THE UNIVERSITY OF SYDNEY

School of Information Technologies
Faculty of Engineering & IT

ASSIGNMENT/PROJECT COVERSHEET - GROUP ASSESSMENT

Unit of Study: COMP5349

Assignment name: Assignment 2 Spark Machine Learning Application

Tutorial time: 4 — 6pm Thu **Tutor name:** Andrian Yang

DECLARATION

We the undersigned declare that we have read and understood the *University of Sydney Academic Dishonesty and Plagiarism in Coursework Policy*, an, and except where specifically acknowledged, the work contained in this assignment/project is our own work, and has not been copied from other sources or been previously submitted for award or assessment.

We understand that failure to comply with the *Academic Dishonesty and Plagiarism in Coursework Policy* can lead to severe penalties as outlined under Chapter 8 of the *University of Sydney By-Law 1999* (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgement from other sources, including published works, the internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

We realise that we may be asked to identify those portions of the work contributed by each of us and required to demonstrate our individual knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

Project team members				
Student name	**Student ID**	**Participated**	**Agree to share**	**Signature**
1. Dongdong Zhang	470161133	Yes / No	Yes/No	
2. An Wang	460350682	Yes / No	Yes / No	
3. Zizhao Wang	470296136	Yes / No	Yes / No	
4.		Yes / No	Yes / No	
5.		Yes / No	Yes / No	
6.		Yes / No	Yes / No	
7.		Yes / No	Yes / No	
8.		Yes / No	Yes / No	
9.		Yes / No	Yes / No	
10.		Yes / No	Yes / No	

SIT Building, J12
The University of Sydney
NSW 2006 Australia

T +61 2 9351 3423
F +61 2 9351 3838
E sit.info@sydney.edu.au
sydney.edu.au/it

ABN 15 211 513 464
CRICOS 00026A

COMP5349 Cloud Computing Assignment 2

Spark Machine Learning Application

SIT 115 2

Dongdong Zhang	470161133
Zizhao Wang	470296136
An Wang	460350682

Table of Contents

Assignment 2: Spark Machine Learning Application

Introduction

This paper presents the implementation of a spark machine learning application that making classification for MNIST data set, as well as the performance analysis of some experiments. There are three stages in this project. In the first stage, a KNN classifier with a PCA module inside is constructed. In the next stage, 12 experiments based on different combinations of variables are implemented to analysis the impact that each variable may cause on the result. In the last stage, 2 classifiers, random forest and multiple layer perceptron are applied on MNIST data set to compare their performance and observe the influences of adjusting parameters in each classifier.

1 KNN Classifier

1.1 Introduction

There are three main steps in stage one. Firstly, data formatting aims to format the given data file to DataFrame that enables the further operation among the data. Secondly, PCA (Principal Components Analysis) compresses the large amount of feature dimensions to more concentrated ones. Finally, classifying testing data by KNN classifier. The design structure is shown in Figure 1.

1.2 Data Preprecessing

1.2.1 Data formatting

The training file and testing file are read in as the DataFrame format. Both of them have 785 columns, the first column is label and the rests correspond to the image's 784 pixels. After read in, the last 784 columns in each DataFrame is assembled to one column and given a name "features", this step helps to call all features directly in the future operations.

1.2.2 PCA

There are too many features in the dataset, and the majority of them are zeros. To reduce the redundant features and promote the efficiency of the following classification section, we implemented PCA to reduce the feature dimensions.

PCA aims to compress a large, redundant dataset to a smaller, concentrated one. The central idea of PCA is transforming the initial feature dimensions to a set of vectors that maintain the largest data variance of the whole dataset. The set of vectors are called **principal components**. And the values of initial features are projected on these principal components.

In our implementation, after assembler, we import PCA from pyspark, and set the number of principal components to *d*. We use the training DataFrame to orient the d principal components, and get a compressed DataFrame "train_vector_pca". Then we transform the testing DataFrame based on these *d* vectors. This method insures that the transformation of both training and testing data have the same baseline.

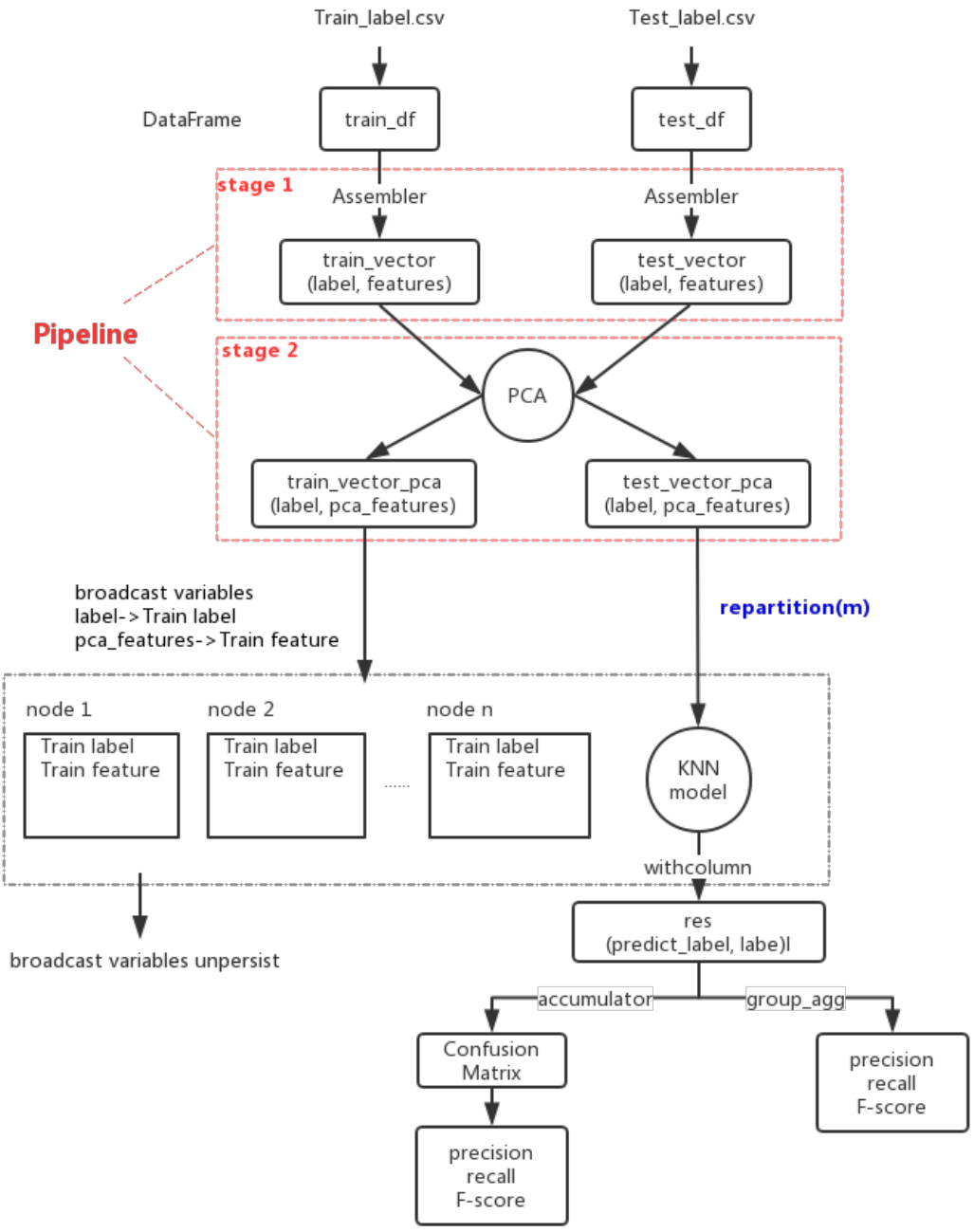

Figure 1. Design Structure

1.3 KNN Classification

We constructed a KNN model with multiple distance calculating methods, main distance method is Euclidean distance. We set the k value as *k*, which means we will predict the testing data's label according to its nearest *k* training data. As the following codes show: func takes the input data frame columns row by row to making the accumulation and returns the predicting labels. The model finally returns a list of prediction result which is added as a new column "predict_label" to test_data by "DataFrame.withColumn" function. A result that contains prediction column and real label column is generated for further evaluation.

```
def predict(self, test_data):
    func = udf(self.knn_spark_fit)
    return test_data.withColumn('predict_label',
           func(test_data.columns[1])).select('predict_label', 'label')
```

The code of KNN model is shown below, that the numpy.argpartition function returns the index of the bottom k values, then numpy.take function returns the *k* labels corresponding to the values. The most frequent number among these *k* labels is the result.

```
dist = np.linalg.norm(
       trainFea.value - np.array(features), axis=1).reshape(trainLab.value.shape)
topK_label = np.take(trainLab.value, np.argpartition(dist, K_knn, axis=0)[:K_knn])
counts_test = np.bincount(topK_label)
Most_test = np.argmax(counts_test)
```

1.4 Parallelization

In this project, parallelization occurs in the csv file reading, assembler, PCA, pipeline operation, and withColumn operation for KNN classification part. In the withColumn operation, the testing data will be distributed to different executors to do the classification while the entire training data is required to participate in the calculation in every node. To enhance the efficiency in this process, we transform training features and training labels to broadcast variables which will be placed at every node as read only variables in advance, when they are called, they can be read directly from each node locally. In order to calculate precision, recall and f-score, the accumulator operation also is parallelization.

1.5 Optimization

We have tried some methods to optimize the project. Adding pipeline component, repartition, broadcast, persist, accumulator. They have contributed significant improvements on the project, the running time of this project reduced from 4 mins to 1.2 mins after implemented the optimizations.

1.5.1 Pipeline component

We integrate the data formatting part (assembler) and PCA as a workflow, these two parts become two stages in pipeline, the code is shown below.

```
assembler = VectorAssembler(inputCols=train_df.columns[1:], outputCol="features")
pca = PCA(k=k, inputCol="features", outputCol="features_pcas")
pipeline = Pipeline(stages=[assembler, pca])
model = pipeline.fit(train_df)
train_pca_result = model.transform(train_df).select(
       col(train_df.columns[0]).alias("label"), "features_pcas")
test_pca_result = model.transform(test_df).select(
        col(test_df.columns[0]).alias("label"), "features_pcas")
```

Then the pipeline acts as an estimator, it is fit to the training data frame and then generate a transformer “model”, the model can be used to transform raw training and testing data frame directly to the format that contains label and principle component features.

1.5.2 Repartition

The testing data frame are manually divided to m parts equally by the repartition function to avoid uneven tasks distribution on the nodes. The value of m can be adjusted to achieve the best program performance, which will be explain in stage two.

1.5.3 Broadcast

Before operation withColumn for KNN, it need train label and features. In order to escape shipping a copy of train data to each task, broadcast is used to transfer variable to each executor. This method can reduce the transmission time and reduce memory usage.

1.5.4 Persist

The persist operation is used in this implementation. The result dataframe element (predict_label, label) will be persisted after collect action. In order to calculate accuracy and other action in the future, it can be reused quickly.

1.5.5 Accumulator

The accumulator operation is used in this implementation. During withColumn operation, except collect result to calculate confusion matrix, the matrix can be recorded with accumulator. Therefore, accumulator for confusion matrix can reduce implementation stage to calculate precision recall and f-score.

1.6 Result

The sample results Table 1 as follow:

Table 1. Sample Result

PCA: 50 K: 5				PCA: 50 K: 10			
label	Precision	Recall	F_measure	label	Precision	Recall	F_measure
0	0.9759	0.9918	0.9838	0	0.9778	0.9908	0.9843
1	0.9724	0.9947	0.9834	1	0.965	0.9965	0.9805
2	0.9814	0.9729	0.9771	2	0.9881	0.968	0.978
3	0.9692	0.9653	0.9673	3	0.9702	0.9683	0.9693
4	0.9796	0.9776	0.9786	4	0.9765	0.9745	0.9755
5	0.973	0.9709	0.9719	5	0.9742	0.9742	0.9742
6	0.9793	0.9885	0.9839	6	0.9762	0.9864	0.9813
7	0.9688	0.965	0.9669	7	0.9667	0.9601	0.9634
8	0.9801	0.96	0.9699	8	0.9842	0.9569	0.9703
9	0.969	0.9594	0.9641	9	0.9633	0.9623	0.9628
PCA: 100 K: 5				PCA: 100 K: 10			
label	Precision	Recall	F_measure	label	Precision	Recall	F_measure
0	0.973	0.9929	0.9828	0	0.9672	0.9918	0.9793
1	0.9683	0.9956	0.9818	1	0.9577	0.9965	0.9767
2	0.9795	0.97	0.9747	2	0.9841	0.9612	0.9725
3	0.9711	0.9644	0.9677	3	0.9721	0.9673	0.9697
4	0.9794	0.9674	0.9734	4	0.9764	0.9684	0.9724
5	0.9676	0.9709	0.9692	5	0.9719	0.9686	0.9702
6	0.9793	0.9885	0.9839	6	0.9772	0.9843	0.9808
7	0.9642	0.9689	0.9665	7	0.9631	0.964	0.9635
8	0.984	0.9476	0.9655	8	0.9841	0.9507	0.9671
9	0.9622	0.9584	0.9603	9	0.9631	0.9574	0.9602

2 Performance Analysis

2.1 Accuracy evaluation

Because the KNN is a lazing learning, and the algorithm doesn't have any random stage, so the accuracy results would be same in the same parameters of PCA and knn as table 1.

Due to every combination evaluation value are more than 90%, in the later part, we wouldn't talk about every combinator influence the accuracy evaluation.

2.2 Parallelism evaluation

We set 12 combinations of different values of reduced dimension *50* and *100* in PCA, nearest neighbour number *5* and *10* in KNN classifier, executors number *e* and executor-cores number *c* is *[(e=4, c=4), (e=8, c=2), (e=8, c=4)],* to obverse their effect on the aspects of prediction accuracy, parallelization, task execution time and I/O cost. In order to analyse each parameter whether influence the execution property, we analyse each parameters performance and fix separately.

2.2.1 KNN stage and repartition

In the KNN stage, we implemented withColumn of spark dataframe to achieve it, every hyperparameters and execution properties can be parallelism. In the KNN stage, with the 8 executors and 4 cores, the repartition number of test data was set one time of core number's result as follow:

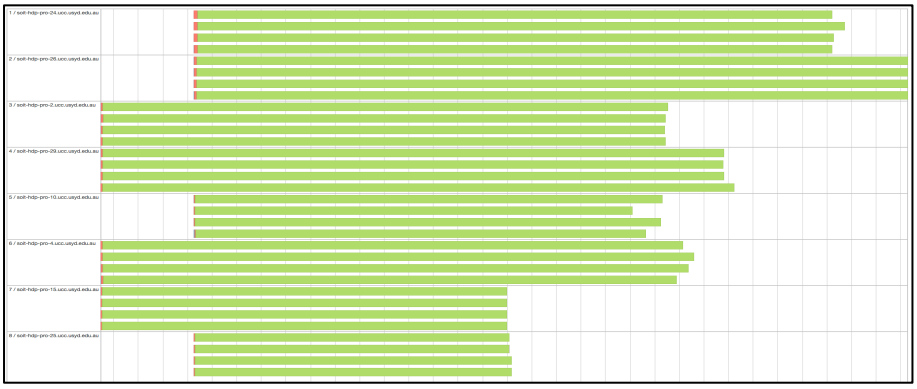

Figure 2. repartition 32 (1*8*4) event time line in KNN stage

Above is one time of core numbers, each part can be running in parallelism. Figure 3 is the repartition number was set double core numbers, as follow:

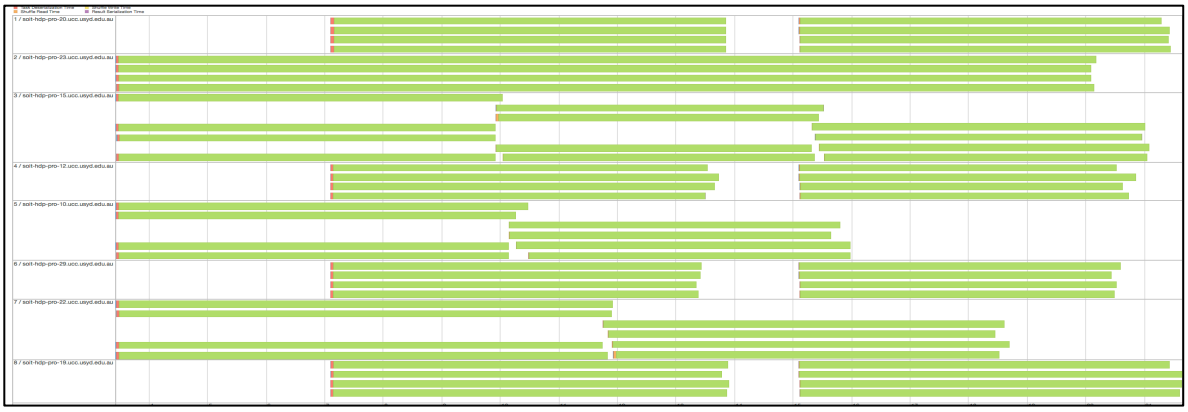

Figure 3. repartition 64 (2*8*4) event time line in KNN stage

With the double core numbers, each part also can be running in parallelism. Besides, in each core, if one small part data finished earlier, this core would receive next small part data to run continually, it is efficient that every core would be usage maximumly rather than unused when core task finished, like Figure 3 repartition 32.

2.2.2 The value of the nearest neighbour k (5, 10) and shuffle read

- *The influence of k value*

Based on fixed every hyperparameters and execution properties except k values of KNN, then we compare the k values which are 5 and 10 then summarized the executors' performance and execution time Table 2 as follow:

Table 2. execution time and total I/O cost with 50&5

PCA	K	Executor & Core	repartition	execution time		I/O cost		
				knn time	Time (mins)	Input (MB)	Shuffle Read	Shuffle Write
50	5	8 & 4	64	25s	1.3	470.6	4.6 MB	28.5 MB
50	10	8 & 4	64	28s	1.3	442.2	16.2 MB	28.5 MB

According to this table, we found the shuffle read values are different, but others weren't influenced. So next part analysis the shuffle read differenced reason.

- *The reason of shuffle read difference*

In the Spark UI, the executors' performance data showed that if tasks are concentrated in one executor, the shuffle read is lower than tasks distributed evenly. Then we found the PCA stage could distributed randomly results in the total tasks unevenly as below:

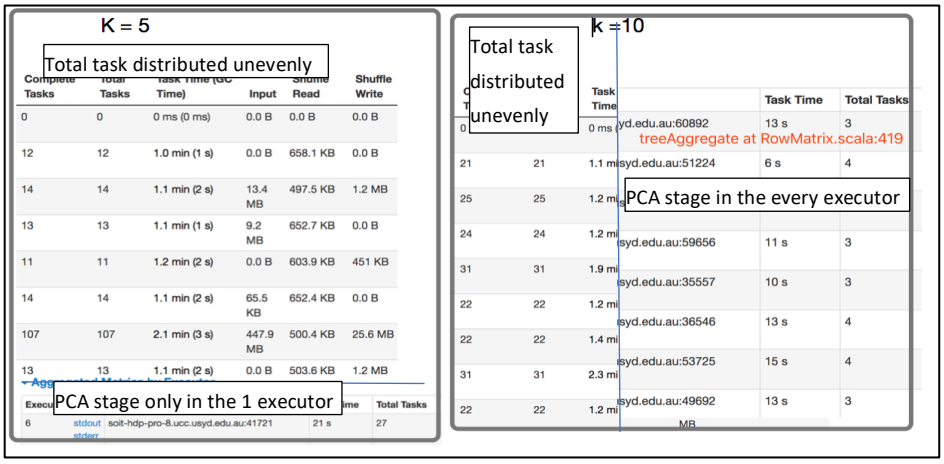

Figure 4. Total executor and PCA stage detail with different k in 50 dimensions

According to Figure 4, we found if "treeAggregate at RowMatrix.scala:419" and "treeAggregate at RowMatrix.scala:122" (PCA stage) tasks only on one executor, the total task distributed unevenly, but its shuffle read size is smaller. In the other hand, if PCA stage tasks distributed evenly, its shuffle read could be larger. Since the PCA is spark.ml module, its distribution could be random, and it couldn't be influence executor performance even though the shuffle read or distribution wouldn't large or unevenly. In addition, we found that if set one core with every executors, the PCA stage hasn't shuffle the detail.

- *Summary of k value and shuffle read*

In conclusion, the k value of KNN couldn't influence any execution time and executor performance, and the PCA stage could result in tasks distribution unevenly randomly so that lead to shuffle read (I/O cost) differently.

2.2.3 Reduced dimension of PCA (50, 100)

Based on fixed 8 executors and each executor has 4 cores with repartition test data in 64, we set the nearest neighbour number 5, then compare the dimension 50 and 100 after PCA. We summary their executors' performance as follow table:

Table 3. I/O cost with different dimension

PCA	50	100
Input	470.6MB	480.6MB
Shuffle Read	4.4MB	4.2MB
Shuffle Write	28.5MB	32.5MB

According to Table 3, the shuffle and I/O was not much different between 50 and 100 dimensions, and all executors were used fully according figure 2 and 3.

Then, we summarized the execution time into Figure 5:

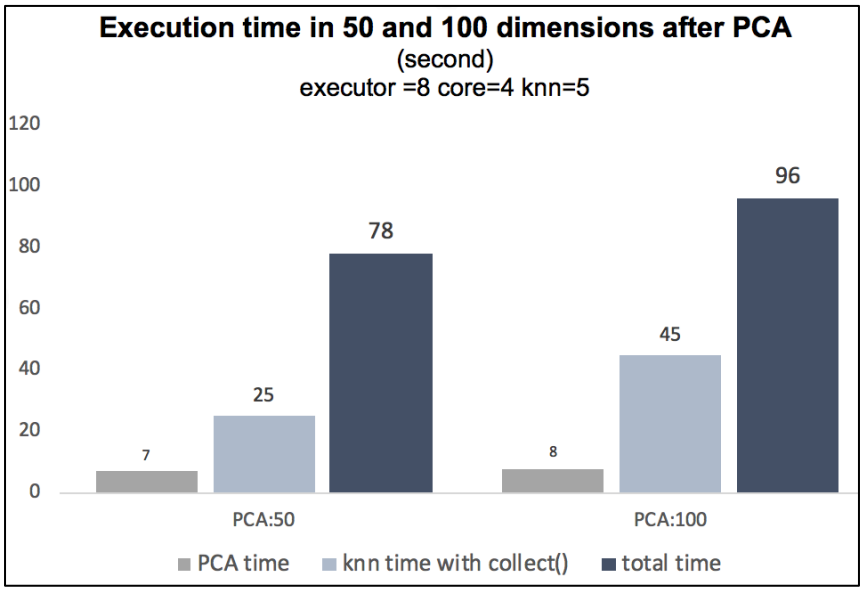

Figure 5. execution time in 50 and 100 dimensions after PCA

In this figure, we found that PCA spent time are not much difference, but the 50 dimensions was faster than 100 after PCA. In the 50 dimensions, knn stage spent 25 seconds, and total time was 78 seconds, but the 100 dimensions spent 45 and 96 seconds in each part.

According to these execution time, we think PCA stage didn't spend much time with more dimensions, but after PCA, more (100) features would spend more time to calculate distance, because it is double times than 50 features.

In conclusion, the number of dimension could influence execution time after PCA.

2.2.4 The number of executors and core

- *Comparing executors with the same total executor cores (8&2 vs 4&4)*

Based on fixed hyperparameters, we compared the program performance of the same amount of total executor cores which are 8 executors, 2 cores and 4 executors, 4 cores, the data is shown in Table 4:

Table 4. execution time and total I/O cost with difference combinator

PCA	K	executor	core	execution time		I/O cost (MB)		
				Time (mins)	KNN time(s)	input	Shuffle read	Shuffle write
50	5	4	4	1.3	31	452.6	3.1	19.6
50	10	4	4	1.4	34	448.4	3.1	19.6
100	5	4	4	2.2	66	476.9	5.1	23.6
100	10	4	4	1.7	47	448.4	4.7	23.6
50	5	8	2	1.5	33	466.9	9.4	19.6
50	10	8	2	1.6	32	468.4	3.6	19.6
100	5	8	2	2.1	56	477	7.1	23.6
100	10	8	2	2.5	53	477	12.9	23.6

Since PCA influence execution time, k value of KNN doesn't influence anything, according to this table, with the same 16 total executor cores, the "4&4" shuffle read size fluctuation was less than "8&2". The reason cloud be two parts that when PCA stage tasks distributed evenly and unevenly. If PCA tasks distributed evenly, less executors could lead to less exchanges, on the other hand, if PCA tasks distributed unevenly, executors exchanges less lead to shuffle read less that it has analysed above.

In conclusion, less executors could lead to less exchanges then result in less shuffle read (I/O cost). Therefore, we think less executors and more cores is the better choice when the total executor cores are same.

- *Comparing different executor (8&2 vs 8&4)*

Based on fixed hyperparameters, we compared in the same 8 executors different cores number performance, the data is shown in Table 5:

Table 5. execution time and total I/O cost with difference combinator

PCA	K	executor	core	execution time		I/O cost (MB)		
				Time (mins)	KNN Time(s)	input	shuffle read	shuffle write
50	5	8	4	1.3	25	470.6	3.6	28.5
50	10	8	4	1.2	19	450	15.2	28.5
50	5	8	2	1.5	33	466.9	9.4	19.6
50	10	8	2	1.6	32	468.4	3.6	19.6
100	5	8	4	1.6	45	450.2	19.9	32.5
100	10	8	4	1.5	33	442.2	19.6	32.5
100	5	8	2	2.1	56	477	7.1	23.6
100	10	8	2	2.5	53	477	12.9	23.6

The difference of shuffle write is distinct, we thought that, with the same executor, the shuffle write increased with cores increase. If cores increase, the number of written serialized data on all executors before transmitting would be more than less cores, because each core has many small size data, therefore more cores need write more quantity small size data in the broadcast, repartition and accumulator.

In conclusion, with same executor, different cores lead to different shuffle write, and shuffle write increased with cores increase.

- *Shuffle write performance*

Above has talked about shuffle write difference between cores, in this part focus on different combinator influence shuffle write, the Table 6 is combined by Table 4 and Table 5:

Table 6. execution time and total I/O cost with difference combinator

PCA	K	executor	core	execution time		I/O cost (MB)		
				Time (mins)	KNN Time(s)	input	shuffle read	shuffle write
50	5	8	4	1.3	25	470.6	3.6	28.5
100	5	8	4	1.6	45	450.2	19.9	32.5
50	5	8	2	1.5	33	466.9	9.4	19.6
100	5	8	2	2.1	56	477	7.1	23.6
50	5	4	4	1.3	31	452.6	3.1	19.6
100	5	4	4	2.2	66	476.9	5.1	23.6

Same total executor cores with same dimensions could cost same shuffle write, and shuffle write increased with total executor cores increase.

3 Spark Classifier Exploration

3.1 Random forest

3.1.1 Data format:

As the report mentioned above, Figure 6. Will show the steps of this stage. In Spark machine learning library, it provides the users wide range of classifier. In this stage, Random Forest and Multilayer Perceptron Classifier: effect of block size will be used. After processed by the classifier, it will return a Dataframe of "label, prediction", then we do the confusion matrix.

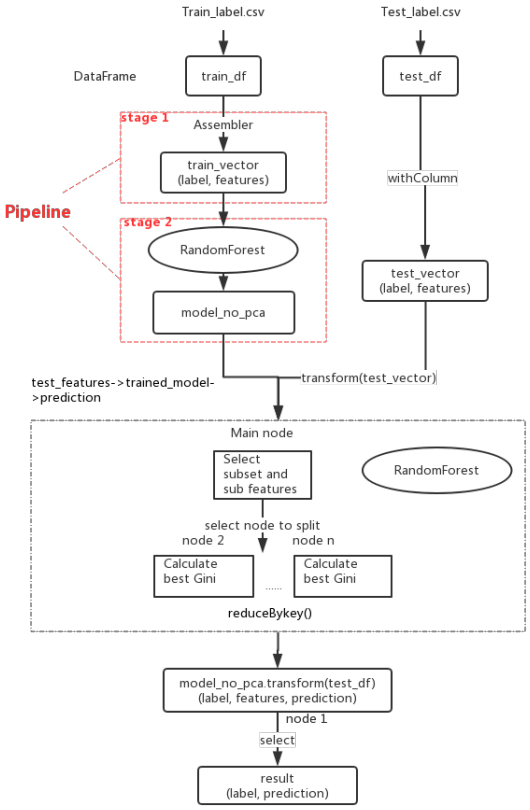

Figure 6. Random Forest Flow

3.1.2 Algorithm introduction:

Random Forest is a very powerful algorithm in Machine Learning area in recent years. It has great capability to predict the test data without any feature compression and it also has remarkable time performance. It can be used to classify both Regression data and Classification dataset.

The main concept of random forest is called bootstrap which means selecting subset z^* of size N from training data (a), then select m features from M features (b.1). Then using equation(1) equation to find the best split point (b.2). The total class is J, class and i∈{1,2,3……J}.

$$\text{Gini Inpurity: } I_{G(P)} = 1 - \sum_{i=1}^{J} {P_i}^2 \qquad (1)$$

Next, the algorithms will split the dataset z^* into two daughter sets (b.3).

Repeat b1-b3 until reach the pre-set tree depth (b), then a decision tree is formalised. Build the forest by repeating (a , b) steps B times, then the Random forest is $\{T\}_1^B$. Figure 7, gives a clear expression of the above steps.

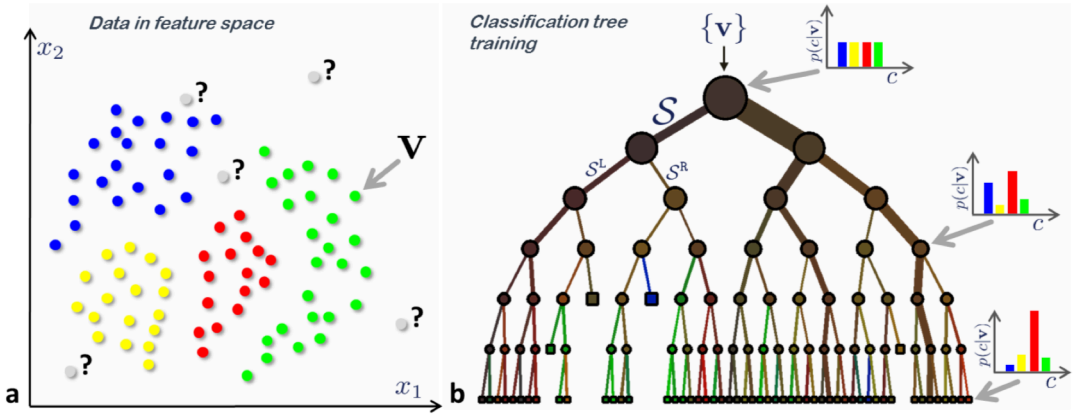

Figure 7. Example of Random Forest (Criminisi et al, 2011)

In order to make a prediction, Random Forest is capable for both Regression and Classification dataset. Furthermore, in classification, it take majority vote from the forest as its prediction. The equation will be given below. If the test point is x:

$$\text{Classification: } \hat{C}_{rf}^B(x) = majority\ vote\ \{\hat{C}_b(x)\}_1^B, \text{ where}$$

$$\hat{C}_b(x) \text{ represents the prediction from a single tree.} \quad (2)$$

3.1.3 Performance Evaluation:

- *Accuracy Comparison Versus Number of trees & Tree depths*

In this part, we use the same dataset but different parameters to test the relationship among these three.

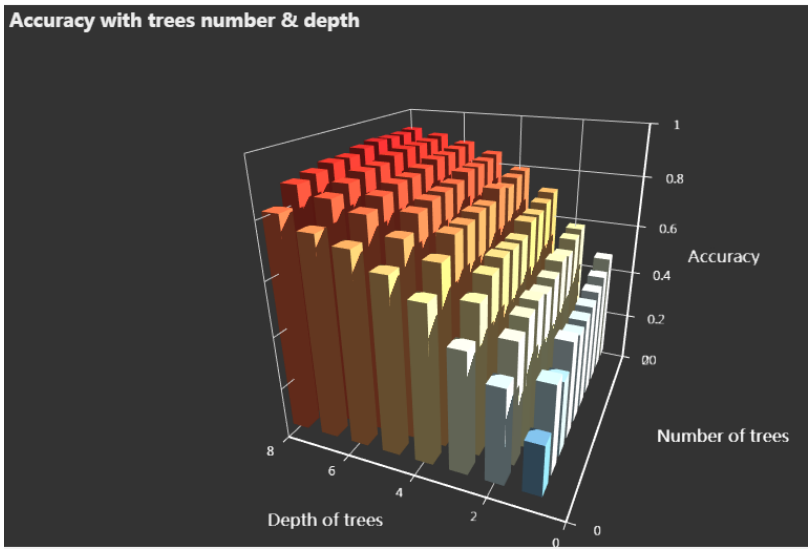

Figure 8. Accuracy Versus Nubmer of trees & Tree depths

The diagram has two parameters to influent the accuracy, which are depth of trees and number of trees. The depth of trees are set from 0 to 8. The number of trees are from 0 to 20. According to the diagram, it illustrates that the accuracy will increase while the depth of the trees and number of trees increase. The increment will slow down until the depth of trees is over 5 and the number of trees is over 16.

- *Time consumption Versus Number of trees & Tree depths*

Number of tree \ Tree Depth	1	2	3	4	5	6	7	8
2	13.0331	11.5708	11.1342	11.7728	11.2343	11.4321	11.4845	11.6007
4	10.5828	10.9374	10.9552	10.7554	10.9173	11.3001	11.6498	12.6115
6	10.5882	11.4599	13.0161	10.7376	10.8412	11.9015	13.0966	13.4886
8	10.938	10.8539	11.3162	11.4064	12.1461	13.0329	13.7942	15.2276
10	12.2548	11.6688	11.8569	12.4178	11.5308	12.4089	13.6785	16.8474

Figure 9. Time consumption Versus Number of trees & Tree depths in pseudo model

According to the algorithm, increasing tree depth and numbers will lead to more computation workload. In order to verify this, we run the code in pseudo executor. The table shows the relationship between depth of trees, number of trees and running time. Generally, it illustrates that the running time will increase since the depth of trees and number of trees increase. There are some fluctuates during the period, but major trend is positively correlated. Moreover, the slowest running time is around 10. The fastest running is 16 roughly. There is no large difference with running time when the tree depth is from 1 to 8 and number of trees is from 2 to 10 within the same dataset. So, as both directions of Number of trees and Tree Depth growing, running time will linearly increase. Since they are linearly related, the intuition is time should not have explosive growth.

However, when the code running on the real cluster with 4 executors and 4 cores, the result becomes different with what we expected before.

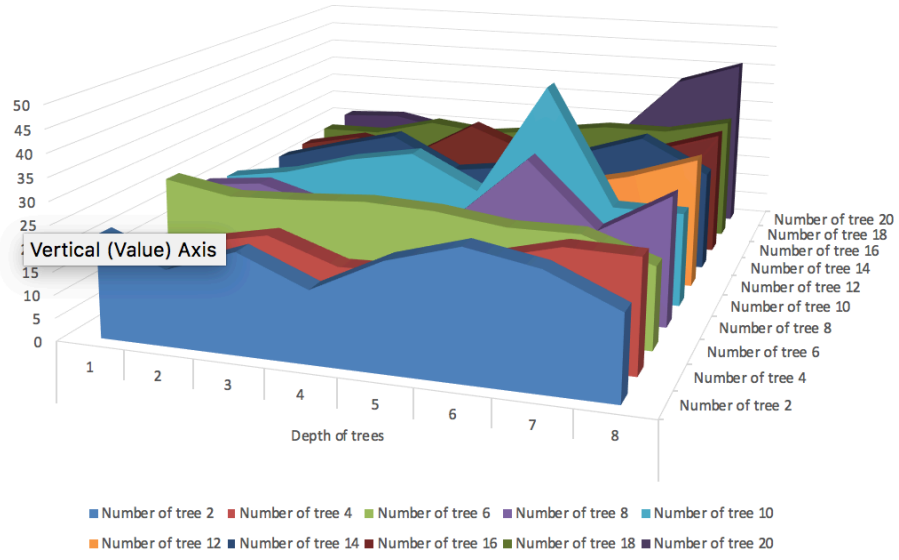

Figure 10. Time Verses Tree Depth & Number of Trees

As Figure 10 shown, when the depth going deeper, the running time will increase in most cases, but there are two dramatic increase, which is 10 trees with 6 depth and 8 trees with 6 depth. We will discuss them in the later section. Moreover, the running time with the same tree depth will perform fluctuation as the tree numbers growing, but they generally keep in the same level.

In conclusion, the tree depth will have bigger impact on running time than number of trees does in the real execution.

3.1.4 Execution statistics

We have set the tree number 5, 10 and 20 to test I/O cost, the table as follow:

table 7. execution statistics of random forest

Tree.num	Depth	Accuracy	Input (GB)	Shuffle Read (MB)	Shuffle Write (MB)	Time (mins)
5	6	0.8279	1.5	26.5	35.3	1.1
10	6	0.8312	1.6	38	50.6	1.2
20	6	0.8734	1.6	60.2	80.2	1.3

In this table, the tree number influence the shuffle read and write, the shuffle cost increased with tree number increase, but the total time influenced less.

3.1.5 Further Discussion:

Random Forest in Spark would be divided into flowing stages shown in Figure 11:

Completed Jobs (17)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
16	countByValue at MulticlassMetrics.scala:42	2018/05/30 09:30:03	1 s	2/2	10/10
15	collectAsMap at MulticlassMetrics.scala:48	2018/05/30 09:30:02	1 s	2/2	10/10
14	first at RandomForestClassifier.scala:140	2018/05/30 09:30:00	0.1 s	1/1	1/1
13	collectAsMap at RandomForest.scala:563	2018/05/30 09:29:58	3 s	2/2	32/32
12	collectAsMap at RandomForest.scala:563	2018/05/30 09:29:57	0.5 s	2/2	32/32
11	collectAsMap at RandomForest.scala:563	2018/05/30 09:29:57	0.5 s	2/2	32/32
10	collectAsMap at RandomForest.scala:563	2018/05/30 09:29:56	0.4 s	2/2	32/32
9	collectAsMap at RandomForest.scala:563	2018/05/30 09:29:55	0.6 s	2/2	32/32
8	collectAsMap at RandomForest.scala:563	2018/05/30 09:29:53	2 s	2/2	32/32
7	collectAsMap at RandomForest.scala:910	2018/05/30 09:29:49	4 s	2/2	32/32
6	count at DecisionTreeMetadata.scala:116	2018/05/30 09:29:45	4 s	1/1	16/16
5	take at DecisionTreeMetadata.scala:112	2018/05/30 09:29:44	1 s	1/1	1/1
4	take at Classifier.scala:111	2018/05/30 09:29:43	1 s	2/2	17/17
3	csv at NativeMethodAccessorImpl.java:0	2018/05/30 09:29:38	2 s	1/1	2/2
2	csv at NativeMethodAccessorImpl.java:0	2018/05/30 09:29:37	1 s	1/1	1/1
1	csv at NativeMethodAccessorImpl.java:0	2018/05/30 09:29:32	5 s	1/1	2/2
0	csv at NativeMethodAccessorImpl.java:0	2018/05/30 09:29:30	1 s	1/1	1/1

Figure 11. Jobs for 20 Trees 6 Depth

It is found that for each increasing tree depth, this algorithm will assign an extra "collectAsMap at RandomForest" job. By viewing the following chart, we found those job will be divided into map("mapPartition") and "collectAsMap" stages. All of them are fully distributed in the 4 executors and the input are similar. Which meaning Tree numbers may increase the input size and tree depth growth would cause more Jobs and input size. From the shuffle perspective, the shuffle read and write workload increasing as the stage ID increasing as Figure 12 shown.

Job ID	Description	Stage ID	Description	Time	Executor Number	Input	Shuffle read	Shuffle Write
		26	countbyvalue	31ms	1,2,3,4		1625B	
16	countByValue at MulticlassMetrics.scala:42	25	countbyvalue	1s	1,2,3,4	17.7MB		1625B
		24	collectasmap	24ms	1,2,3,4		1625B	
15	collectAsMap at MulticlassMetrics.scala:48	23	map	1s	1,2,3,4	17.7MB		1625B
14	first at RandomForestClassifier.scala:140	22	First at RFC	97ms	3	64KB		
		21	collectAsMap	2s	1,2,3,4		19.9MB	
13	collectAsMap at RandomForest.scala:563	20	mapPartitions	0.3s	1,2,3,4	189.9MB		19.9MB
		19	collectAsMap	0.2s	1,2,3,4		14.3MB	
12	collectAsMap at RandomForest.scala:563	18	mapPartitions	0.2s	1,2,3,4	189.9MB		14.3MB
		17	collectAsMap	0.2s	1,2,3,4		10MB	
11	collectAsMap at RandomForest.scala:563	16	mapPartitions	0.3s	1,2,3,4	189.9MB		10MB
		15	collectAsMap	0.1s	1,2,3,4		7.1MB	
10	collectAsMap at RandomForest.scala:563	14	mapPartitions	0.2s	1,2,3,4	189.9MB		7.1MB
		13	collectAsMap	0.1s	1,2,3,4		5.1MB	
9	collectAsMap at RandomForest.scala:563	12	mapPartitions	0.4s	1,2,3,4	189.9MB		5.1MB
		11	collectasMap	0.2s	1,2,3,4		3.5MB	
8	collectAsMap at RandomForest.scala:910	10	mapPartitions	2s	1,2,3,4	105MB		3.5MB
		9	collect AsMap	1s	1,2,3,4		16.7MB	
7	collectAsMap at RandomForest.scala:910	8	flat map	3s	1,2,3,4	105MB		16.7MB
6	count at DecisionTreeMetadata.scala:116	7	Coutn DT	4s	1,2,3,4	105MB		
5	take at DecisionTreeMetadata.scala:112	6	Decision Tree	1s	3	64kb		
4	take at Classifier.scala:111	5	Classifier	98ms	2		896B	
3	csv at NativeMethodAccessorImpl.java:0	3	csv	2s	3,4	17.5M		
2	csv at NativeMethodAccessorImpl.java:0	2	csv	1s	2	64k		
1	csv at NativeMethodAccessorImpl.java:0	1	csv	5s	1,4	104.6		
0	csv at NativeMethodAccessorImpl.java:0	0	csv	1s	1	64k		

Figure 12. Detailed information of stages and jobs in 20 Trees and 6 Depth

Back to the question in previous section, the running time of 10 trees with depth 6 is dramatically higher than the expected value. By observing it, the input size of 10 trees with 6 depth is slightly smaller than 20 trees with 6 depth, but the running time is much longer than the last one. In order to figure out the reason, our group compare the stages details between these two model. After analysing the result, stages from 7-10 in 10 trees with 6 depth has significant delay.

10	mapPartitions at RandomForest.scala:534	+details	2018/05/30 08:55:38	9 s	16/16	105.0 MB			1904.1 KB
9	collectAsMap at RandomForest.scala:910	+details	2018/05/30 08:55:35	3 s	16/16			16.7 MB	
8	flatMap at RandomForest.scala:903	+details	2018/05/30 08:55:21	14 s	16/16	81.7 MB			16.7 MB
7	count at DecisionTreeMetadata.scala:116	+details	2018/05/30 08:55:12	9 s	16/16	105.0 MB			

Figure 13. Delay stages in 10 Trees with 6 Depth

By checking each of them, we found the different reason for those delay. In stage 7(shown in Figure 14), all the works are assignment to executor 4 and they could be optimized by distributing on all executors.

In stage 8(shown in Figure 15), the problem is the late start of first 3 executors. All the 4 executor was start at the same time in 20 Trees with 6 Depth model.

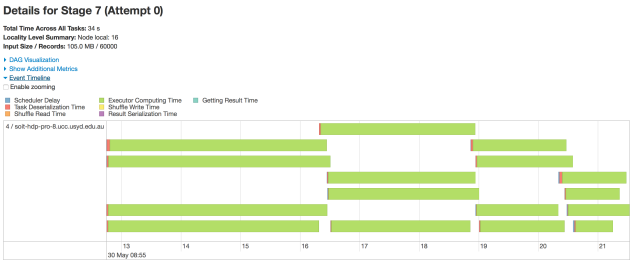

Figure 14. Stage 7 executor in 10 Trees with 6 Depth

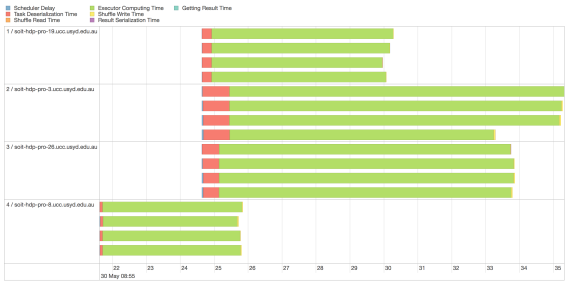

Fig 15. Stage 8 executor in 10 Trees with 6 Depth

In stage 10(shown in Figure 16), input size for both two model are 105MB. However, the spark only assign 1 executor for this task and it cause the delay very much.

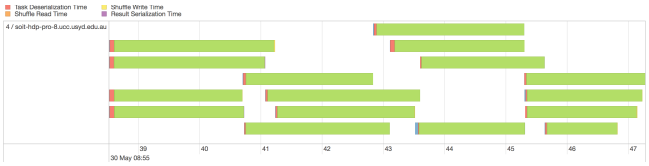

Figure 16. Stage 10 executor in 10 Trees with 6 Depth

In summary, when the classifier has same depth, their input would increase by the growth of tree numbers. However, when the system unable to fully assign the task to different executors or the executors do not start at same time, the running time will be longer than the expected time.

3.2 Multilayer Perceptron Classifier: effect of block size

3.2.1 Algorithm introduction:

Multilayer Perceptron is a kind of deep learning neural network, it has an input layer, multiple hidden layer and an output layer. Its structure is shown in figure 17.

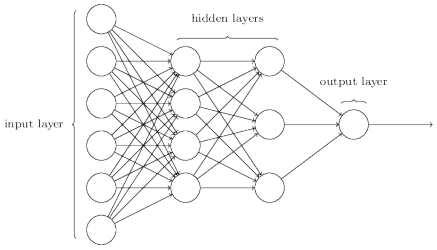

Fig 17. Multilayer Perceptron example structure (Ledell, 2017)

Input layer accept the input data, each node represents an input data. The number of neurons corresponding the number of features.

Hidden layers have no direct connection with the outside world, they map input data to output data linearly. The number of hidden layers should be at least one, for each input data with given weight *w* and bias *b* on certain nodes on K+1 layers, the output should be:

$$y(x) = f_K(\dots f_2(w_2^T f_1(w_1^T x + b_1) + b_2) \dots + b_K)$$

The sigmoid function is adopted by the nodes in hidden layer:

$$f(z_i) = \frac{1}{1 + e^{-z_i}}$$

Output layers corresponding to the classes. Nodes in this layer adopt softmax function:

$$f(z_i) = \frac{e^{z_i}}{\sum_{k=1}^{N} e^{z_k}}$$

3.2.2 Structure

The design structure is shown in figure 8. We combine assembler and mlp as a pipeline and construct the pipeline as a model to fit the training data frame. Then the fitted model transform the testing data frame and generate the prediction result.

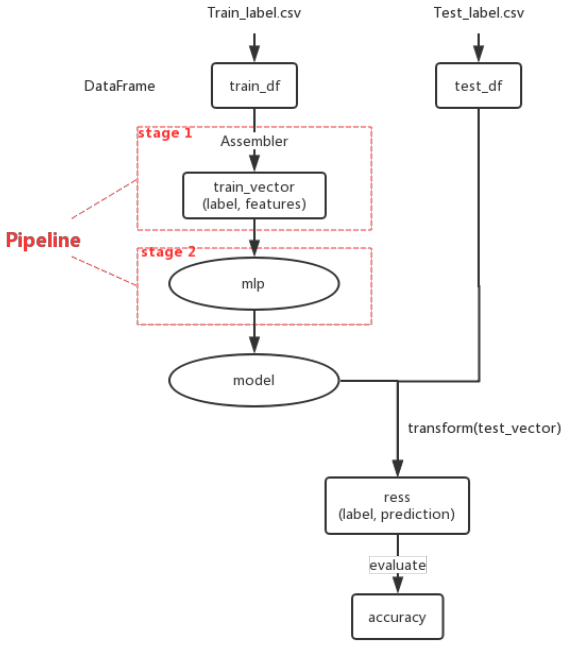

Figure 18. Design Structure of Multiple Later Perceptron Classifier

3.2.3 Accuracy analysis

We use the MultilayerPerceptronClassifier to make the prediction, we set fitted hidden layers number (784, 50, 10), iteration number(10) with 2 executors and 2 executor-cores and get accuracies with slight difference by adjusting the block size only. We set block size is 64, 512 and 2048, the accuracy as figure 9:

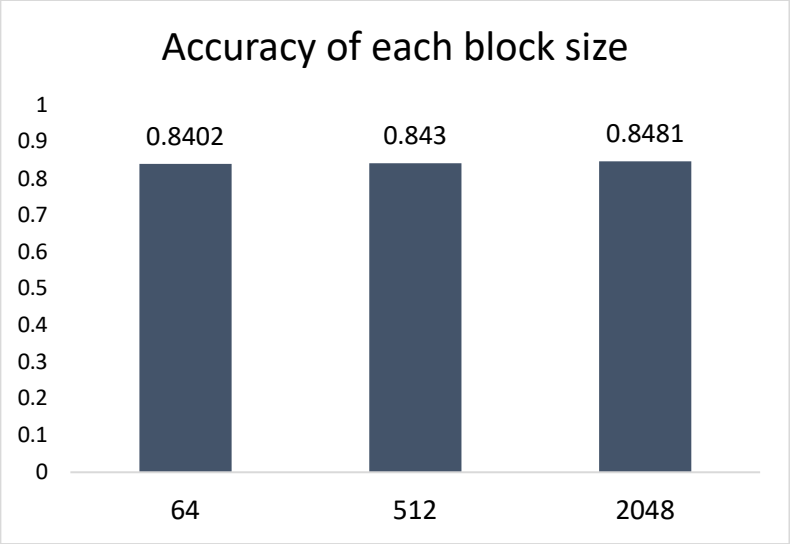

Figure 19. Accuracy of block size (64/512/2048)

According to this figure, the accuracy doesn't show a distinct difference, so if the block size in a reasonable range, it could not influence accuracy. But if the size is too small, noise data would influence training stage, besides, if the size is too large, less parts to iterate in every epoch so that the accuracy could reduce quickly.

3.2.4 Execution statistics

Execution time with different block size figure as follow:

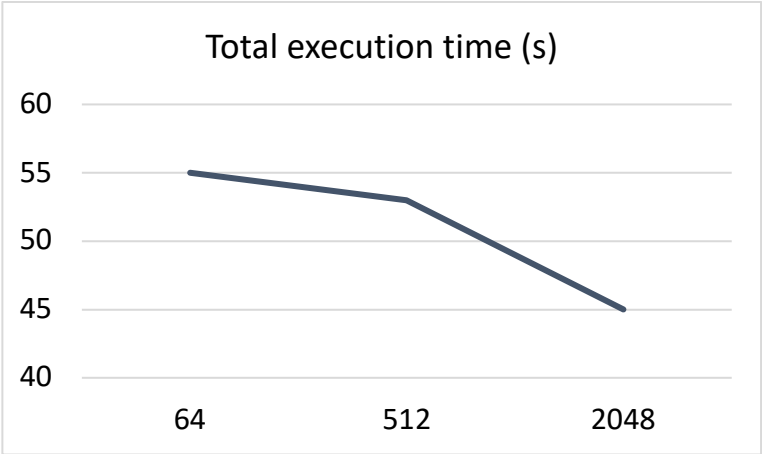

Figure 20. Total execution time of block size (64/512/2048)

In this figure, the total execution time could be decreased with block size increase. Because larger block size would split data less parts, and with one epoch less parts would be using less time to training data and predict data.

The executor total I/O cost table as follow:

table 8. different block size executor I/O cost

block size	input	shuffle read	shuffle write
64	4.5GB	0.0B	1KB
512	4.5GB	510B	1KB
2048	4.5 GB	516B	1KB

The shuffle cost is near zero, however input volume is not only same, but also large that more data read from Hadoop or Spark storage.

In conclusion, block size can reduce total execution time and doesn't influence I/O cost.

3.3 Comparison random forest and multilayer perception

According to the table 7 and 8, compared random forest and multilayer perception, with the similarly accuracy, the multilayer perception not only shuffle write and read, but also execution time and total executor cores number are all are lower than random forest.

In addition, when the executors more than 4, after our many tests, the MLP implementation executors' task would distributed unevenly, and executors' task time were distinctly different, the executors spark UI as figure 21.

sk ed	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Time (GC Time)	Input	Shuffle Read	Shuffle Write
B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
B	4	0	0	6	6	3 s (52 ms)	0.0 B	0.0 B	0.0 B
B	4	0	0	7	7	2 s (0.1 s)	0.0 B	0.0 B	0.0 B
B	4	0	0	415	415	1.8 min (4 s)	1.6 GB	0.0 B	41.1 M
B	4	0	0	7	7	3 s (0.1 s)	0.0 B	0.0 B	0.0 B
B	4	0	0	8	8	4 s (84 ms)	0.0 B	0.0 B	0.0 B
B	4	0	0	6	6	2 s (34 ms)	0.0 B	0.0 B	0.0 B
B	4	0	0	6	6	3 s (34 ms)	0.0 B	0.0 B	0.0 B
B	4	0	0	6	6	3 s (0.2 s)	0.0 B	0.0 B	0.0 B

Figure 21. once test MLP with 8 executors and 4 cores

In conclusion, If the dataset is relatively small or is images set, and users want to set executors less than four, the spark_MLP could be a better choice, but if dataset is large, implementation of random forest can be distributed evenly.

Reference

[1] A. Criminisi, J. Shotton, and E. Konukoglu, "Decision Forests: A Unified Framework for Classification, Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning", *Foundations and Trends® in Computer Graphics and Vision*, vol. 7, no. 2-3, pp. 81-227, 2012.

[2] E. Ledell, "ledell/sldm4-h2o", *GitHub*, 2018. [Online]. Available: https://github.com/ledell/sldm4-h2o/blob/master/sldm4-deeplearning-h2o.Rmd. [Accessed: 30- May- 2018].

[3] "Classification and regression - Spark 2.3.0 Documentation", *Spark.apache.org*, 2018. [Online]. Available: https://spark.apache.org/docs/latest/ml-classification-regression.html#multilayer-perceptron-classifier. [Accessed: 30- May- 2018].