THE UNIVERSITY OF SYDNEY

COMP5349 Cloud computing Assignment2:

Spark Machine Learning Application

Group name: SIT_115_7

Yuanqi Pang 460490911
Penghao Li 470075025
Haoran Liu 470162749

1. Stage 1: KNN design

In this section, all the algorithm design procedure and consideration will be introduced. The whole algorithm design is based on pyspark dataframe, extensive approaches have been tried for choosing the best method for solution. Both RDD solution and dataframe solution is tried and dataframe showed better performance.

Initially, the group wanted to come up with a solution that do not import any python libs except the Pyspark module. So the first approach was done by cross join 2 data file and group by test data id to group up all the training features. This beta version is executed in Jupyter notebook locally, the test on a small set of data is successful. However, the memory crashed at the group by period when testing with the whole data set.

The group had to give up the crossjoin and search for python lib approach. As the distance calculation is based on matrix calculation, it is a good choice to import numpy module for matrix calculation. The KNN procedure can be simply decomposed into 3 part: calculate the distance, find the closest k number of distances, select the majority label among k. These 3 part can all be easily approached by using numpy. The following snippet shows the quickest approach of KNN after comparison of various functions. Using argpartition method means the distances list is not actually sorted, instead, it just return the index of the nearest points.

```
dists = np.sum((trainfeature.value-np.array(test_feature))**2,axis=1)**0.5
sortedlabels = np.take(trainlabel.value, np.argpartition(dists,num_k)[:num_k])
p = int(np.bincount(sortedlabels).argmax())
```

Figure 1. KNN solution

After finding the quickest approach to KNN solution, the work focus on optimize the design pattern and parallelize algorithm and finally confirm the design pattern of the stage 1.

As shown in figure 2, after the initialize of spark session and parameters and read file from HDFS. The test data was repartitioned to maximum the parallelize, the number of partition is set to be number of executors * executor cores to ensure that each execute-core are assigned at least 1 task. A persist method is used after the repartition to set the storage to memory only. As the default storage level for Spark is memory and disk and the reduced dimension data is small enough to be stored in each memory. It is better to store data on memory for executing efficiency.

Pipeline is used to integrate the procedure of assemble the vectors and PCA dimension reduction, besides, it is reusable by train data and test data, after fit the pipeline to train data, the test data and train data went through the pipeline and the dataframe of reduced dimension and label was extracted.

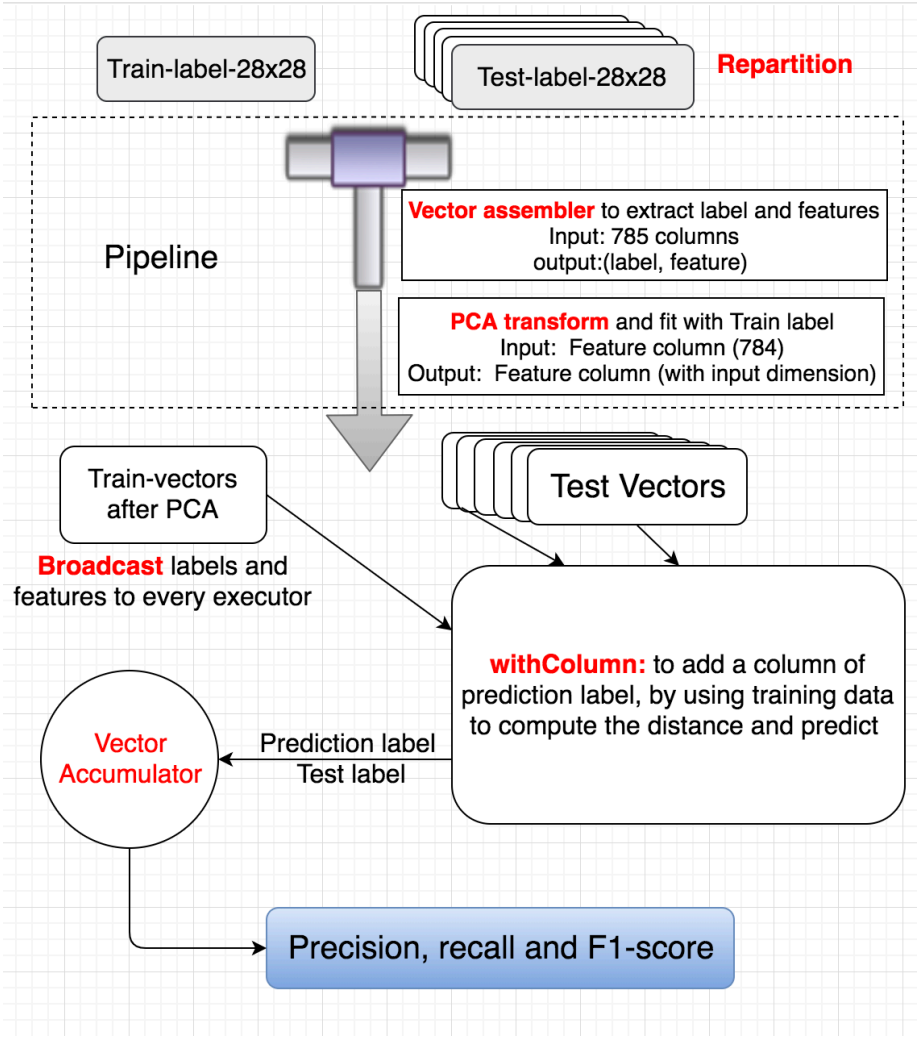


Figure 2. Design pattern of stage 1

The training data was not partitioned as the training data need to be as a whole and be used by all executors in distance calculation. A broadcast method is used to send the training label and features to all executors. Though spark can have a

shared memory, it is better to have the training data broadcasted to all executors and cached on each. The running time on cluster prominently reduced after using the broadcast.

A withColumn method is used for the test dataframe to generate a column of prediction label. Before the withColumn section, a vector accumulator is set up to collect all the predict label and test label and generate a confusion matrix of the prediction. At the withColumn period, the accumulator will have collected all the prediction in the confusion matrix. Since Spark has lazy evaluation effect, a collect method is used right after the withColumn to force evaluation. After the KNN prediction the broadcast item is unpersisted to release the memory.

```
knnpredictions = test_vectors.withColumn("predict_label",
    knn_udf(test_vectors["pcafeatures"],test_vectors["test_label"])).collect()
```

As shown in code snippet, this part cost the most time in execution, as the test vectors are divided into parts and send to every executor cores, this stage can be executed totally in parallel and all tasks separate uniformly on every executor cores. The output confusion matrix is used to calculate precision, recall and F1-score, the prediction evaluation was converted to dataframe and show in tabular form in the terminal. The pair-wise prediction and real test label is saved as txt file and stored in HDFS. Figure 3 shows the result sample of this project.

```
confusion matrix for prediction:
[[ 973    1    1    0    0    1    3    1    0    0]
 [   0 1130    2    0    0    0    2    0    0    1]
 [   8    2 1001    1    2    0    1   14    3    0]
 [   0    1    5  974    1   13    0    7    5    4]
 [   1    5    0    0  950    0    4    1    1   20]
 [   4    0    0    9    1  866    5    1    3    3]
 [   3    3    0    0    3    2  947    0    0    0]
 [   0   19    5    0    2    0    0  996    0    6]
 [   6    0    5   12    6   10    4    4  923    4]
 [   5    6    3    7    5    3    1    9    3  967]]
+-----+---------+------+--------+
|label|precision|recall|F1-score|
+-----+---------+------+--------+
|  0.0|    0.973| 0.993|   0.983|
|  1.0|    0.968| 0.996|   0.982|
|  2.0|    0.979|  0.97|   0.974|
|  3.0|    0.971| 0.964|   0.967|
|  4.0|    0.979| 0.967|   0.973|
|  5.0|    0.968| 0.971|   0.969|
|  6.0|    0.979| 0.989|   0.984|
|  7.0|    0.964| 0.969|   0.966|
|  8.0|    0.984| 0.948|   0.966|
|  9.0|    0.962| 0.958|    0.96|
+-----+---------+------+--------+
```

```
prediction: 7  test label: 7
prediction: 2  test label: 2
prediction: 1  test label: 1
prediction: 0  test label: 0
prediction: 4  test label: 4
prediction: 1  test label: 1
prediction: 4  test label: 4
prediction: 9  test label: 9
prediction: 5  test label: 5
prediction: 9  test label: 9
prediction: 0  test label: 0
prediction: 6  test label: 6
prediction: 9  test label: 9
prediction: 0  test label: 0
prediction: 1  test label: 1
```

Figure 3. result sample

2. Stage 2: performance evaluation

The test environment is the soit hdp cluster, with 30 nodes and 4 cores on each node. Local test is on macbook with 4 core processor.

2.1 Prediction evaluation

Extensive testing come to the conclusion that the change of number of executors and executor cores will never change the prediction, the parallelism do not affect the prediction result. Thus the comparison of prediction result is among different number of reduced dimension and number of nearest neighbors. PCA and K number is designed to be within 50-100 and 5-10, 50 and 100 is selected for PCA number for better comparison, number of nearest neighbors is set to be 5 and 9, odd number is set for K in case number of nearest neighbors of different label are the same.

The figure 4 shows the confusion matrix and precision, recall and F1-score under different circumstances. The overall accuracy is beyond 97%, which is extremely high compare to other machine learning algorithm. However, it is weird that PCA 100 had even worse performance than PCA 50. When predicting with higher data integrity, the result is ought to be more accurate. Another testing of PCA 100 and k equals to 3 is done for further comparison. The further result tended to be better than K equals 5 or 9. It is indicated that when executing with higher dimension. The data integrity is better. Thus the nearest point are more likely to be with the same label and increasing K value may increase the number of disruption.

<table>
<tr><th>K \ PCA</th><th>50</th><th>100</th></tr>
<tr><td>5</td><td>confusion matrix for prediction:
[[972 1 1 0 0 1 4 1 0 0]
[0 1129 3 0 0 0 2 0 0 1]
[7 2 1004 1 2 0 0 13 3 0]
[0 0 3 975 1 11 0 7 9 4]
[2 2 0 0 960 0 4 1 0 13]
[3 0 0 10 1 866 6 1 3 2]
[3 4 0 0 2 2 947 0 0 0]
[0 18 6 1 3 0 0 992 0 8]
[4 0 3 13 4 6 3 3 935 3]
[5 5 3 6 7 4 1 6 4 968]]
+-----+---------+------+--------+
|label|precision|recall|F1-score|
+-----+---------+------+--------+
| 0.0| 0.976| 0.992| 0.984|
1.0	0.972	0.995	0.983
2.0	0.981	0.973	0.977
3.0	0.969	0.965	0.967
4.0	0.98	0.978	0.979
5.0	0.973	0.971	0.972
6.0	0.979	0.989	0.984
7.0	0.969	0.965	0.967
8.0	0.98	0.96	0.97
9.0| 0.969| 0.959| 0.964| +-----+---------+------+--------+</td><td>confusion matrix for prediction: [[973 1 1 0 0 1 3 1 0 0] [0 1130 2 0 0 0 2 0 0 1] [8 2 1001 1 2 0 1 14 3 0] [0 1 5 974 1 13 0 7 5 4] [1 5 0 0 950 0 4 1 1 20] [4 0 0 9 1 866 5 1 3 3] [3 3 0 0 3 2 947 0 0 0] [0 19 5 0 2 0 0 996 0 6] [6 0 5 12 6 10 4 4 923 4] [5 6 3 7 5 3 1 9 3 967]] +-----+---------+------+--------+ |label|precision|recall|F1-score| +-----+---------+------+--------+ | 0.0| 0.973| 0.993| 0.983
1.0	0.968	0.996	0.982
2.0	0.979	0.97	0.974
3.0	0.971	0.964	0.967
4.0	0.979	0.967	0.973
5.0	0.968	0.971	0.969
6.0	0.979	0.989	0.984
7.0	0.964	0.969	0.966
8.0	0.984	0.948	0.966
9.0	0.962	0.958	0.96	 +-----+---------+------+--------+</td></tr>
<tr><td>9</td><td>confusion matrix for prediction: [[971 1 1 0 0 1 5 1 0 0] [0 1131 2 0 0 0 1 0 0 1] [8 1 1002 0 1 0 2 14 4 0] [0 0 1 978 1 12 0 6 7 5] [0 5 0 0 956 0 5 0 1 15] [3 0 0 7 2 869 7 1 1 2] [3 4 0 0 2 1 948 0 0 0] [0 17 4 0 2 0 0 991 0 14] [4 0 4 11 4 7 2 3 935 4] [5 5 2 5 9 4 1 5 2 971]] +-----+---------+------+--------+ |label|precision|recall|F1-score| +-----+---------+------+--------+ | 0.0| 0.977| 0.991| 0.984|
1.0	0.972	0.996	0.984
2.0	0.986	0.971	0.978
3.0	0.977	0.968	0.972
4.0	0.979	0.974	0.976
5.0	0.972	0.974	0.973
6.0	0.976	0.99	0.983
7.0	0.971	0.964	0.967
8.0	0.984	0.96	0.972
9.0| 0.959| 0.962| 0.96| +-----+---------+------+--------+</td><td>confusion matrix for prediction: [[973 1 1 0 0 1 3 1 0 0] [0 1131 2 0 0 0 1 0 0 1] [10 4 994 2 1 0 2 14 5 0] [0 2 1 978 1 12 0 6 5 5] [1 7 0 0 949 0 5 1 1 18] [4 0 0 6 2 868 7 1 1 3] [6 4 0 0 3 2 943 0 0 0] [0 22 4 0 3 0 0 989 0 10] [5 0 5 12 6 6 2 6 929 3] [4 6 2 7 6 4 1 8 3 968]] +-----+---------+------+--------+ |label|precision|recall|F1-score| +-----+---------+------+--------+ | 0.0| 0.97| 0.993| 0.981
1.0	0.961	0.996	0.978
2.0	0.985	0.963	0.974
3.0	0.973	0.968	0.97
4.0	0.977	0.966	0.971
5.0	0.972	0.973	0.972
6.0	0.978	0.984	0.981
7.0	0.964	0.962	0.963
8.0	0.984	0.954	0.969
9.0	0.96	0.959	0.959	 +-----+---------+------+--------+</td></tr>
</table>

Figure 4. Confusion matrix and precision, recall and F1-score for different k and n

When PCA is reduced to lower dimensions, increase K value will increase number of points with correct label, hence increasing the accuracy.

2.2 Execution time evaluation

The time evaluating is set to be comparison between 2 K value, 2 PCA value and 3 executor are core pairs, total number of executer-cores are set to be 8, 8, 16. Executing with 2 executors, 4 cores and 4 executors, 2 cores have the same total cores, while the performance is not the same, when executing with PCA value equals 50, the performance appears similar, when PCA increase, the computational

load increase for calculating Euclidean distance. Figure 5 shows that 4 executor with 2 cores execute faster than 2 executor with 4 cores which is quite weird as more cores on 1 executor should reduce the shuffle read cost. This situation may be because of the execution time on cluster is not stable, and using 2 executor may more likely to be allocated with an overloaded executor which may increase the execution time.

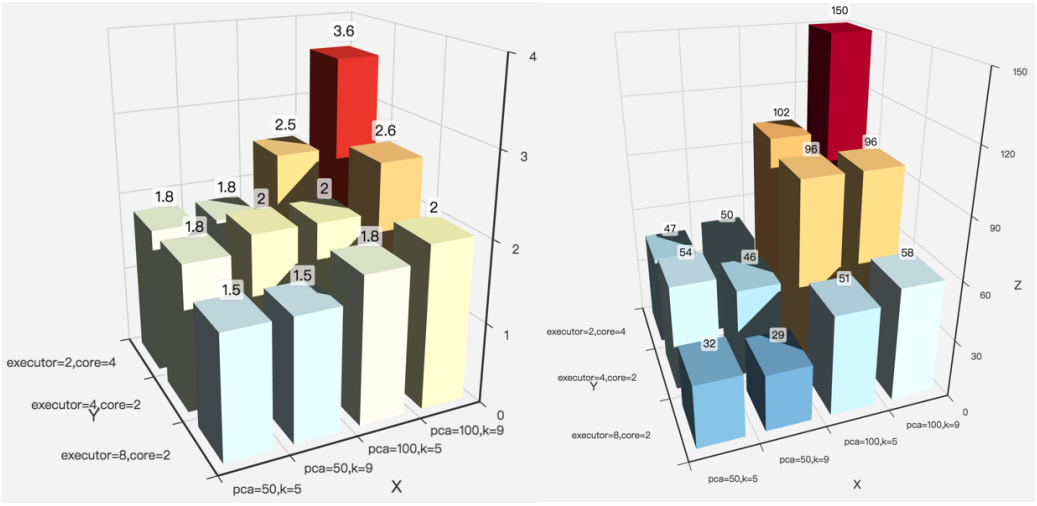


Figure 5. Total execution time(min) and KNN stage execution time(s)

As the execution time on cluster including the queueing time, the comparison of execution time of KNN stage is more obvious. The KNN stage execution time reduce rapidly when increasing the number of total executor-cores. In conclusion, the program is able to execute faster with more cores as the parallelism can handle over 8 executors and 2 cores.

2.3 Parallelism evaluation

As mentioned in the algorithm design, a repartition method is used after the csv was read to the dataframe, thus all actions performed on test vector is in parallel. It is indicated in figure 6 that all cores are assigned same amount of tasks and execution time is nearly the same in the KNN prediction stage. This stage occupies half of the execution time, which is the most important period for parallelism.

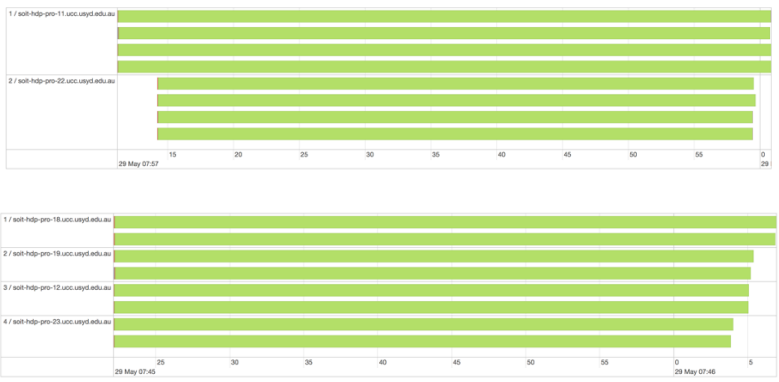


Figure 6. KNN stage execution parallelism with 2,4,8 executors

Besides, the overall parallelism had better result on fewer executors, when executor number is 2 or 4 all tasks will always be uniformly allocated to all executors. While the executor number is set to be 8, the most time-consuming actions will run in parallel, and other stages will be randomly in parallel. This may because the dataset of this task is quite small, and the parallelism of other stages is controlled by Spark automatically, these stage tasks are assigned by Spark and executed by 1 executor if the task is light task.

2.4 IO Cost

Number of nearest neighbors has no effect on shuffle read or write. Thus it is unnecessary to include K in the comparison. PCA will affect size of broadcast value. So 3 type of executor and core pairs as well as PCA is selected for I/O comparison.

	2 executors 4 cores		4 executors 2 cores		8 executors 2 cores	
PCA	50	100	50	100	50	100
Shuffle Read	2.6MB	2.6MB	8.1MB	8MB	12.2MB	4.5MB
Shuffle Write	13.5MB	13.5MB	13.5MB	13.5MB	20.7MB	20.7MB
Input	476.9MB	424MB	476.9MB	448.3MB	448.3MB	477MB

Figure 7. shuffle size comparison

It is indicated in Figure 7 the PCA value will not affect the shuffle read, and when there is fewer number of executors the shuffle read decreased. It is easy to explain that fewer cores will have fewer blocks of data, an executor will hold more data thus less data transfer between executor cores. The shuffle write size is linear to total executor cores, more cores will need data from the executor read the csv file. As mentioned above the tasks may not separate uniformly on every executor when the number of executor is set to 8. Another interesting finding is that when the tasks do not uniformly separate, the shuffle read size will be extremely small. The Figure 7 shows that when executing with 8 executors and 2 cores with PCA 100, there is only 4.5 MB shuffle read. In conclusion, it is not always a bad thing that parallelism did not reaching the max, it will reduce the shuffle size as some short period will be executed by a single executor. Thus the best situation is to have a perfect balance between shuffle read size and parallelism.

3. Stage 3: additional algorithm

3.1 Logistic Regression

Introduction

Logistic regression is a statistical model widely applied to binary dependent variable. In the SPARK classification library both binomial and multinomial Logistic Regression API can be selected to use. The basic theory of Logistic Regression is to find the cost function and reduce the losses by the gradient decent method. To optimize this algorithm there could be a series of relative methods such as regularization and dataset pre-processing. The activate will select sigmoid function which can prevent the value flowing out the range [0, 1]. Besides, the label can be judge binarily by defining whether the value of y is above 0.5. To make the whole method multinomial, researchers usually apply the 'one-vs-all' method. In addition, it can lead the function curve to be convex for gradient decent. The cost function is a main part of Logistic Regression. It can indicate the loss between the predicted values on the fitting curve and the real values which the test data holds. The optimization process of Logistic Regression is to process gradient decent to this function. The cost function:

$$\text{Cost}(h_\theta(x), y) = -ylog\left(h_\theta\left(x^{(i)}\right)\right) - (1-y)\log\left(1 - h_\theta(x)\right)$$

To reduce the losses and improve the accuracy, the gradient decent method should be. Refer to the official API document of Logistic Regression on SPARK website, the default parameter set is (maxIter =100, regParam=0.1, elasticNetParam=1.0)

```
(regParam=0.1, elasticNetParam=1.0, family="multinomial")
```

Accuracy evaluation

In this experiment, some other parameters were tested to try to improve the accuracy. However, most of the results are even worse. When the bigger max iteration was applied (e.g. 150) the accuracy obviously dropped; when the smaller max iteration was applied, some of them can only very slightly improve the accuracy. The optimal situation is maxIter = 80, which can only improve the accuracy of no PCA version by 0.01% (from 92.69% to 92.70%). Considered about this the max iteration number kept default. The accuracy is shown in Figure 8: dimension 50 is 0.9119, dimension 100 is 0.9219 and raw data is 0.9269.

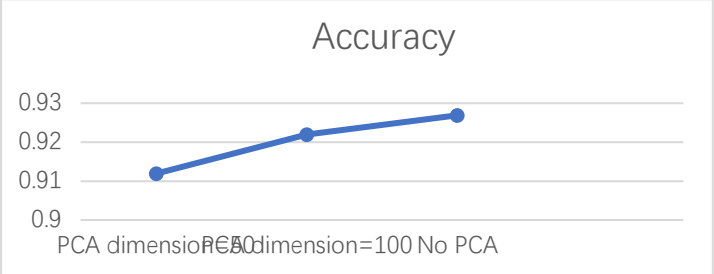
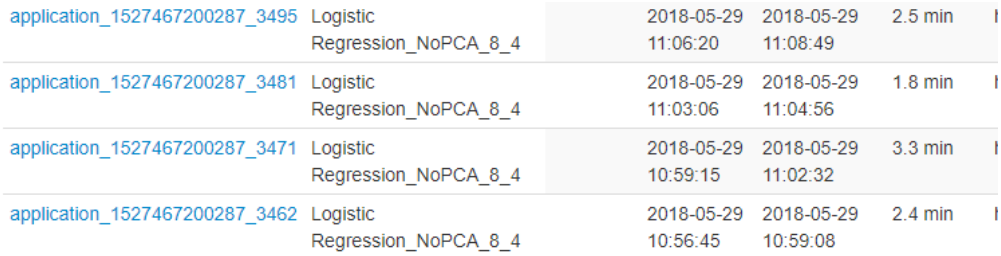


Figure 8 The accuracy of each dimension

How to prepare& collect

For the preparation of data, the requirement of Logistic Regression is selecting 3 different dimensions of PCA. Raw data, 100 dimensions and 50 were selected. This is

the main part of data preparation. Meanwhile, each dimension was executed under 3 different situations: 2 executors and 4 cores, 4 executors and 2 cores, and 2 executors and 4 cores to discover the difference between the running time lengths. At last, this program will output the accuracy of the model. Two programs including one PCA version and one no PCA version were developed. For users, to apply the different dimension numbers on the PCA version, the number can be input in the command line. According to the cluster is very crowded by plenty of students running programs at the same time, in many cases a program will wait in a queue to run rather than immediately. Even running the same program, the duration (including waiting time) will be totally different. (as in Figure 9) In this case, this report will calculate the time duration in a different method.



application_1527467200287_3495	Logistic Regression_NoPCA_8_4		2018-05-29 11:06:20	2018-05-29 11:08:49	2.5 min
application_1527467200287_3481	Logistic Regression_NoPCA_8_4		2018-05-29 11:03:06	2018-05-29 11:04:56	1.8 min
application_1527467200287_3471	Logistic Regression_NoPCA_8_4		2018-05-29 10:59:15	2018-05-29 11:02:32	3.3 min
application_1527467200287_3462	Logistic Regression_NoPCA_8_4		2018-05-29 10:56:45	2018-05-29 10:59:08	2.4 min

Figure 9 The different duration of the same program

The method is to observe the event time line in the job view. Comparing with different executions of a same program, no matter how long the waiting time is, the periods between the submission time of the first job and the complete time of the last job are almost same (still having some small deviations due to the unstable performance of the busy CPUs in the cluster).

After running the program in different para-meters the running time table was created.

Settings / Dimensions	2 executors, 4 cores	4 executors, 2 cores	8 executors, 2 cores
PCA-Dimension 50	48s	95s	59s
PCA-Dimension 100	59s	97s	73s
Raw	46s	44s	39s

Figure 10. Running time of different settings

Execution statistics& Interesting Findings

In Figure 10 it clearly shows this program ran parallelized in the gradient decent stage, which distributes many little processes to all executors. For other settings the structure diagrams are almost same except the executor numbers. Meanwhile, in the executor tab in the SPARK history website it also displays the tasks each executor processed. (Figure 11). In Table 1 it displays that without PCA process the running time can be shorter. Comparing the 50 and 100 dimensions version, small dimension of PCA will save time but the accuracy will drop. The reason is that the smaller dimension data has smaller complexity, but the loss of the original data is bigger. However, the 8 executors &2 cores version only has the best performance in the no PCA version. When applying PCA, the 2 executors and 4 cores version is faster. This is a little strange. The most probable reason is this code combined assemble, PCA and Logistic Regression model as a pipeline. Only assemble and Logistic Regression run parallelized. PCA does

not corporate with the Logistic Regression model very well.

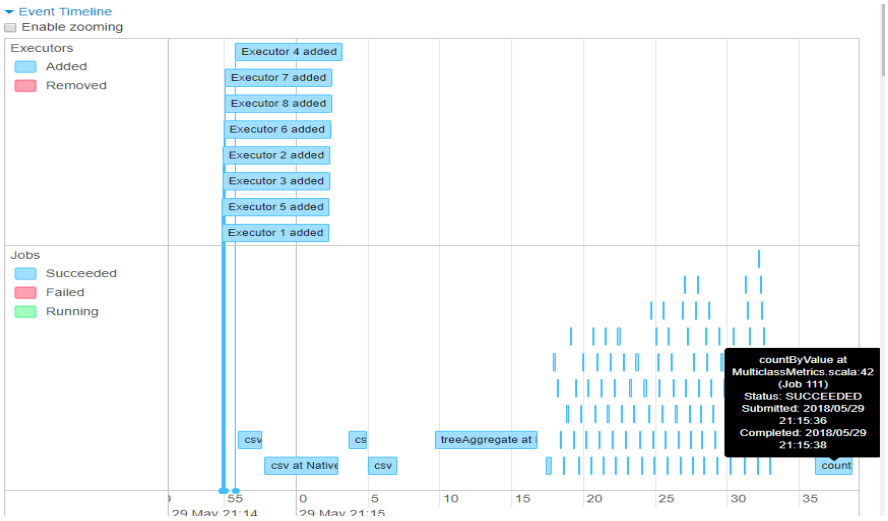


Figure 11. The Event time line for 8 executors 2 cores

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	(GC Time)	Input	Shuffle Read
driver	172.16.176.95:42429	Active	0	0.0 B / 384.1 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B
1	soit-hdp-pro-4.ucc.usyd.edu.au:52807	Active	0	0.0 B / 956.6 MB	0.0 B	2	0	0	274	274	43 s (1.0 s)	439 MB	756 KB
2	soit-hdp-pro-9.ucc.usyd.edu.au:52634	Active	0	0.0 B / 956.6 MB	0.0 B	2	0	0	275	275	31 s (0.5 s)	475.5 MB	567.3 KB
3	soit-hdp-pro-12.ucc.usyd.edu.au:58000	Active	0	0.0 B / 956.6 MB	0.0 B	2	0	0	260	260	30 s (0.6 s)	406.4 MB	2.3 MB
4	soit-hdp-pro-4.ucc.usyd.edu.au:52290	Active	0	0.0 B / 956.6 MB	0.0 B	2	0	0	274	274	44 s (0.6 s)	358.3 MB	972.7 KB
5	soit-hdp-pro-19.ucc.usyd.edu.au:51958	Active	0	0.0 B / 956.6 MB	0.0 B	2	0	0	276	276	26 s (0.5 s)	425.7 MB	2.4 MB
6	soit-hdp-pro-16.ucc.usyd.edu.au:54592	Active	0	0.0 B / 956.6 MB	0.0 B	2	0	0	287	287	28 s (0.5 s)	420.1 MB	2.6 MB
7	soit-hdp-pro-20.ucc.usyd.edu.au:57819	Active	0	0.0 B / 956.6 MB	0.0 B	2	0	0	272	272	29 s (0.6 s)	434.7 MB	573.4 KB
8	soit-hdp-pro-14.ucc.usyd.edu.au:53429	Active	0	0.0 B / 956.6 MB	0.0 B	2	0	0	270	270	29 s (0.7 s)	406.4 MB	2.4 MB

Figure 12. Executor tasks

Multilayer Perceptron Classifier

There will be three main parts which is related to the multilayer perceptron classifier discussed in this section. The first part is the introduction of this classifier applied in this case. The second part will mention how to prepare data for the classifier and how to collect the output. The other part is the analysis of the performance of this classifier in the prediction process and some interesting findings discovered about the algorithm.

Introduction

In terms of the introduction of the algorithm, what we used in this case is a classic neural network. It is a neural network that contains three levels which are input layer, hidden layer and output layer. Each neuron has a weight for an input, an offset, and an activation function, so the overall process in this task is the input, and then it goes through the first layer of neuron operations (multiply the weights, plus paranoid, to activate the function operations) to get the output. Then the output of the first layer can be treated as input of the second layer…until the operation to the output layer, and then get the result. Neural networks rely on such set of mechanisms for calculations and predictions.

Accuracy

The accuracy could be affected by many hyperparameters such as maximum iteration, layers, blockSize and seed. We want to discover the impact of the size of hidden layer on the prediction accuracy. Therefore, the maximum iteration is set as 100, the blockSize is set as 128 and the seed is set as 1234 which are the default values. In terms of layer size, we only test three values of hidden layer size and the accuracy statistic data is shown below.

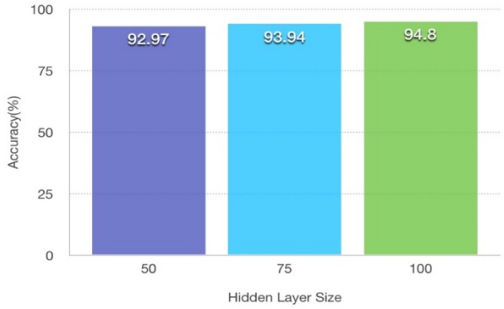


Figure 13. Accuracy of each hidden layer size

In detail, when the hidden layer size is set 50, the accuracy is equal to 92.97%. With hidden layer size increasing to 75, the accuracy will increase to 93.94%. And the accuracy is highest in the test statistics which is equal to 94.8% when the hidden layer size is 100. It is clearly that the accuracy will increase when we increase the hidden layer size.

How to prepare& collect

As mentioned before, a Multilayer Perceptron Classifier has many hyperparameters such as maximum iteration, layers, blockSize that could affect the execution statistics as well as the prediction accuracy. In this case, what we want to investigate is the impact of the size of hidden layers on execution statistics and prediction accuracy. The number of units in the hidden layer is set between five and ten times the number of units in output layer (Here, the three values I set are 50, 75, 100 respectively). In the preparing stage, spark.read method is used to read the raw dataset and vectorAssembler is used to extract features from the file. Therefore, the input layer has 784 input units which is equal to the features of training data. In addition, the output layer has 10 units which is equal to the number of labels. The predicted labels could be obtained after applying the pipeline to the test dataset. The evaluator could get the prediction accuracy by evaluating the prediction data. Therefore, the accuracy would be displayed at the end.

Execution statistics& Interesting Finds

In related to the execution statistics, I have used three different values in setting the number of executors and cores. The data is shown in Table 2.

Params / HiddenLayerSize	2 executors, 4 cores	4 executors, 2 cores	8 executors, 2 cores
50	70s	81s	107s
75	100s	87s	117s
100	111s	109s	106s

Figure 14. Running time of different settings

It is clear that both of the hidden layer size and different executors setting have impact on the running time. In general, more hidden layer size would result in consuming more running time but the accuracy would be increased at the same time. Therefore, we should balance these two factors in the future similar tasks. In addition, the performance of 2 executors& 4 cores and 4 executors& 2 cores is similar after many of my experiments which is out of my expectation so I think the running time is not only influenced by the setting of executors and cores, it can be also influenced by the cluster's situation such as running time, number of users and so on.

When I collect these data and execute the task, one of the most interesting things I found was the performance of 8 executors& 2 cores. More specifically, I suppose that the task can be executed by 8 executors in parallel which could reduce the running time. However, I found that the running time is increased compared with the task which I set 4 executors or 2 executors. After I reviewed the detailed timeline, I think I found the reason. The timeline is shown in the Figure15.

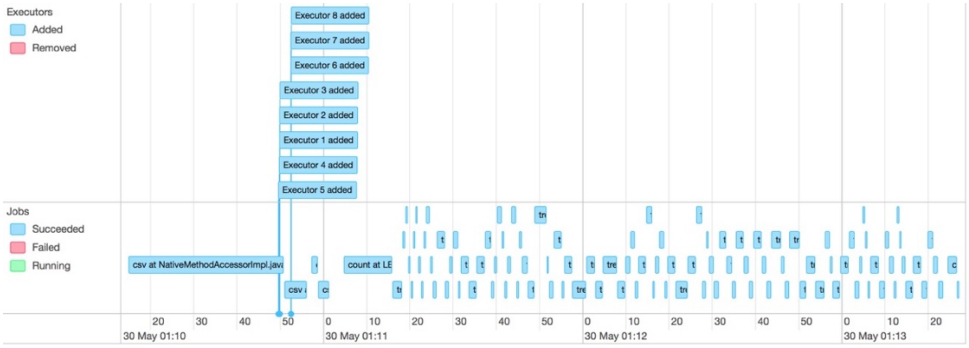


Figure 15. The Event time line for 8 executors 2 cores

As the figure shows, although 8 executors have been added at the beginning, the actual jobs were doing by 3 or 4 executors in parallel. I think the algorithm allocates executors based on computational complexity. In this experiment, the computation does not require 8 such executors to calculate, so only 3 or 4 executors are assigned to perform the task.

Comparison of two algorithms

In the point of accuracy, Logistic regression is slightly lower than Multilayer Perceptron Classifier when the hidden layer size is large. However, this does not mean that Logistic regression is less accurate than Multilayer Perceptron Classifier. We can improve the accuracy by adjusting other parameters of the two algorithms in order to obtain the higher accuracy. Besides, in the point of execution statistics, comparing the Event time table of Logistic Regression and Multilayer Perception Classifier in 8 executors (Figure 11 and Figure 15), Logistic Regression has the bigger calculation amount because the 8 executors are all parallelized but for Multilayer Perception Classifier there are only 4. This is due to the iterations of gradient decent to update the weights. In related to the running time, Logistic regression's overall performance is better than Multilayer Perceptron Classifier. This should be attributed to the low algorithm complexity of Logistic regression. Overall, both of these two algorithms have their advantages and disadvantages, in this case, we need to make choices in accuracy and runtime.