consuming(s) When Limiting CPUs

CPUs	Time Consuming
8	1.32
7	1.35
6	1.41
5	1.44
4	1.60
3	1.87
2	2.36
1	4.06



(a) Original Graph



(b) Reciprocal on y Axis

Fig. 3. Time consuming when using different number of CPUs

I found that as more CPUs available, the speed became faster, but the increasing rate is decreasing. In order to predict the CPU cores the Spark version needs to surpass the speed of the pandas version, I inverted this relationship and used a similar MLP to predict(see predict_cpus.py in source code). I computed that the Spark version needs 28 CPUs to be faster than pandas version. Although in a single laptop, this is almost impossible, if we run Spark on a cluster, this condition could be easily attained, since a cluster usualy consists of at least 4 computers with 32 CPUs in total.

From the experiments above, we got the following results:

(1) Spark is excellent in dealing a large amout of data, since most of its operations are lazy and it doesn't need to store the entire data at the beginning.
(2) It takes some time for Spark to make a distribution on each worker. So if you're running a small application, avoid using Spark.
(3) Spark has its strength when working on clusters. So try to increase your workers when Spark doesn't work well.

5 CONCLUSION

In this paper, I first realized a decision tree algorithm both on pandas and Spark, and verified its correctness by making cross validation. The accuracy was high, meaning that the algorithm was correct. Then I compared the speed and CPU utilities of the two programs. From the preceeding analysis, we can see that Spark can make CPU work much more efficiently than pandas. However, the assigning procedure takes time. So it's not a good choice to use Spark when CPU is not vital to your program. And the latter experiment shows that increasing the number of workers is very helpful, since CPU can be multiplied through connecting many computers and the advantage of distribution can be greatly revealed. The main contribution of this paper is that it tells how to write a complex machine learning program in Spark and how to choose between Spark and common platforms if the program is not as simple as a single MapReduce.

REFERENCES

[1] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[2] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (Lisbon, Portugal) *(EuroSys '07)*. Association for Computing Machinery, New York, NY, USA, 59–72. https://doi.org/10.1145/1272996.1273005

[3] O'reilly. 2015. Learning spark lightning-fast big data analysis. *Oreilly and Associates Inc* 10 (2015), 375.

[4] J. R. Quinlan. 1995. Quinlan, J. R. : C4.5 Programs for Machine Learning, Morgan Kaufmann, San Mateo, California (1992). *Journal of Japanese Society for Artificial Intelligence* 10 (1995), 475–476.

[5] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[6] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[7] Zhihua Zhou. 2016. Machine Learning. *China Civil Entrepreneur* 03, No.21 (2016), 93–93.