COMP5349 - Assignment 2

Spark Machine Learning Application

Group Number: SIT457_7

Group Member: Bo Wang (bwan3986), Ruoyu Zhuang (rzhu9225)

Date: 2018/5/30

1 Introduction

In this assignment, we have performed spark machine learning for the prediction of the MNIST data set with hand-written digits. The train dataset and test dataset contain 60,000 and 10,000 images/labels respectively where each image has a resolution of 28 x 28 (feature dimension of 748). Three stages, namely design of kNN classifier, corresponding performance analysis and spark classifier exploration are involved.

In the first stage, the each procedure of our kNN classifier is described in details following a general design workflow, including data loading, data pre-processing, data classification, and results evaluation. There are several performance optimization features in data structure, parallelization and data locality involved. A sample result are provided in this section as well.

In the second stage, the kNN classifier was executed on the dataset 12 times among variations of executor and core numbers, number of reduced dimensions and number of nearest neighbours. The execution statistics such as runtime in total and main stages such as PCA and classification, and input as well as shuffling cost are recorded to compare the performances. Then the direct acyclic graph of transformation chain and physical plan in Spark for some stage are analysed and discussed.

In the third stage, decision tree classifier and multilayer perceptron classifier from Spark ML library are experimented. Three variations of one hyperparameter which should have impact on the selected classifiers' time complexity are tested and their performances in Spark are compared. The hyperparameters are input feature number and size of hidden layer for decision tree and multilayer perceptron, respectively. There are also some interesting findings presented as extensive analysis in the end.

2 Design

The KNN classifier is implemented on Spark with Python and the architecture of the implementation could be seen in figure 1 below.

The blue blocks stands for intermediate variables with their data types in grey. The methods applied in the workflow and some of their brief description are given in green. Four inputs including files and values are marked as orange.

Detailed explanations of each stage are provided in the following sections.

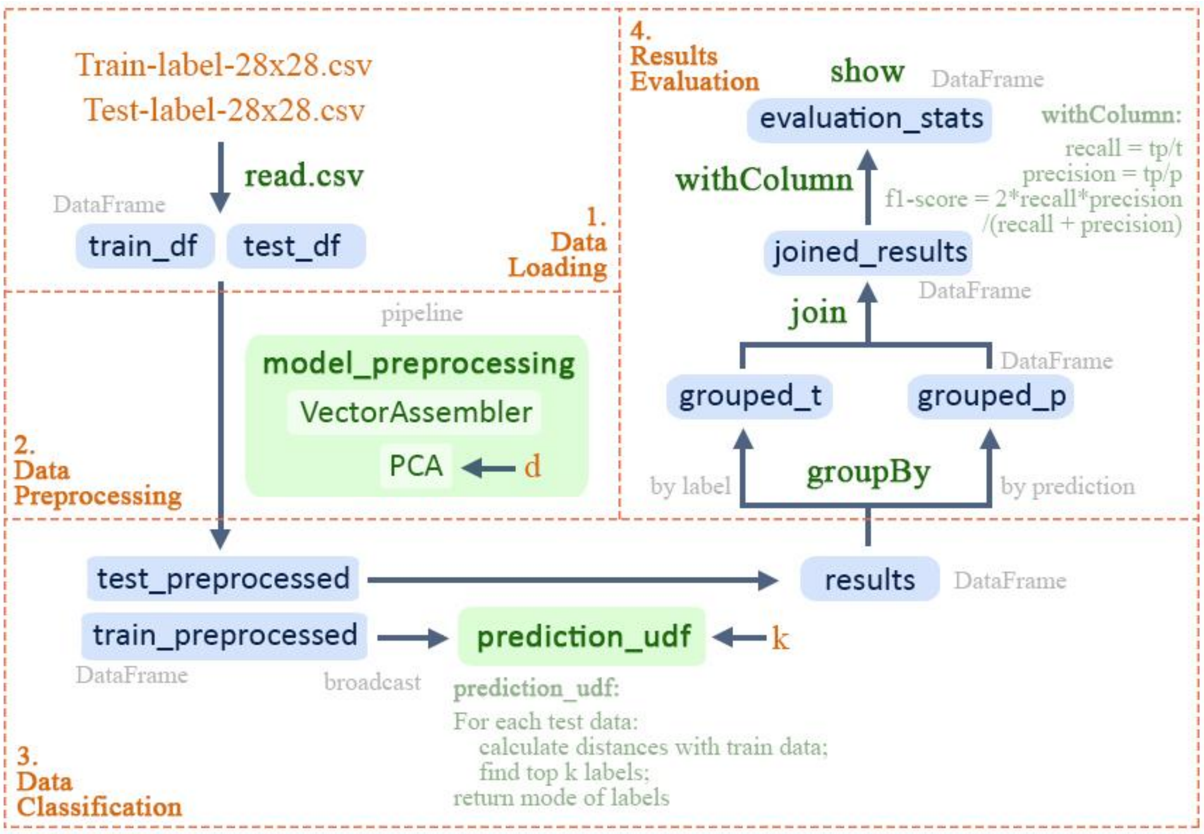


Figure 1. Workflow in Spark

2.1 Data Loading

In this implementation, a SparkSession is first built as entry point for fully functionalities which include dataframe creation, dataframe table registration and etc. Then we loaded both the training dataset and testing dataset into two separated dataframes (test_df and train_df) with each consists of two columns. Each row of the first columns from both dataframes represents one true label whereas that of the rest columns (the second column to the 749th column) store the corresponding raw features.

2.2 Data Pre-processing

The VectorAssembler is applied to combine the columns containing the raw features into one single feature vector such that could be used to fit the PCA model and the KNN classifier in the further steps. The train data feature vector is then used to fit the PCA model where the 784 feature dimensions are reduced to 50 &100 feature dimensions. The VectorAssembler and PCA are regarded as two transformers and pipelined together.

To enforce better parallelization, we have repartitioned the test_preprocessed dataframe into 20 partitions as show in the following code snippet.

```
test_preprocessed = model_preprocessing.transform(test_df) \
                    .select(["label", "features"]).repartition(20)
```

However, the number of partitions required extra care to determine due to the trade-off between the cost of inter-partition communication and lower execution time for each partition with smaller sizes (i.e. larger number of parallelization tasks).

After transforming both the training feature vector and the testing feature vector to a lower dimension, the low dimension training feature vector and their corresponding label are collected as array and broadcasted where training feature array is reshaped as no. of data x no. of features (i.e. each data point is represented by one column and each row represents a particular PCA pre-processed feature). Since the prediction using KNN classifier requires input of train data whose size is large, it is expensive to copy the data multiple times for different tasks. Fortunately, broadcast function in Spark helps broadcasting such information across nodes in clusters.

When the broadcast function is called as shown in the following code snippet, the training data and label arrays are not sent each of the executor but cached deserialized locally at a storage level of MEMORY_AND_DISK (i.e. if memory does not fit store in disk) and serialized into locally saved blocks.

```
    train_data_bc = spark.sparkContext.broadcast(np.array([i[1] for i in broadcast_data]) \
                    .reshape((60000, d)))
    train_label_bc = spark.sparkContext.broadcast(np.array([i[0] for i in broadcast_data]))
```

When the executors are created, they only carry the lightweight information of the value identifiers and their corresponding no. of blocks. The broadcasted values would only be copied into each executor once at the time the values are firstly accessed by the executor though value method.

2.3 Data Classification

To perform KNN classifier, we defined a UDF (user defined function) and map this function to each row of the vectorized test data feature Dataframe using withColumn as shown below. The prediction results are returned as integer type and stored the dataframe column named "prediction".

```
    prediction_udf = udf(prediction, IntegerType())
    result_knn = test_preprocessed.withColumn("prediction", prediction_udf("features")) \
                .select("prediction", "label")
```

The detailed UDF is showing in the following code snippet.

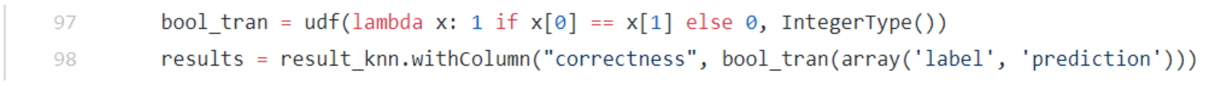
```
    dist_array = np.linalg.norm(train_data_bc.value - test_X, axis=1)
    dist_idx = np.argpartition(dist_array, k, axis=0)[:k]

    klass = np.take(train_label_bc.value, dist_idx)
    result = mode(klass, axis=0)[0]
    return int(result)
```

In the prediction process, the Euclidian distances between features from a particular test data point and that from each train data point firstly are computed. This is achieved by performing the column-wise subtraction between the broadcasted training data feature array and the single testing data feature vector followed by the 2-order normalization within each resulted column.

The indirect partition is then preformed using Introsort algorithm (i.e. hybrid of quicksort and heapsort) to get the indices with the k lowest Euclidian distances. The obtained indices are used to retrieve their corresponding labels from the broadcasted training data label array. Finally, the test data point is classified as the most common class appeared in the k points. In this stage, the results with two columns, the predicted and true labels are collected for further analysis.

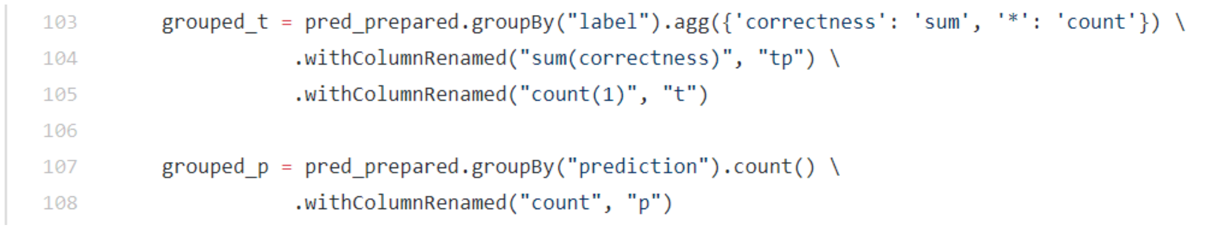
2.4 Results Evaluation

In the evaluation stage, the correctness of predictions is computed with a lambda UDF and saved into a dataframe column named “correctness” as shown below.



```
bool_tran = udf(lambda x: 1 if x[0] == x[1] else 0, IntegerType())
results = result_knn.withColumn("correctness", bool_tran(array('label', 'prediction')))
```

Then the counts of the true positive and total counts for each label are calculated with groupBy method and aggregation within each group to form the dataframe “grouped_t”. Similarly, the counts of the total positive prediction are calculated with groupBy and count method to form the dataframe “grouped_p”.



```
grouped_t = pred_prepared.groupBy("label").agg({'correctness': 'sum', '*': 'count'}) \
            .withColumnRenamed("sum(correctness)", "tp") \
            .withColumnRenamed("count(1)", "t")

grouped_p = pred_prepared.groupBy("prediction").count() \
            .withColumnRenamed("count", "p")
```

Afterwards, the two dataframes “grouped_t” and “grouped_p” are inner joined by label of the dataset (0-9) to form dataframe with columns named “label”, “tp”, “p” and “t” which represents the counts of true positive, positive and label occurrences respectively. The resultant dataframe is persisted for the following jobs. Finally, the precisions, recalls and f1-scores are computed for each label with withColumn method using the following formulas and sorted by label.

$$Precision = \frac{tp}{p}$$

$$Recall = \frac{tp}{t}$$

$$F1 - score = 2 * \frac{\frac{tp}{p} * \frac{tp}{t}}{\frac{tp}{p} + \frac{tp}{t}}$$

2.5 Parallelization

All the testing dataset is read into the dataframe in a SparkSession. Then automatically partitioned and run in parallel by Spark during the pre-process stage, which allows more flexible scheduling at the beginning of the whole application. After that, the test data is repartitioned to 20 partitions to ensure better parallelization in the following stages.

In the prediction stage, the UDF is applied to each row of the dataframe of the testing dataset, parallelized with the rows from different partitions applying the UDF concurrently.

In the evaluation stage, the computation of prediction correctness has similar condition as the one in prediction. In addition, all goupBy, aggregation, count and inner join methods are running in parallel, and the operations are optimized by Spark DataFrame API during transformation from logical plan to physical plan.

2.6 Performance Optimization Features

(1) Data Structure

In our design, inside the main flows of pre-processing, classification and evaluation, most of the transformations and calculations are based on Spark DataFrame API, which has sufficient optimizing methods for better parallelism.

All the data is stored as array of objects (e.g. DataFrame and np.Array) and primitive datatypes (e.g. Int) instead of collection classes (e.g. HashMap) and numeric IDs (Integer IDs) instead of string type are used in our dataframe. Both implementations effectively reduced the memory consumptions and thus resulted the better performance.

(2) Level of Parallelization

By default, Spark automatically determines the number of tasks in “map” process based on the size of files and the “reduce” operations in Spark uses the largest number of parent partitions. In our design, Spark divided the testing dataset into a small number of partitions during csv reading (i.e. “map” process) and a sequent low number of partitions in UDF mapping process during the prediction. Thus, we have forced the testing dataset to be repartitioned into 20 partitions with each partition contains approximate 500 records.

Based on the Spark documentation, 2-3 tasks per core in the cluster is recommended. [1] Therefore, we decide to set one of the execution parameter as the number of executors to be 5 and the number of cores in executor to be 2.

(3) Broadcast training dataset

As aforementioned, we have broadcasted the training dataset due to its relatively larger size and high frequency of usage in each task. By broadcasting, the size of each serialized task and the cost of launching a job in cluster are largely reduced.

(4) Data Persist

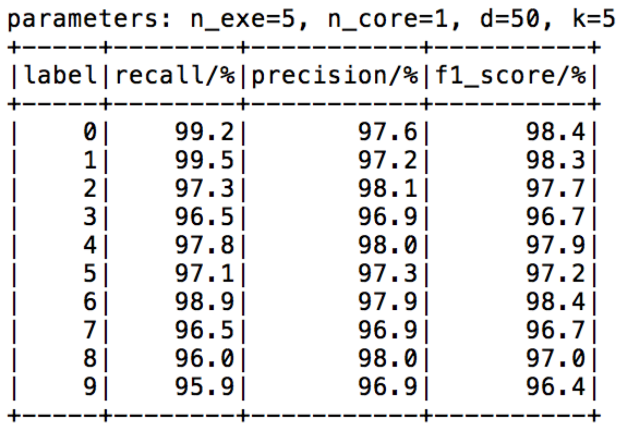
As previously mentioned, since the results from classifications required reusing during the evaluation process, we persist the resultant dataframe into the MENORY-ONLY storage level such that the further processes would be much faster without recalculation.

(5) Data Locality

With a relatively high parallelized algorithm and reasonable partitioning, it becomes possible for executors to work on the intermediate results locally, which are persisted from the same partitions in the previous stages. In practical, most of tasks in prediction and evaluation stages have the locality level of NODE-LOCAL and even PROCESS-LOCAL.

2.7 Sample Results

The sample result from the KNN classification is shown below and the corresponding hyperparameters are also listed where "n_exe", "n-core", "d" and "k" stand for the number of executors, the number of executor cores, the reduced dimension and the nearest neighbour number respectively.



```
parameters: n_exe=5, n_core=1, d=50, k=5
+-----+--------+-----------+----------+
|label|recall/%|precision/%|f1_score/%|
+-----+--------+-----------+----------+
|    0|    99.2|       97.6|      98.4|
|    1|    99.5|       97.2|      98.3|
|    2|    97.3|       98.1|      97.7|
|    3|    96.5|       96.9|      96.7|
|    4|    97.8|       98.0|      97.9|
|    5|    97.1|       97.3|      97.2|
|    6|    98.9|       97.9|      98.4|
|    7|    96.5|       96.9|      96.7|
|    8|    96.0|       98.0|      97.0|
|    9|    95.9|       96.9|      96.4|
+-----+--------+-----------+----------+
```

Figure 2. Sample Results of kNN

3 Performance Analysis

3.1 Execution Environment and Hyperparameters

Our kNN algorithm is implemented on SIT lab cluster with different combinations of hyperparameters to compare the performance.

Some of the basic settings of the **cluster environment** are:

- Nodes: 30 identical Linux virtual machines (1 master node, 29 slave nodes)
- Node Physical Capacity: 4 cores, 8G memory, 250G storage
- Spark Scheduler Mode: FIFO
- Spark Driver Memory: 1G
- Spark Executor Memory: 2G
- Spark Submit Deploy Mode: client

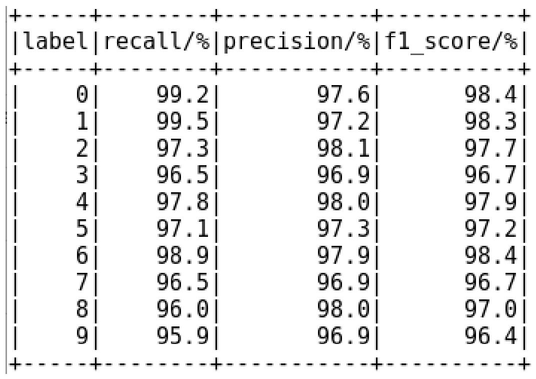
There are 3 **hyperparameters** taken into account:

- Executor Number * Executor Cores: 4*1, 4*2, 5*2
- d (number of reduced dimension in PCA): 50, 80
- k (number of nearest neighbours in kNN): 5, 10

So in total, 12 combinations would be discussed in this section.

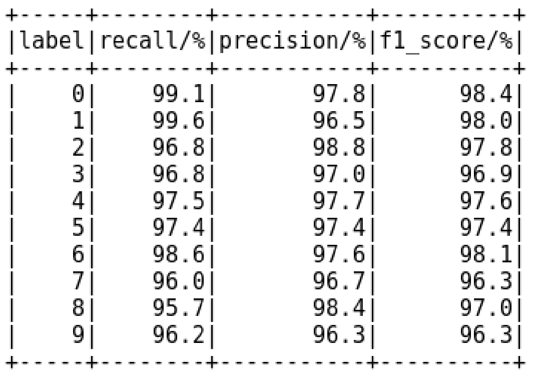
3.2 Results

The results with different combinations of d (reduced dimensions) and k (nearest neighbours) are given in the figure 3 to 6 below:



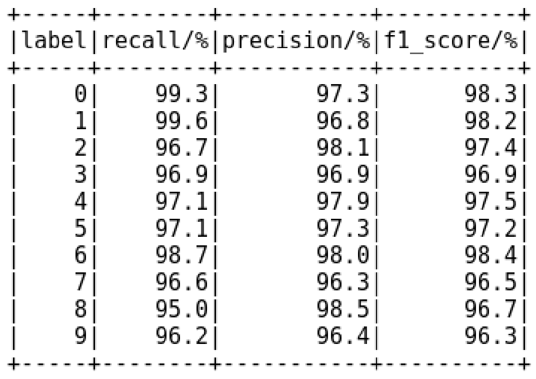
```
+-----+--------+-----------+----------+
|label|recall/%|precision/%|f1_score/%|
+-----+--------+-----------+----------+
|    0|    99.2|       97.6|      98.4|
|    1|    99.5|       97.2|      98.3|
|    2|    97.3|       98.1|      97.7|
|    3|    96.5|       96.9|      96.7|
|    4|    97.8|       98.0|      97.9|
|    5|    97.1|       97.3|      97.2|
|    6|    98.9|       97.9|      98.4|
|    7|    96.5|       96.9|      96.7|
|    8|    96.0|       98.0|      97.0|
|    9|    95.9|       96.9|      96.4|
+-----+--------+-----------+----------+
```

Figure 3. Results of d=50, k=5



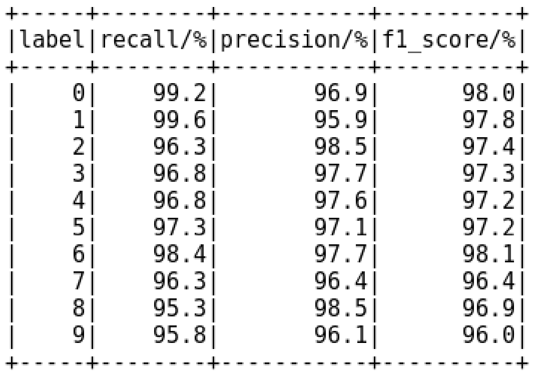
```
+-----+--------+-----------+----------+
|label|recall/%|precision/%|f1_score/%|
+-----+--------+-----------+----------+
|    0|    99.1|       97.8|      98.4|
|    1|    99.6|       96.5|      98.0|
|    2|    96.8|       98.8|      97.8|
|    3|    96.8|       97.0|      96.9|
|    4|    97.5|       97.7|      97.6|
|    5|    97.4|       97.4|      97.4|
|    6|    98.6|       97.6|      98.1|
|    7|    96.0|       96.7|      96.3|
|    8|    95.7|       98.4|      97.0|
|    9|    96.2|       96.3|      96.3|
+-----+--------+-----------+----------+
```

Figure 4. Results of d=50, k=10



```
+-----+--------+-----------+----------+
|label|recall/%|precision/%|f1_score/%|
+-----+--------+-----------+----------+
|    0|    99.3|       97.3|      98.3|
|    1|    99.6|       96.8|      98.2|
|    2|    96.7|       98.1|      97.4|
|    3|    96.9|       96.9|      96.9|
|    4|    97.1|       97.9|      97.5|
|    5|    97.1|       97.3|      97.2|
|    6|    98.7|       98.0|      98.4|
|    7|    96.6|       96.3|      96.5|
|    8|    95.0|       98.5|      96.7|
|    9|    96.2|       96.4|      96.3|
+-----+--------+-----------+----------+
```

Figure 5. Results of d=80, k=5



```
+-----+--------+-----------+----------+
|label|recall/%|precision/%|f1_score/%|
+-----+--------+-----------+----------+
|    0|    99.2|       96.9|      98.0|
|    1|    99.6|       95.9|      97.8|
|    2|    96.3|       98.5|      97.4|
|    3|    96.8|       97.7|      97.3|
|    4|    96.8|       97.6|      97.2|
|    5|    97.3|       97.1|      97.2|
|    6|    98.4|       97.7|      98.1|
|    7|    96.3|       96.4|      96.4|
|    8|    95.3|       98.5|      96.9|
|    9|    95.8|       96.1|      96.0|
+-----+--------+-----------+----------+
```

Figure 6. Results of d=80, k=10

Generally, the kNN classifier has successfully classified most images with all recalls, precisions and f1 scores higher than 95%. It works better in figuring out digit 0, 1 and 6, while it is also more likely to be wrongly predicted as 3, 7 and 9, which might be due to the similar shapes of these digits in hand-writing. And the values of d and k actually have little influence on the accuracy.

3.3 Accuracy and Execution Statistics

(1) General Analysis

The summary of accuracies and execution statistics for 12 sets parameters are tabled below. The accuracies only depend on the values of k and the number of reduced feature dimensions, which merely change for our selections of hyper-parameters.

The total number of jobs for all cases are the same whereas that of stages and tasks vary with different execution setups. When the number executor-core changed from single core to two cores, the number of tasks increase from 14 to 16 which is due to the extra shuffling introduced

Table 1. Total Execution Statistics for kNN

Num of Features	k	Num of Executors	Num of Cores per Executor	Accuracy	Jobs/Stage/ Tasks	Total Task Time	Total Execution Time	Input	Shuffle Read	Shuffle Write
50	5	4	1	97.48%	11/14/252	13 min	3.5 min	475.8 MB	3.1 MB	4.1 MB
50	5	4	2	97.48%	11/16/277	13 min	2.2 min	476.8 MB	7.5 MB	12.4 MB
50	5	5	2	97.48%	11/16/287	13 min	1.6 min	453.3 MB	11.4 MB	14.3 MB
50	10	4	1	97.40%	11/14/252	13 min	3.7 min	451.6 MB	3.1 MB	4.1 MB
50	10	4	2	97.40%	11/16/277	13 min	2.1 min	476.9 MB	7.2 MB	12.4 MB
50	10	5	2	97.40%	11/16/287	14 min	1.7 min	453.4 MB	11.4 MB	14.3 MB
80	5	4	1	97.35%	11/14/252	18 min	5.1 min	475.8 MB	4.9 MB	6.5 MB
80	5	4	2	97.35%	11/16/277	21 min	3.4 min	476.9 MB	8.7 MB	14.8 MB
80	5	5	2	97.35%	11/16/287	20 min	2.3 min	476.1 MB	13.3 MB	16.7 MB
80	10	4	1	97.22%	11/14/252	18 min	5.0 min	447.4 MB	4.9 MB	6.5 MB
80	10	4	2	97.22%	11/16/277	21 min	3.2 min	476.9 MB	9 MB	14.8 MB
80	10	5	2	97.22%	11/16/287	21 min	2.5 min	476.2 MB	13.2 MB	16.7 MB

Table 2. Detailed Execution Statistics for PCA, Broadcast, and kNN Operations

Num of Features	k	Num of Executors	Num of Cores per Executor	PCA Execution Time	Broadcast Execution Time	kNN Execution Time
50	5	4	1	17 s	6 s	2.9 min
50	5	4	2	13 s	3 s	1.8 min
50	5	5	2	12 s	3 s	1.2 min
50	10	4	1	17 s	5 s	3.2 min
50	10	4	2	11 s	3 s	1.7 min
50	10	5	2	13 s	3 s	1.3 min
80	5	4	1	14 s	6 s	4.6 min
80	5	4	2	12 s	4 s	3.0 min
80	5	5	2	11 s	3 s	1.9 min
80	10	4	1	22 s	10 s	4.3 min
80	10	4	2	12 s	4 s	2.8 min
80	10	5	2	14 s	4 s	2.0 min

in the PCA processes. Similarly, the number of tasks would increase when a higher parallelization level is implemented.

The total inputs are consistent among the combinations, except for the different staged PCAs, which actually depends on the Spark scheduler at the beginning of the processing, such as different executor allocations and waiting times.

The total task time describes the simple sum of the runtime of each task which is majorly influenced by the number of feature dimensions though some fluctuations could be observed due to the variation of the cluster performance. On the other hand, the real execution time could be calculated using following formula:

$$Real\ Execution\ Time \cong \frac{Total\ Task\ Time}{N_{Executor} * N_{Core\ per\ executor}}$$

Where $N_{Excutor}$ and $N_{Core\_per\_executor}$ are the number of executors and the number of cores per executors respectively. In practise, the real execution time would be slightly larger than the result calculated which is attributed from factors such as the network traffic and the non-perfect parallelization.

(2) Detailed Analysis for Some Stages

The following two figures indicate that if the number of partitions does not match the number of the total number of cores (i.e. the product of executor number and core number per executor), the tasks could not be evenly distributed into each core. Consequently, some cores would stay idle without contributing to parallelization of the computation, and the total task execution time would be bottlenecked by the runtime of cores with more tasks.

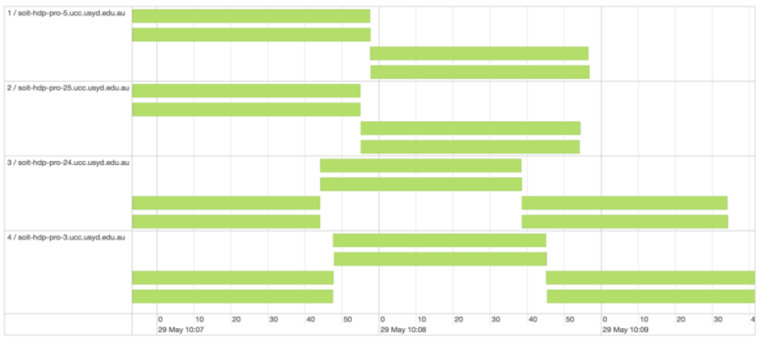


Figure 7. Event timeline with 4 executors and 2 cores for each executor. Four cores stay idle waiting for other four cores to finish their tasks and the total length of timeline is bottlenecked by the core run for longer time.

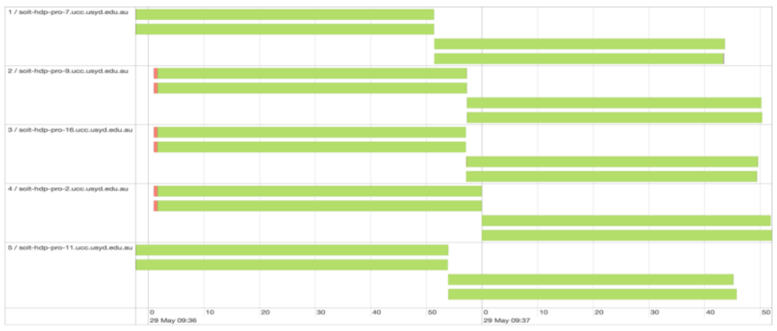


Figure 8. Event timeline with 5 executors and 2 cores for each executor. All cores finish their tasks at similar timestamp thus the whole timeline is shorter.

The table also shows the trend that the shuffle read and shuffle write increase with the increase of the number of the executors. The shuffle read measures the total read serialized data at all executors at the start of one stage while the shuffle write measures the total written serialized data at all executors before transmission between executors. Thus, the larger number of executors resulting in the larger size of data transmission in the shuffle between stages essentially produces the larger shuffle read and write. It also should be noticed that when dealing with dataframe, Spark would optimize the Query Plan by moving operations and determining where the shuffle is required though the Exchange process.

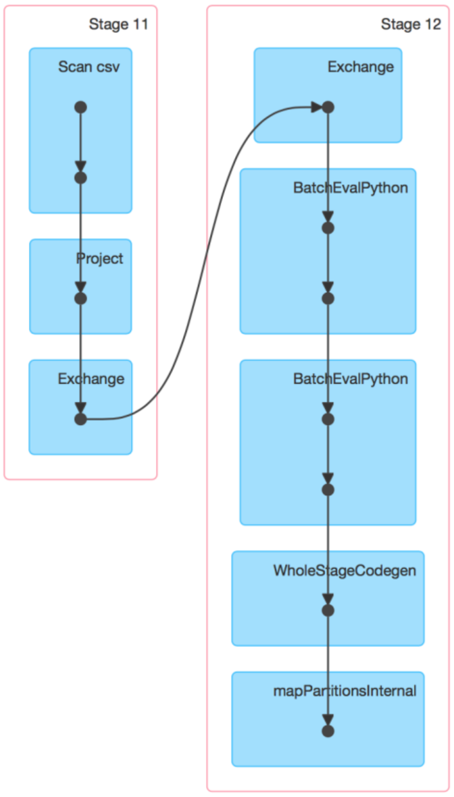
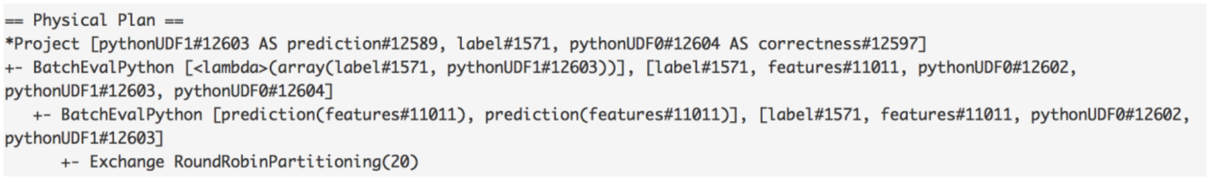


Figure 9. Stages for kNN prediction

The figure illustrated the two stages involved with kNN predictions, where the Stage 11 shows the procedure of dimensional reduction of test dataset which includes the data reading from csv file, project dataset to lower dimension using trained PCA and shuffle optimization with Exchange. During the stage switch, the dataset has been repartitioned to 20 partitions. In Stage 12, the dataset in each partition into two UDFs (i.e. one for kNN classification and the other one for generate result correctness for further evaluation). From the Physical Plan shown below, it could be seen that after partition to 20 partitions, Spark has called the prediction function twice and the lambda function once to complete the prediction.



```
== Physical Plan ==
*Project [pythonUDF1#12603 AS prediction#12589, label#1571, pythonUDF0#12604 AS correctness#12597]
+- BatchEvalPython [<lambda>(array(label#1571, pythonUDF1#12603))], [label#1571, features#11011, pythonUDF0#12602,
pythonUDF1#12603, pythonUDF0#12604]
   +- BatchEvalPython [prediction(features#11011), prediction(features#11011)], [label#1571, features#11011, pythonUDF0#12602,
pythonUDF1#12603]
      +- Exchange RoundRobinPartitioning(20)
```

Figure 10. Physical Plan of kNN

4 Spark Classifier Exploration

4.1 Algorithm Description

(1) Decision Tree Classifier

In the Decision Tree Classifier (DTC), a series of specially designed questions is generated based the features of the training data. Once the answer for current question is determined, a follow-up question would be generated until the final one about the class label is reached.

With the trained DTC, the classification would be straightforward. When predicting, the classification starts from the root node of the tree and forwards to either an internal node where a follow-up question pop-up or a leaf node where corresponding class label would be assigned.

In spark ml library, CART algorithm is employed that a binary tree grows with features and thresholds at each node with maximum information gains [2]. During the training, the decision tree would recursively partition the dataset to subsets that are as pure as possible for a particular label. The impurity at each node is calculated by Gini impurity function$D = \sum_i^C f_i(1 - f_i)$ as shown below by default:

$$G(T,s) = \frac{N_{left}}{N} D(T_{left}) + \frac{N_{right}}{N} D(T_{right})$$

Where N, Nleft, Nright represent the total, left and right node size respectively. C and fi represent the total number of unique labels and the frequency of label i respectively.

The main factors for the time complexity of DTC is the input size, which depends on numbers of instances and features. The former could be reduced by sampling, which is usually used in Random Forest, while dimensionality reduction methods like PCA would work to improve the latter.

(2) Multilayer Perceptron Classifier

Multilayer Perceptron Classifier (MLPC) contains multiple layers of nodes which is allocated based on the feedforward artificial neural network. Each layer in the network are fully connected to the next one and the first size of the first layer and the last layer are the input layer and the output layer respectively. All the nodes in the hidden layers and the output layer map their inputs into output through feeding a linear combination of the inputs and their corresponding node weights $\boldsymbol{w}$ and bias $\boldsymbol{b}$ into an activation function.

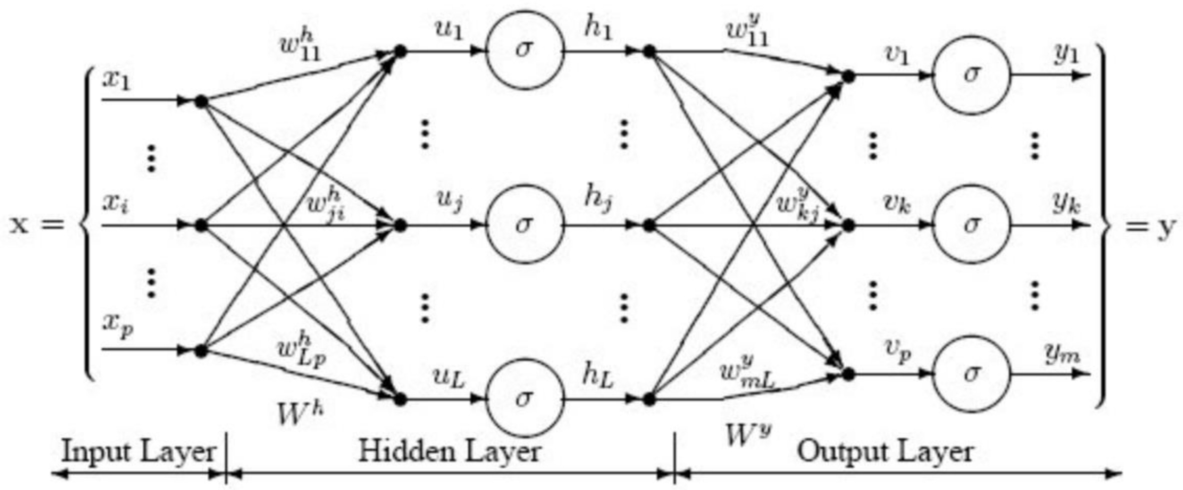


Figure 11 Typical structure of multilayer perceptron network [3]

As shown in the figure, the output from the first node in the first hidden layer h1 could be calculated as:

$$h1 = g(\sum_{k=1}^{p} \boldsymbol{w_{1k}^T} \, \boldsymbol{x_k} + b_1)$$

Where the activation function g is the sigmoid function by default:

$$g(x) = \frac{1}{1 + e^{-x}}$$

Consequently, the output y with K+1 layers could be expressed as:

$$y(\boldsymbol{x}) = g_K(\dots g_2\big(\boldsymbol{w_2^T} g_1\big(\boldsymbol{w_1^T x} + b_1\big) + b_2\big) \dots + b_K)$$

Whereas in the output layer the softmax function is used:

$$g_K = \frac{e^x}{\sum_{k=1}^{m} e^{x_k}}$$

In addition, MLPC in Spark uses the backpropagation to learn the model and the L-BFGS to optimize the loss function.

With such a lot hyperparameters, the time complexity of MLPC will be affected by many of them, for example, the number and size of hidden layers, max number of iterations and block size.

4.2 Data Pre-processing and Result Collection

The data is loaded and pre-processed with the same procedure as that in our kNN. The train and test data are separately read in as DataFrame and then assembled by VectorAssembler. All training data (60000 rows) are input into these two classifiers, and all testing data (10000 rows) transformed with the same pre-processing procedures would be used to test and compare the different models.

For Decision Tree Classifier, PCA with 20 and 200 as number of reduced dimensions is applied in the pre-processing stage, and also a classifier without PCA as baseline. For Multilayer Perceptron Classifier, input is the full feature matrix assembled with no feature selection method involved.

The prediction results are directly sent to MulticlassClassificationEvaluator, which is a well-defined evaluator in Spark ML library. Since accuracy is the only evaluating value concerned in this workload, the output will be the accuracy calculated by the evaluator and printed in the console.

4.3 Execution Statistics and Accuracy

All the experiments under this section utilize 5 executors and 2 executor-cores in SIT cluster environment.

(1) Decision Tree Classifier

As the training set is rather large, we choose 12 and 5 as the values of maxDepth and minInstancesPerNode instead of the default values in the model to get higher and comparable accuracies.

```
        dt = DecisionTreeClassifier(maxDepth=12, minInstancesPerNode = 5, \
                                    labelCol="label", featuresCol="features")
```

From the execution plan of Decision Tree below, it can be found that the first Codegen is for classification and the second is for evaluation. Decision Tree is actually to transform data then perform hash aggregate operations on them.

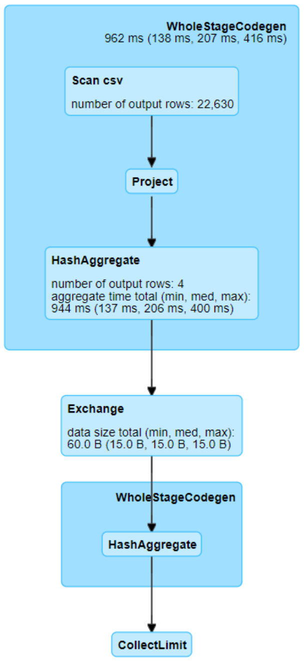

WholeStageCodegen
962 ms (138 ms, 207 ms, 416 ms)
Scan csv
number of output rows: 22,630
Project
HashAggregate
number of output rows: 4
aggregate time total (min, med, max):
944 ms (137 ms, 206 ms, 400 ms)
Exchange
data size total (min, med, max):
60.0 B (15.0 B, 15.0 B, 15.0 B)
WholeStageCodegen
HashAggregate
CollectLimit

Figure 12. Execution Plan of Decision Tree

Three numbers of feature dimensions are tested: 20, 200 and all features.

Table 3. Decision Trees with Different Dimensionalities

Num of Features	Accuracy	Execution Time	Jobs/ Tasks	Total Task Time	Input	Shuffle Read	Shuffle Write
784 (all)	87.3%	46 s	29/448	4.8 min	4.0 GB	350.2 MB	441.1 MB
200	82.2%	1.2 min	26/334	3.5 min	1.4 GB	121.1 MB	244.2 MB
20	82.7%	47 s	26/334	1.6 min	0.9 GB	2.6 KB	34.3 MB

It can be found that although the model with all features ranked the first in accuracy, its input and shuffle read/write are fairly large and the runtime is also longer than dimensionality-reduced models. In addition, the accuracy of 20 features is even higher than that of 200 features.

It might be concluded as in DTC for this dataset, the utilization of PCA could significantly reduce the runtime and I/O cost without much loss in accuracy. In the other word, Decision Tree Classifier could benefit a lot from feature selection methods like PCA. That is probably due to the influences of noisy features is more remarkable on the classifiers with simply hypothesis like Decision Tree.

(2) Multilayer Perceptron Classifier

The max iteration in multilayer perceptron is set to be 100 and the block size is 30. One fixed hidden layer with different sizes of 50, 70 and 100 are tested.

```
    perceptron = MultilayerPerceptronClassifier(maxIter=100, layers=layers, \
                                                blockSize=30, seed=1234)
```

Table 4. Multilayer Perceptron with Different Hidden Layer Sizes

Size	Accuracy	Execution Time	Jobs /Tasks	Task Time	Input	Shuffle Read	Shuffle Write
100	94.7%	1.8 min	108/1248	9.5 min	38.7 GB	419.2 MB	524.8 MB
70	93.9%	1.4 min	110/1272	8.2 min	39.5 GB	290.0 MB	363.4 MB
50	93.0%	1.5 min	112/1296	7.1 min	40.3 GB	205.9 MB	257.6 MB

As shown in table2, larger hidden layer size would result in a higher accuracy, longer runtime and larger shuffle read/write. So the selection of hidden layer size in Multilayer Perceptron is a trade-off between accuracy and efficiency.

However, it need to be further studied why the input size increases when hidden layer size decreases.

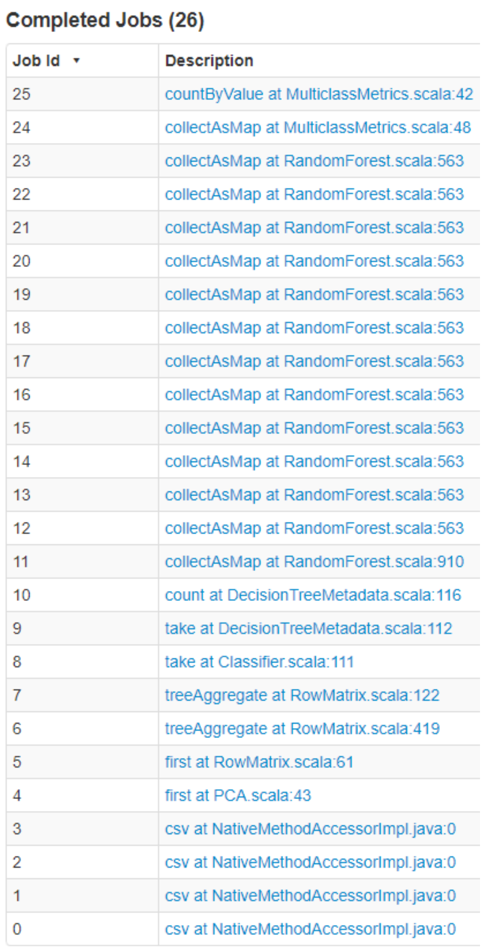
(3) Comparison

It could be seen from the comparison between table 1 and table 2 that the average accuracy of MLPC (~94%) is significantly higher than that of DT (~84%) for our hyperparameter setups. In contrast the execution time for DT (4.8 mins) is much shorter than that of MLPC (7.1 mins – 9.5 mins) when both classifiers using the full dimension of features. Furthermore, the I/O cost for DTC (4G) is much lower than that of MLPC (38.7 GB – 40.3 GB) excluding PCA dimension reduction.

4.4 Extensive Analysis

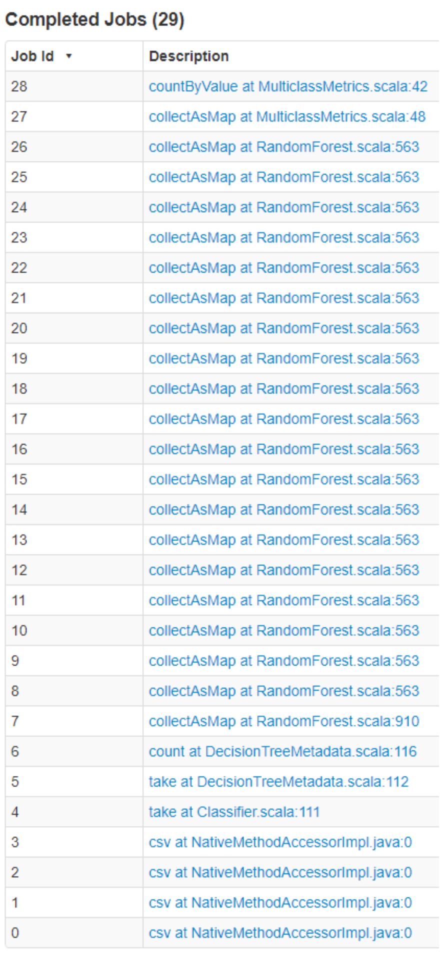
(1) Decision Tree Classifier

In the executions of Decision Tree, with such large number of training samples, the value of maxDepth seems to determine the number of jobs. For instance, with 200 features, there are 4 jobs for loading data, 4 jobs for PCA, 4 jobs for classifier setup, 12 jobs for tree calculation and 2 jobs for evaluation. The hypothesis also holds for other numbers of PCA output features as well as different maxDepth. However, if with all features in, there are 7 more tree calculations than maxDepth, as shown in the figure below.



Completed Jobs (26)

Job Id ▾	Description
25	countByValue at MulticlassMetrics.scala:42
24	collectAsMap at MulticlassMetrics.scala:48
23	collectAsMap at RandomForest.scala:563
22	collectAsMap at RandomForest.scala:563
21	collectAsMap at RandomForest.scala:563
20	collectAsMap at RandomForest.scala:563
19	collectAsMap at RandomForest.scala:563
18	collectAsMap at RandomForest.scala:563
17	collectAsMap at RandomForest.scala:563
16	collectAsMap at RandomForest.scala:563
15	collectAsMap at RandomForest.scala:563
14	collectAsMap at RandomForest.scala:563
13	collectAsMap at RandomForest.scala:563
12	collectAsMap at RandomForest.scala:563
11	collectAsMap at RandomForest.scala:910
10	count at DecisionTreeMetadata.scala:116
9	take at DecisionTreeMetadata.scala:112
8	take at Classifier.scala:111
7	treeAggregate at RowMatrix.scala:122
6	treeAggregate at RowMatrix.scala:419
5	first at RowMatrix.scala:61
4	first at PCA.scala:43
3	csv at NativeMethodAccessorImpl.java:0
2	csv at NativeMethodAccessorImpl.java:0
1	csv at NativeMethodAccessorImpl.java:0
0	csv at NativeMethodAccessorImpl.java:0



Completed Jobs (29)

Job Id ▾	Description
28	countByValue at MulticlassMetrics.scala:42
27	collectAsMap at MulticlassMetrics.scala:48
26	collectAsMap at RandomForest.scala:563
25	collectAsMap at RandomForest.scala:563
24	collectAsMap at RandomForest.scala:563
23	collectAsMap at RandomForest.scala:563
22	collectAsMap at RandomForest.scala:563
21	collectAsMap at RandomForest.scala:563
20	collectAsMap at RandomForest.scala:563
19	collectAsMap at RandomForest.scala:563
18	collectAsMap at RandomForest.scala:563
17	collectAsMap at RandomForest.scala:563
16	collectAsMap at RandomForest.scala:563
15	collectAsMap at RandomForest.scala:563
14	collectAsMap at RandomForest.scala:563
13	collectAsMap at RandomForest.scala:563
12	collectAsMap at RandomForest.scala:563
11	collectAsMap at RandomForest.scala:563
10	collectAsMap at RandomForest.scala:563
9	collectAsMap at RandomForest.scala:563
8	collectAsMap at RandomForest.scala:563
7	collectAsMap at RandomForest.scala:910
6	count at DecisionTreeMetadata.scala:116
5	take at DecisionTreeMetadata.scala:112
4	take at Classifier.scala:111
3	csv at NativeMethodAccessorImpl.java:0
2	csv at NativeMethodAccessorImpl.java:0
1	csv at NativeMethodAccessorImpl.java:0
0	csv at NativeMethodAccessorImpl.java:0

Figure 13. Job List of Two Decision Trees

The possible reason might be relatively large numbers of features (with all 784 features in this case) will have different classification split method, binary split instead of multiclass split for each node as it exceeds certain value. [2]

(2) Multilayer Perceptron Classifier

As aforementioned, the I/O cost increases with the decrease of the hidden layer size in MLPC. It is majorly from the larger number of epochs for classifier to optimize the loss function using L-BFGS algorithm with smaller layer size. It could be illustrated from the fact that the total number of treeAggragte at LBFGS jobs are 100, 102, 104 for MLPC with layer size of 100, 70, 50 respectively. However, due to larger size of the hidden layer (i.e. larger size of **w** at output of the first hidden layer), the execution time for L-BFGS optimization at each job is generally longer.

5 Bibliography

[1] http://spark.apache.org/docs/latest/tuning.html

[2] https://spark.apache.org/docs/latest/mllib-decision-tree.html

[3] https://spark.apache.org/docs/2.2.0/api/scala/index.html#org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator.html