Index

Spark Machine Learning Application

1 Introduction

In this assignment, we use the spark framework and the pyspark.ml library to implement machine learning. Basically, we have designed and implemented a KNN classifier and analyzed the performance of the algorithm. Then we used some high-level functions from the pyspark.ml library algorithms to implement Random Forest Classifier and Multilayer Perceptron Classifier and analyzed the performances.

2 Design

2.1 Architecture of implementation

The Figure 1 below is the architecture. The inputs are train and test .csv files. At first, we read in those two files and store into DataFrame type with label and data separated. Then we victoried them and gave titles to some columns. Then, the train data will be pass into the PCA function and become a PCA model as output. The train and test data will be put into the PCA model and transformed into DataFrame by selected with specific title. Now, for better performance in KNN function, the two DataFrame variables will be transformed into array then matrix and then RDD type. The benefits in performance will be mentioned later. Then the test data will be classified by our KNN function. Finally, we will do statistics and work

out the precision, recall, accuracy and f1-score

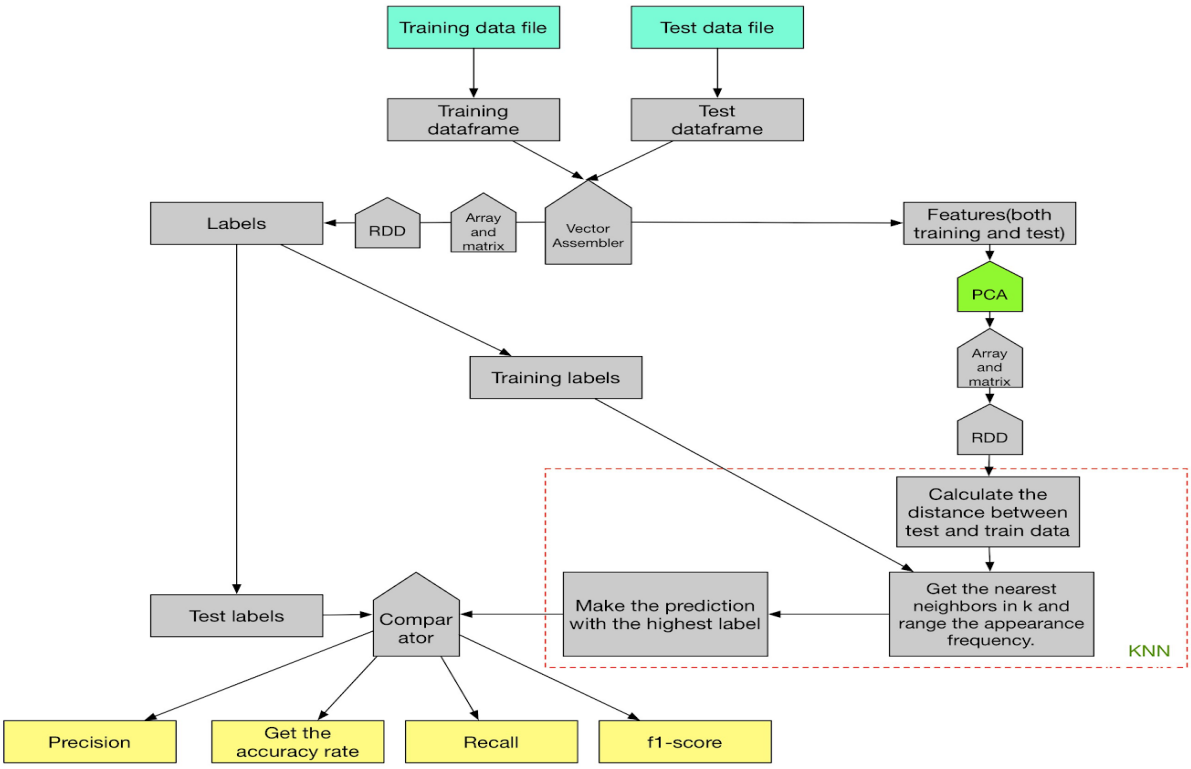

Figure 1 Architecture of Implementation

2.2 KNN function

The Figure 2 below is the KNN function.

The input of KNN includes parallelized matriculated results of train and test data after PCA function.

Test input minuses train input, then the output is the distances of a test record to each train record in each dimension. Then calculate the quadratic sum of the previous output and then the square root is the array of distances in all dimensions. Then we sort and count and sort again, now we have a dictionary which contains the train labels sorted by the distance to the test data.

Pick out the first record in the dictionary as the output. That the predict

label of one input data.

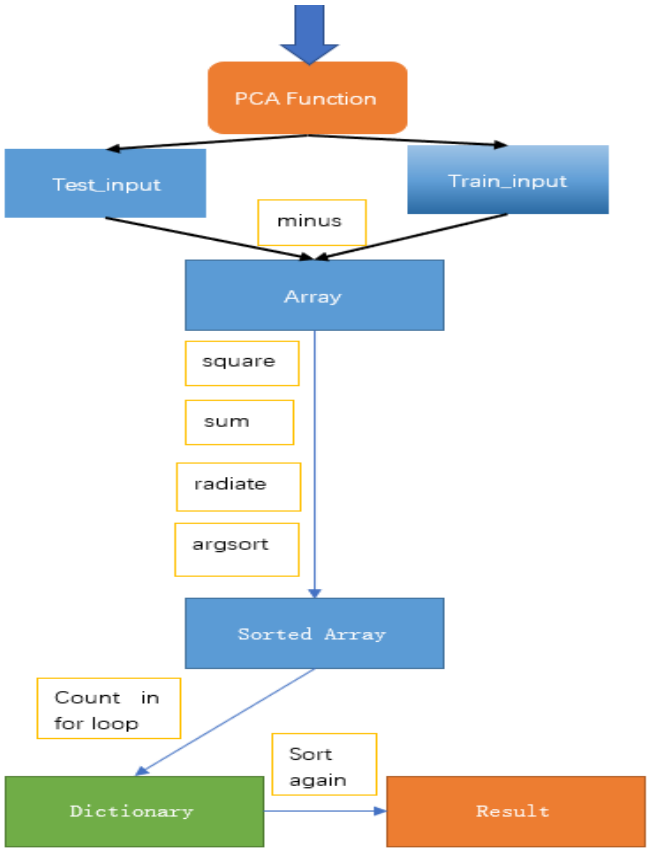

Figure 2 KNN Function

2.3 PCA

PCA is the function which can reduce the dimensions of the input data. In PCA, only the most important and characteristic elements can be reserved which the others will be delete. The parameter reduced dimension d is the dimensions we want to reserve. The larger d will be more precise but also will cost more time in processing.

2.4 Performance optimization

```
local_pca=np.array(pca_result.collect())#train data
local_pca=local_pca.reshape((60000,50))

test_local_pca=test_local_pca.reshape((10000,50)) #test data
test_local_label=test_local_label.reshape((10000,1))
```

```
test_local_label=test_local_label[:,0]

test_data_pca_rdd = sc.parallelize(test_local_pca)

def KNNFunction(inX):
    ...

gain_label_rdd = test_data_pca_rdd.map(KNNFunction)
```

This short code snippet is the key for better performance.

Before the traversing, the input data will be preprocessed into the wanted datatype. So that, only the indispensable operations are in the KNN function. Then we use the RDD and map operation to traverse all test data records using KNN function for reducing processing time with the help of distributed file system. Then output here is the predicted labels.

2.5 Sample result

This sample is run on our own machine which has 1 executor with 16 cores. The reduced dimension d is 50 and the value k of KNN is 10.

num-executors=1, executor_cores = 16					
d, k	label	precision (%)	accuracy (%)	recall (%)	F1-score (%)
50,10	0	97.98	97.52	80.98	88.68
80,10		97.49	97.33	80.07	87.92
80,7		97.49	97.38	80.41	88.13
50,7		98.08	97.53	81.00	88.73
50,10	1	97.08	97.52	84.08	90.11

80,10		96.42	97.33	83.39	89.43
80,7		97.00	97.38	83.27	89.61
50,7		97.41	97.53	83.88	90.14

Table 1 Sample Result of First Stage

The Table 1 above is part of the sample result of first stage. The whole table is in the appendix.

The time cost is 168.3 seconds.

The I/O cost is:

Input	Shuffle Read	Shuffle Write
636.7 MB	0.0 B	15.5 MB

Figure 3 I/O Cost of Sample Result

3 Performance analysis

3.1 Execution environment and hyperparameters

This program is run on our own machine which has 1 executor with 16 cores.

Two values of reduced dimension d are chosen: 50 and 80.

Two values of k in KNN function are chosen: 7 and 10.

Three values of core numbers are chosen: 4, 8 and 16.

So, we have 12 results.

We observed three indexes: time cost, I/O cost.

3.2 Results

Case number	Executor-core number	d, k	summary execution statistics such as **execution time** and **total I/O cost**
Case 1:	16	50,10	168.315s, 15.5MB
		50,7	163.12s, 15.5MB
		80,10	229.217s, 15.5MB
		80,7	229.216s, 15.5MB
Case 2:	8	50,10	157.217s, 8.3MB
		50,7	157.217s, 15.5MB
		80,10	241.419s, 8.3MB
		80,7	241.116s, 8.3MB
Case 3:	4	50,10	160.415s, 0MB
		50,7	154.615s, 15.5MB
		80,10	246.917s, 8.3MB
		80,7	238.516s, 0.0MB

Table 2 Results of Performance Analysis

3.3 Performance analysis

In the Table 2, we can observe that, for most time cost in the records, the more cores used, the less time costed.

In our program, many operations are computing intensive, such as PCA and KNN. Both need to traverse the whole dataset. And we used RDD to separate those computing into distributed file system for parallel processing. The more cores we can use, the more data can be

processing at the same time. That's why we can use less processing time with more cores.

While for the I/O cost, we can observe that, the more cores are used, the more I/O resources costed.

The key is also about the distributed processing. RDD is a distributed file system which can processing more data at the same time. While it is distributed, it has duplicate data for redundant. That's why the more cores we use, the more I/O resources costed.

4 Spark Classifier Exploration

4.1 Random forest classifier

4.1.1 Basic introduction

Random forest is a common fusion learning method in classification and regression problems. This algorithm builds multiple decision trees based on different subsets of training data and combines them into a new model. The prediction result is a combination of all decision tree outputs, which reduces fluctuations and improves the accuracy of the prediction. For the random forest classification model, the prediction result of each tree is considered as a vote. The category that got the most votes is the predicted category.

4.1.2 Data preprocessing

Input the .csv files and get two DataFrame type variables. Then we

use VectorAssembler method to add title to label and data columns for two DataFrame variables and get two vectors. Then use StringIndexer to convert them into DataFrame type with data types are integer(label) and vectors(data/feature). Now, we can use the random forest API in pyspark.ml.feature package.

4.1.3 Output collecting

Use indexToString method to convert integer index into string index. Then use pipeline to generate a model of random forest classifier. Using this model to test data, we can get a result which is predictions. And use evaluator method to work out the accuracy.

4.2 Multilayer Perceptron classifier

4.2.1 Basic introduction

The multi-layer perceptron is extended by perceptron. The most important feature is that there are multiple neuronal layers. The first layer is identifying as the input layer, and the final layer is the output layer. So, all the other layer in the middle layer is hidden layer.

A single perceptron cannot identify some linearly data, using MLP can help to overcome the weaknesses

4.2.2 Data preprocessing

```
train_assembler = VectorAssembler(inputCols=train_df.columns[1:],
    outputCol="features")
train_vectors = train_assembler.transform(train_df).select(col(train_df.columns[0]).alias("label"),"features")
train_vectors.show(5)
```

Extracting feature index data from source data, some columns of data need to be converted into feature vectors. For the module data fitting in the next step, the data format should be integer (label) and vectors (features). Then we can use the MultilayerPerceptronClassifier method.

4.2.3 Output collecting

Definition of the layer and then set the parameters for the trainer and use the trainer and data to train model which can be used to predict as below:

```
+-----+--------------------+----------+
|label|            features|prediction|
+-----+--------------------+----------+
|    7|(784,[202,203,204...|       7.0|
|    2|(784,[94,95,96,97...|       2.0|
|    1|(784,[128,129,130...|       1.0|
|    0|(784,[124,125,126...|       0.0|
|    4|(784,[150,151,159...|       4.0|
+-----+--------------------+----------+
only showing top 5 rows
```

4.3 Comparing of their performance

4.3.1 Result of Random forest classifier

Tree number	accuracy (%)	execution time, total I/O cost
10	80.055	41.1s,40.4MB
20	83.9733	70s, 60.7MB
50	85.4267	48.7s,119.5MB

4.3.2 Result of Multilayer Perceptron classifier

block Size(MB)	Accuracy (%)	execution time, total I/O cost
10	0.9487	49.5s, 818.9 MB

20	0.946	49.1s, 819.9 MB
30	0.9477	50.0s, 824 MB

4.3.3 Analysis

For the Random Forest, once we add more Tree in the Random Forest, higher accuracy can we get. But by increasing this parameter, computer need more execution times and total I/O cost. So, we need to balance the performance and the time

For the multilayer perceptron classification, block size may not influence the accuracy by testing the MNIST data set. And if we run the program twice with the same input parameter, the result still has a slightly different. Our group think this is due to the function itself, maybe it has some random functions inside.

In a conclusion, from the performance of the two algorithms after the dataset and parameters adjustment, we also verified the factors that affect the accuracy of the two algorithms theoretically.

For random forests, in general, the deeper the decision tree, the smaller the deviation of the model fitting and the higher the accuracy, but the greater the cost of fitting at the same time.

For multi-layer perceptron, the accuracy rate depends on the amount of data. The greater the amount of data, the better the training effect, but at the same time we should pay attention to the problem of overfitting.

5 Appendix

num-executors=1, executor_cores = 16					
d, k	label	precision (%)	accuracy (%)	recall (%)	F1-score (%)
50,10	0	97.98	97.52	80.98	88.68
80,10		97.49	97.33	80.07	87.92
80,7		97.49	97.38	80.41	88.13
50,7		98.08	97.53	81.00	88.73
50,10	1	97.08	97.52	84.08	90.11
80,10		96.42	97.33	83.39	89.43
80,7		97.00	97.38	83.27	89.61
50,7		97.41	97.53	83.88	90.14
50,10	2	98.91	97.52	80.83	88.96
80,10		98.52	97.33	79.81	88.18
80,7		98.42	97.38	80.21	88.39
50,7		98.53	97.53	81.20	89.03

50,10	3	97.50	97.52	81.42	88.74
80,10		98.29	97.33	79.61	87.97
80,7		97.79	97.38	80.23	88.14
50,7		97.60	97.53	81.37	88.75
50,10	4	97.95	97.52	80.71	88.50
80,10		97.93	97.33	79.33	87.66
80,7		97.93	97.38	79.65	87.85
50,7		97.94	97.53	80.76	88.53
50,10	5	97.43	97.52	79.47	87.54
80,10		96.88	97.33	78.45	86.70
80,7		96.99	97.38	78.75	86.93
50,7		97.09	97.53	79.71	87.54
50,10	6	97.53	97.52	80.89	88.43
80,10		97.53	97.33	79.56	87.63
80,7		97.93	97.38	79.63	87.84
50,7		97.63	97.53	80.89	88.47

50,10	7	96.87	97.52	82.10	88.88
80,10		96.40	97.33	81.18	88.14
80,7		96.23	97.38	81.71	88.38
50,7		96.79	97.53	82.31	88.97
50,10	8	98.32	97.52	80.14	88.30
80,10		98.52	97.33	78.67	87.48
80,7		98.21	97.38	79.22	87.70
50,7		98.12	97.53	80.41	88.39
50,10	9	95.78	97.52	82.63	88.72
80,10		95.57	97.33	81.38	87.90
80,7		95.94	97.38	81.44	88.10
50,7		96.14	97.53	82.36	88.72