INFO3406 Assignment 2 Report

Introduction

With the advent of big data, how to improve the capacity of processing massive data has been a research focus. Currently, the most widely-accepted resolution, in terms of improve computational capacity, is distributed computing. Among various plans about parallel computing, Hadoop which firstly brought up by Google and Apache Spark, are commonly implemented in academia and industry. The importance of mastering related technique is increasingly recognised by the data scientist communities all over the world.

This study aims to implement Machine Learning algorithms on a distributed-computing basis. With the tuning of parallelization configurations, the best distributed computing plan would be investigated and discussed. The performance of Machine learning algorithms is also going to be viewed and discussed. Through this research, the practical experience on constructing distributed-computing system and related theoretical knowledge will be gained. This would undoubtedly enable researcher to acquire state-of-the-art techniques, and also makes them more competitive and capable to cope with the challenges of data explosion.

Methods

A. Locality-sensitive hashing (LSH)

Hashing, as a common method to map raw data into data of fixed size, have various applications in different areas. In terms of cryptography, the cryptographic hash function should be collision resistant. That is the reason why hash functions like MD5 tend to map data into buckets evenly.

Locality-sensitive hashing, which is also a common technique used in Machine Learning, is named by its unique features of local sensitivity. In other words, the hashing function used in LSH is locally sensitive. That is, the locality sensitive hashing function, different from those hashing functions like MD5, is more likely to map similar items into same buckets. There are various LSH algorithms based on different distance metrics; certainly, the use of different distance metric also leads to different locality sensitive functions. One of instance would be that the Min-hashing algorithms is based on the Jaccard similarity.

In this research, different from the use of common LSH algorithm like Simhash, Nilsimsa Hash will be used in order to reduce the dimensions and lighten the computation. Nilsimsa hashing is also a LSH algorithms and the most interesting part is that making a small modification to data does not substantially change the resulting hash or signature, compared with the original one. The detailed algorithms is as followed:

```
$H = [\,0,0,0 \ldots, 0]\ initialise\ H_1, H_2, H_3 \ldots, H_l$
$\boldsymbol{for\ all}\ characters\ i\ with\ i = 0,1 \ldots, p-1-w, \boldsymbol{do}$
    $\boldsymbol{for\ all}\ trigram\ from\ window\ \{\,B_i, \ldots, B_{i+w-1}\}\ \boldsymbol{do}$
        $H[Hash(trigram) mod\ l] + +$
    $\boldsymbol{end\ for}$
$\boldsymbol{end\ for}$
$h = sign(H)$
```

***Algorithms 1** : the Nilsimsa hash algorithm*

As we can see, Nilsimsa algorithm calculates a hash digest for a text or a number by taking a trigram of characters of text or digits of number within a sliding window moving over the text or the number as input. With the help of this moving window, similar data would be mapped into adjacent buckets instead of distant ones.

Originally, Nilsimsa algorithm would map data into $2^{15} = 32768$ buckets; in the code, we could choose the number of buckets with the help of MODE operation. The corresponding code is in the part called NilsimsaHash().

B. K-nearest neighbours and distance metrics

The data used in this research, which is Display Advertising Challenge dataset, mostly is categorical. After being processed by LSH, the features are still measured discretely. In other words, Euclidean distance does not suit the data we used here. Therefore, Manhattan distance, is going to be applied in this case.

Firstly, we used the Sum of Absolute Differences (SAD) to compare the test data and training data. To be specific, the SAD between two data points is:

$$D_{SAD} = \sum_{x,y,c} |\, TestingData\ j - TrainingData\ i\,|$$

The smaller the SAD is, the more similar two data points are. It is also more likely that two data points share the same label. The K in this algorithms, is the number of training data points used to predict the query data. In some cases, there may be reweighting in those K data points based on their distance from the query data. However, in this research, the weight for every training data point is the same.

K-nearest neighbours algorithm can be briefly described as:

```
$A\ positive\ integer\ \boldsymbol{k}\ is\ specified, along\ with\ a\ new\ sample$
$We\ select\ the\ \boldsymbol{k}\ entries\ in\ our\ database\ which\ are\ closest\ to\ the\ new\ sample$
$We\ find\ the\ most\ common\ classification\ of\ these\ entries$
$This\ is\ the\ classification\ we\ give\ to\ the\ new\ sample$
```

Algorithms 2: the K-nearest neighbours algorithm

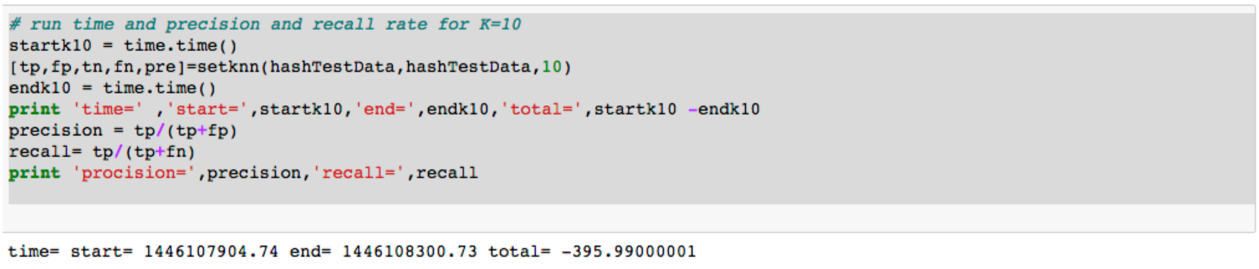
Experiments and analysis of results

A. Speed

Firstly, we are going to explore how the number of neighbours would affect the running time. The analysis is same as controlled experiments. In other words, we only change the number of parameter K and ceteris paribus (a.k.a with other things the same).

We run the code to classify totally 991 testing records and the classifier is built on 1000 training records. We chose the K values of 1, 3, 5, 10, 30, 100 and used time.time() function in Python to calculated running time. The outcome is as follows:

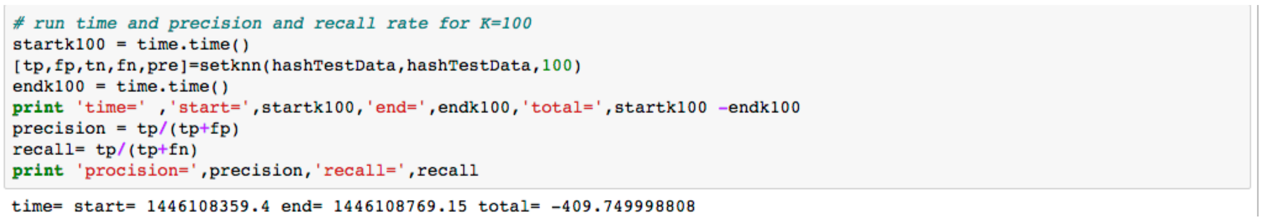
The least time used by the program is 396 seconds with the k equal to 10.



```
# run time and precision and recall rate for K=10
startk10 = time.time()
[tp,fp,tn,fn,pre]=setknn(hashTestData,hashTestData,10)
endk10 = time.time()
print 'time=' ,'start=',startk10,'end=',endk10,'total=',startk10 -endk10
precision = tp/(tp+fp)
recall= tp/(tp+fn)
print 'procision=',precision,'recall=',recall
```

```
time= start= 1446107904.74 end= 1446108300.73 total= -395.99000001
```

The most time used by the program is 410 seconds with the k equal to 100.



```
# run time and precision and recall rate for K=100
startk100 = time.time()
[tp,fp,tn,fn,pre]=setknn(hashTestData,hashTestData,100)
endk100 = time.time()
print 'time=' ,'start=',startk100,'end=',endk100,'total=',startk100 -endk100
precision = tp/(tp+fp)
recall= tp/(tp+fn)
print 'procision=',precision,'recall=',recall
```

```
time= start= 1446108359.4 end= 1446108769.15 total= -409.749998808
```

Here is the table and line graph for time needed to run program with the change of K:

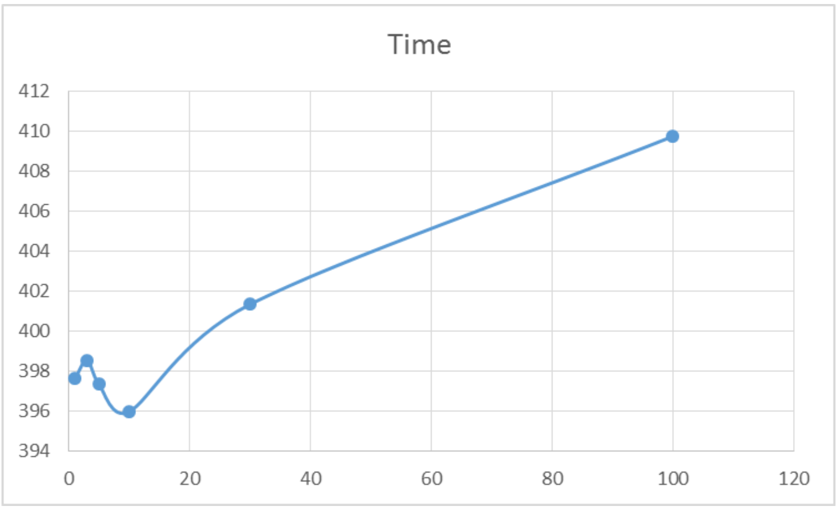


Figure 1.K verses Time graph

K	1	3	5	10	30	100
Running Time	397.64	398.56	397.36	395.99	401.35	409.74

Table 1. K verses Time table

We can see that the time needed fluctuates with the increase of K and hits the bottom around K equal to 10. After that, the required time is increasing with the inflating K. It can be concluded that the optimal K is around 10 and with the increase of K, time required to run the code is correspondingly inclining.

One plausible reason for this is that when K under 10 the paralleled drivers in Spark is more likely handle the computation simultaneously. That is the reason why the time does not changes a lot when K is changing blow 10. However, when K is over 10 or more, even the paralleled drivers in Spark cannot compute data as a whole. This requires to separate the load and compute them step by step. In other words, Spark can only cope with the computation with k nearest neighbours at most. As long as the real K is under the optimal number, the program would not run with the sacrifice of time. However, if the real K is over the optimal number, spark still needs to consume more time to complete the computation just like singe threading program.

When it comes to the number of cores, it is more flexible and the difference in running time is subtle. The changes of core number is completed in command line with the command like:

--executor-cores 1

--executor-cores 4

--executor-cores 10.

Similar with the experiment we run before, we hold all conditions unchanged and only try to change the number of core used in Spark. The training set and testing set is also just same as before. Here is the change in time with changing number of cores.

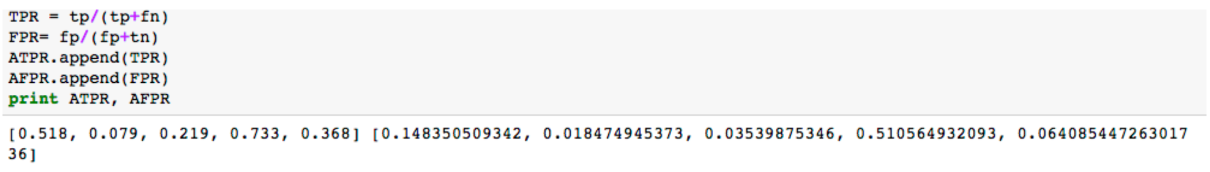
Cores number	1	4	10
Running Time	405.36	395.41	403.52

Table 2. Core number versus time table

We can see here 4 is the optimal number of cores to run the programme. The reason for this is that the Spark is running on my Laptop with Intel's i5 processors (Intel's i5 processors is four-core processors). In Spark, each driver pregame is allocated into one core and there would no more waste of resources on task-serialization (like one-driver Spark) or unbalanced resource allocations (like 10-driver Spark).

B. Classifier accuracy

5-fold cross validation is used here to depict the ROC curve. The outcome is as follows:



```
TPR = tp/(tp+fn)
FPR= fp/(fp+tn)
ATPR.append(TPR)
AFPR.append(FPR)
print ATPR, AFPR
```

```
[0.518, 0.079, 0.219, 0.733, 0.368] [0.148350509342, 0.018474945373, 0.03539875346, 0.510564932093, 0.06408544726301736]
```

The specific number can be viewed in the follow table:

Fold	TPR	FPR
1	0.079	0.018
2	0.219	0.035

3	0.368	0.064
4	0.518	0.143
5	0.733	0.510

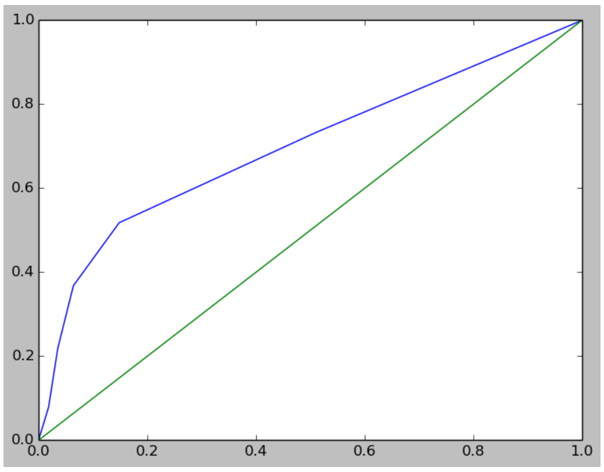
Table TPR and FPR

Related indicators are calculated as follows:

$$Precision = \frac{TP}{TP + FP} \quad Recall = \frac{TP}{TP + FN}$$

The last record is used as the measurement of accuracy (FOLD 3), whose precision rate is 0.657 and recall rate is 0.368. We can see the precision of KNN is pretty high, which reach 0.657. It is reasonable to say that the performance of classifier is satisfactory.

Also, corresponding ROC is generated by Python, which can be viewed in the following graph (Graph ROC). In the construction of ROC, two dummy points (0, 0) and (1, 1) are used to show the complete axis.



Graph ROC

Discussion

With the help of this research, the practical experience and related theoretical knowledge on Spark has been gained. It can be seen that Parcelled Computation is the most promising resolution to Big Data. In practice, it proves that time required to run the program can be lessened significantly with the help of Spark or Map-reduce technique.

Also, Spark or Hadoop is relatively new technique and related resources does not as sufficient as other mature techniques. This requires us to gather information on our own and this also highlight the importance of self-learning. This is especially the case when it comes to MOOC. The online open course from Edx, *CS190.1x Scalable Machine Learning*, really benefits me deeply.

Besides, through this research, it is found that the types of data in real world is widely ranged. The data we need to handle may be not only limited into numeric data. Like in this research, the way to cleanse and process data is never seen by me before. This includes the use of sparse vector and Label-point in PySpark. My experience and skills on handling large discrete data have been improved a lot.

Conclusions and future work

It can be seen that the MapReduce technique indeed is able to decrease the running time of Machine learning Algorithm. Apart from the analysis of data, the scalability of algorithms is also deserve attention. MapReduce is undoubtedly is a necessary skill which is needed mastering. For Knn running on distributed systems, there would be an optimal K to minimise running time. The K in this research is around 10. Below this, the parallel computation may be not reach the most efficient stage and above this, the time for running code may be extended. The tuning of parallelization configurations is actually a systematic science; it requires constant Trial and error.

In terms of future work, the first thing should be about gaining more experience about MapReduce. In this assignment, we have implemented this technique via virtual machine. Still, we need to implement MapReduce in real existing multi computers. Also, in this case, we only used KNN as the classifier and we have seen its performance on paralleled computation. It should be noticed that other algorithms, which is not distance-based, may perform differently under Map-Reduce technique. Future work may focus more on the performance of Tree-based and other Machine learning algorithms on distributed systems.