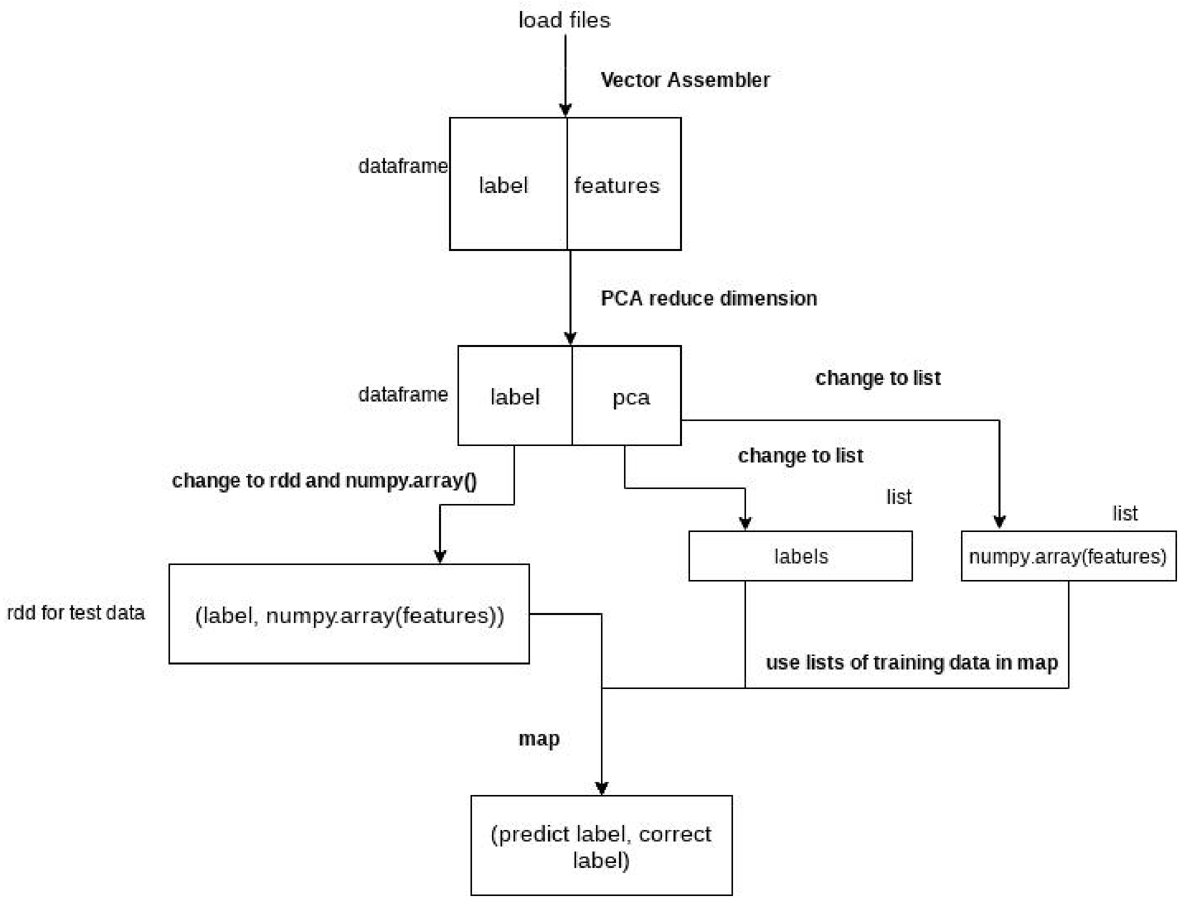
Table of Contents

Design



It first loads files from *hdfs://soit-hdp-pro-1.ucc.usyd.edu.au/share/MNIST*, then uses vector assembler to combine all feature columns to a single column called *features*. The work would be on label column (default name *_c0*) and features.

It uses PCA to reduce the dimension of features to *nPCADimension*, which is a hyper-parameter for both training data and test data. It changes training data and test data to **RDD (for parallelization)**, and changes the type of element in features from vector to **numpy.array() (using numpy.array to calculate distance is fast)**. It changes the type of training data labels and training data features to list for following calculation and **broadcasts** them **(broadcasts the lists, so that all the machines would cache a read only variable)**.

```
#Use PCA reduce dimension
pca = PCA(k=nPCADimension, inputCol="features", outputCol="pca")
train_model = pca.fit(train_vectors)
train_pca_result = train_model.transform(train_vectors).select(["_c0","pca"])
test_pca_result = train_model.transform(test_vectors).select(["_c0","pca"])
```

```
train_l = train_pca_result.select("_c0")
train_d = train_pca_result.select("pca")

#Change to numpy.array
def num(record):
  lab, vec = record
  return (np.array(vec) ,lab)
trainlabels_list = train_l.rdd.map(lambda x: x.asDict()["_c0"]).collect()
trainfeatures_list = train_d.rdd.map(lambda x: np.array(x.asDict()["pca"])).collect()
temp_test = test_pca_result.rdd.map(num).cache()

#broadcast training data
sc = spark.sparkContext
trainlabels = sc.broadcast(trainlabels_list)
trainfeatures = sc.broadcast(trainfeatures_list)
```

Then it uses *temp_test.map(knn)*, for each element *(numpy.array(features), label)* in test data, calculating the distance between it and all the 60,000 training data, finding *nKNNDimension* (a hyper-parameter) smallest distance and corresponding labels, and then it finds the label who has the largest amount among the *k* training data.

```
def knn(record):
  datatest, test_label = record
  #find all 60,000 distances
  distances1 = (trainfeatures.value-datatest)**2
  distSquareMatSums = distances1.sum(axis = 1)

  #find nKNNDimension smallest distances
  sorteIndices = np.argsort(distSquareMatSums)
  indices = sorteIndices[:nKNNDimension]

  #find corresponding labels and label with largest amount
  labels = {}
  for i in indices:
     if trainlabels.value[i] not in labels:
        labels[trainlabels.value[i]] = 0
     labels[trainlabels.value[i]] += 1
  key_max = max(labels.keys(), key=(lambda o: labels[o]))

  #(predict label, correct label)
  return (key_max,test_label)
```

After it gets all the 10,000 *(predict label, correct test label)* tuples, it uses this result to calculate accuracy, precision, recall, and f1 score.

Sample result

There were two output file generated: the *raw result file* and the *brief report file*.

The raw result file contains the detailed result for the 10, 000 times' predictions in a tuple format: *(prediction, gold result)*. Raw result file would be stored in the HDFS server and divided into multiple parts according to the number of cores in the experiment. Below is the first three lines of a sample raw result file (which represents three correct prediction in a row):

```
(7, 7)
(6, 6)
(2, 2)
… ...
```

The brief report file is the file we used for viewing concrete results and debugging. Below is the full content of a sample brief report file:

```
+---+        ### comment: debug - printing label samples
|_c0|
+---+
|  5|
|  0|
+---+
only showing top 2 rows

+--------------------+ ### comment: debug - printing PCA samples
|                 pca|
+--------------------+
|[880.731433034386...|
|[1768.51722024166...|
+--------------------+
only showing top 2 rows

start calculation
145.040019989    ### comment: net time
9741          ### comment: raw accuracy
statistics
0
precision: 97.39
recall: 99.18
f1: 98.28
1
```

```
precision: 96.83
recall: 99.74
f1: 98.26
2
precision: 98.62
recall: 97.00
f1: 97.80
3
precision: 97.12
recall: 96.93
f1: 97.03
4
precision: 98.44
recall: 96.33
f1: 97.38
5
precision: 97.62
recall: 96.52
f1: 97.07
6
precision: 98.03
recall: 98.54
f1: 98.28
7
precision: 97.16
recall: 96.40
f1: 96.78
8
precision: 97.82
recall: 96.71
f1: 97.26
9
precision: 95.30
recall: 96.43
f1: 95.86
```

Performance Analysis

Parameters to Tune

All the parameters we tuned in the performance experiments are listed in the following table, in which the important parameters are bold:

Name	**Type**	**Description**
--expname	string	The name used to identify each experiment.
--pcad	int	The dimension in PCA.
--knnd	int	The distance in KNN.
--num-executors	int	Number of executors.
--executor-cores	int	The number of cores to use in each executor.
--partition	int	The number of partitions. In the experiments introduced in this report, it is always set to the product of --num-executors and --executor-cores. We explicitly reserved a slot for this parameter only for further experiment purposes, in which we may test the partition number's impact on execution performance.

The parameters were set with the following script (local_submit.sh):

```
#!/bin/bash

spark-submit \
        --master yarn \
        --deploy-mode client \
        --executor-memory 2G \
        --num-executors $4 \
        --executor-cores $5\
        stage1knn.py --expname $1 --pcad $2 --knnd $3 --partition $6
```

Performance to Measure

We focused on two aspects of performance: *prediction performance* and *execution performance*.

For prediction, we measured: *overall prediction accuracy* and *each label's precision, recall and f1-score*.

For execution, we measured: *execution time* and *total I/O cost*. To measure the performance of our KNN implementation, we described execution time as *net time* (the time costed by KNN) and *total time* (the time displayed in Spark History Server). Total I/O cost is calculated as the summary of *Shuffle Read* and *Shuffle Write* displayed in Spark History Server.

Experiment Design

To measure the performance under different parameter settings, we designed the experiment as below:

For dimension in PCA, we set two levels: 60 and 100.

For distance in KNN, we set two levels: 6 and 10.

For parallelism, we set three levels for number of executors and number of cores in each executor: 2-2 (4 cores in total), 5-4 (20 cores in total), 8-4 (32 cores in total).

Overall, we had twelve (2 * 2 * 3) independent rounds of experiments, which is listed in the table below:

Serial	***Raw serial***	**--pcad**	**--knnd**	**--num-executors**	**--executor-cores**
01	*01*	60	6	2	2
02	*03*	60	10	2	2
03	*02*	100	6	2	2
04	*04*	100	10	2	2
05	*05*	60	6	5	4
06	*07*	60	10	5	4
07	*06*	100	6	5	4
08	*08*	100	10	5	4
09	*09*	60	6	8	4
10	*11*	60	10	8	4
11	*10*	100	6	8	4
12	*12*	100	10	8	4

Experiment Results: Prediction

Without consideration of execution performance, we just focused on four rounds of experiments (serial 01-04, 05-08, and 09-12 had identical prediction performance) to see --pcad and --knnd's influence on prediction performance.

Overall accuracy

Serial	**--pcad**	**--knnd**	**Overall accuracy**
01	60	6	97.41%
02	60	10	97.35%
03	100	6	97.26%
04	100	10	97.04%

We can see that the overall accuracy was satisfying in all four cases. Although the differences were quite slight, one conclusion we are certain about is the less the dimension in PCA, the more accuracy prediction we would get. However, this conclusion does not necessarily suggest that we shall use extremely low PCA dimensions such like 2 or 3.

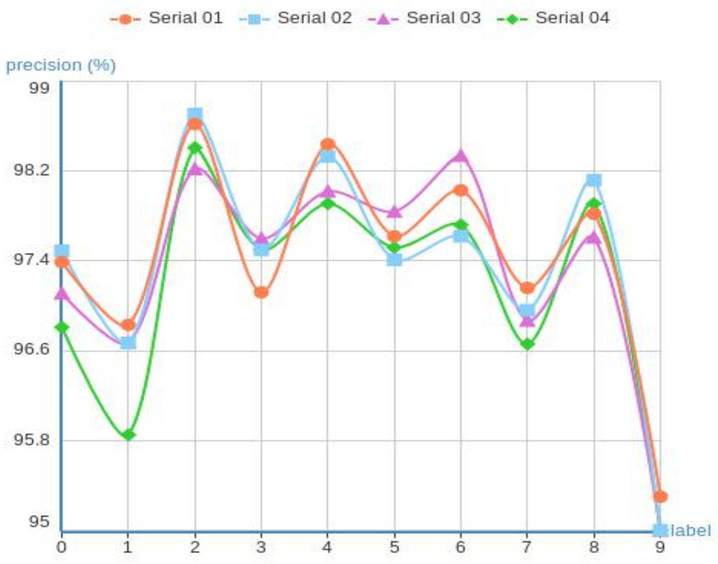
Each label's precision, recall and f1-score

Label	**Precision (01 - 02 - 03 - 04)**	**Recall (01 - 02 - 03 - 04)**	**F1-score (01 - 02 - 03 - 04)**
0	97.39% - 97.49% - 97.11% - 96.81%	99.18% - 99.08% - 99.29% - 99.18%	98.28% - 98.28% - 98.18% - 97.98%
1	96.83% - 96.67% - 96.67% - 95.85%	99.74% - 99.65% - 99.65% - 99.65%	98.26% - 98.13% - 98.13% - 97.71%
2	98.62% - 98.71% - 98.22% - 98.41%	97.00% - 96.71% - 96.51% - 95.93%	97.80% - 97.70% - 97.36% - 97.15%
3	97.12% - 97.50% - 97.60% - 97.50%	96.93% - 96.73% - 96.44% - 96.63%	97.03% - 97.12% - 97.01% - 97.07%
4	98.44% - 98.33% - 98.02% - 97.91%	96.33% - 96.23% - 95.82% - 95.52%	97.38% - 97.27% - 96.91% - 96.70%
5	97.62% - 97.41% - 97.84% - 97.52%	96.52% - 97.09% - 96.64% - 96.86%	97.07% - 97.25% - 97.24% - 97.19%
6	98.03% - 97.62% - 98.34% - 97.72%	98.54% - 98.64% - 98.75% - 98.43%	98.28% - 98.13% - 98.54% - 98.08%
7	97.16% - 96.96% - 96.87% - 96.66%	96.40% - 96.30% - 96.21% - 95.82%	96.78% - 96.63% - 96.53% - 96.24%
8	97.82% - 98.12% - 97.61% - 97.91%	96.71% - 96.20% - 96.41% - 96.00%	97.26% - 97.15% - 97.00% - 96.94%

9	95.30% - 94.94% - 94.66% - 94.54%	96.43% - 96.63% - 96.63% - 96.13%	95.86% - 95.78% - 95.64% - 95.33%

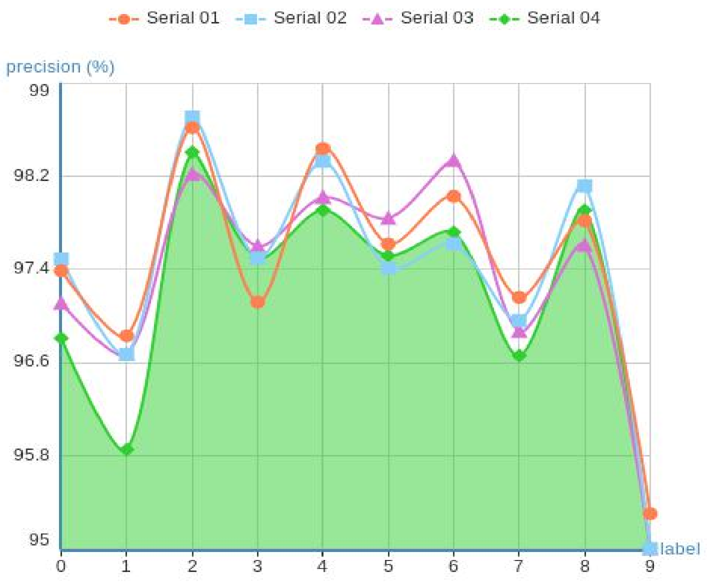
Overall, the statistics demonstrated that our KNN implementation performed very well regarding prediction.

In the case of hand-written recognition, precision is considered more important than recall and f1-score. Therefore, we analyzed the precision of the four rounds of experiments:



We represented each round of experiments (serial 01-04) with separate curves.

From the graph we concluded that none of the four rounds had outperformed others regarding precision. However, we did found that round 04 (--pcad 100, --knnd 10) performed the worst:

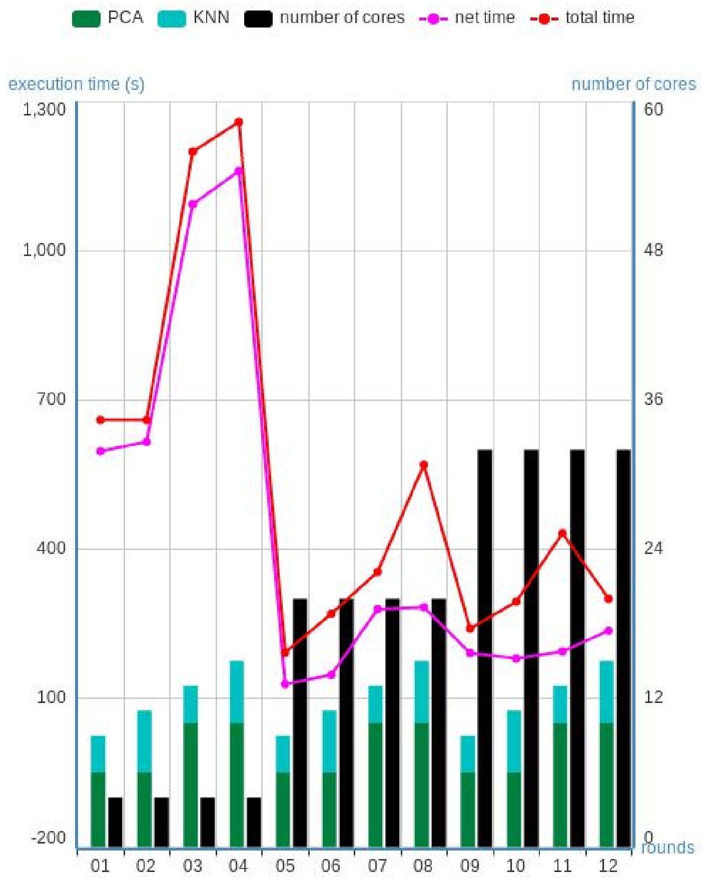


To conclude, we confidently believe that we can finally achieve a dominating prediction performance if we further tune the --pcad and --knnd towards the smaller value direction.

Experiment Results: Execution

Serial	***Raw serial***	**Parameter brief**	**Net time (s)**	**Total time (s)**	**Total I/O (Shuffle Read + Shuffle Write) (MB)**
01	*01*	60-6-2-2	597	660	7.4 (2.5 + 4.9)
02	*03*	60-10-2-2	616	660	7.4 (2.5 + 4.9)
03	*02*	100-6-2-2	1094	1200	12.3 (4.1 + 8.2)
04	*04*	100-10-2-2	1161	1260	12.3 (4.1 + 8.2)
05	*05*	60-6-5-4	128	192	42.7 (19.0 + 23.7)
06	*07*	60-10-5-4	147	270	42.7 (19.0 + 23.7)
07	*06*	100-6-5-4	279	354	48.5 (21.5 + 27.0)
08	*08*	100-10-5-4	283	570	33.7 (6.7 + 27.0)
09	*09*	60-6-8-4	191	240	33.7 (4.3 + 29.4)
10	*11*	60-10-8-4	180	294	9.4 (4.4 + 5.0)
11	*10*	100-6-8-4	194	432	55.9 (23.3 + 32.6)
12	*12*	100-10-8-4	236	300	39.8 (7.2 + 32.6)

Execution time



We represented the computing difficulty with one combine-colored pillar (green and cyan): the summary of PCA's dimension and KNN's distance. The higher the two values were, the more difficult the computing was.

We represented the computing power with the black pillar: number of cores. The number of cores is the product of the number of executors and the number of cores in each executor.

We represented the execution time with two separate curves (red for total time, and pink for net time).

According to our experiment design, the net time (time costed by our KNN implementation) should always be lower than the total time, which was proved by the above graph. Particularly, in experiment round 08 and 11, the total time was abnormally much higher than the net time. We assumed that was because of the cluster's extreme busy state in the experiment duration.

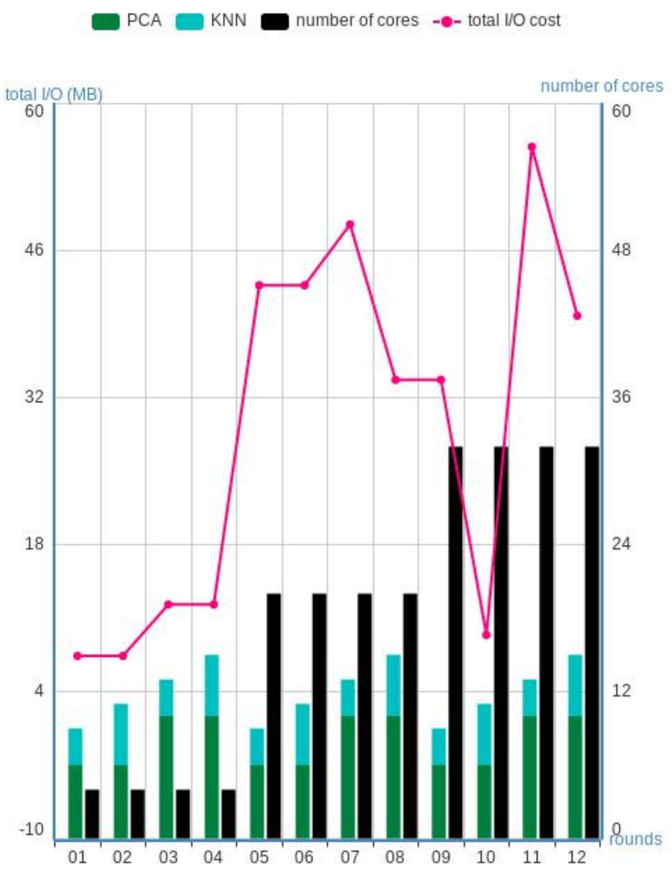
With the same amount of allocated computing power (round 01-04, 05-08, 09-12), the execution time was increased when the computing difficulty was increased.

When the allocated computing power was increased, the execution time was significantly decreased, on which the computing difficulty somehow would not have much impact.

Additionally, it is worth mentioning that, the performance improvement when switching from 20-cores to 32-cores was not as significant as when switching from 4-cores to 20-cores. This phenomenon demonstrated that we can easily achieve great optimization in performance once we move the computing from local to cluster. However, it would cost much more computing resources than a reasonable amount, to get the same level of performance optimization when we are already enjoying the cluster environment.

In conclusion, the two curves' shape matched our expectation.

Total I/O Cost



We represented the total I/O cost with the pink curve.

Due to the randomness of data locality, this curve was not able to yield a clear pattern. Nevertheless, we can safely conclude that, with the number of cores increased, the amount of I/O was increased.

Since the number of partitions was set to the number of cores in our experiments by default, the relationship between the number of partitions and the total I/O cost seemed interesting. However, tuning the number of partitions would not only possibly (due to its randomness nature) influence the total I/O cost, but also definitely influence the execution time, which would bring extra unnecessarily complexity to this report. Therefore, this issue is considered as future study.

Spark Classifier Exploration

Brief description of Decision Tree Model

The decision tree model is a greedy algorithm that executes a heuristic binary breakdown of the feature space. The tree predicts the same label for each bottommost (leaf) division. Each division is chosen greedily by selecting the best split from a set of possible splits, in order to increase the information gain at a tree node.

Investigation

The decision tree model makes prediction based on the parameter of dimension size, however it would predict less accurate if the size of the dimension is decreased. Please refer to table 1 in decision tree performance analysis. The initial test case accepts five in dimension size, and the algorithm predicted fifty seven percent, when the dimension is increased in second test case, it slightly increases the percentage to sixty five. This could be an indication that the algorithm is likely to be suitable with large data sets because it needs more data to produce better output.

Preparation of Data

The application takes the data directly from MNIST database by the following process:

- ⇨ Importing all the necessaries Spark Libraries in python
- ⇨ Using argument parser to allow application for accepting dimension parameter
- ⇨ Initiate the Spark Session to allow spark starting
- ⇨ Point to data source within the MNIST database, we are only using train and test datasets in the form of CSV files.
- ⇨ Read the data from MNIST database by using Spark CSV reader
- ⇨ Assemble both train and test data into a single column named “feature”. The output will be displayed in table with two columns and defaulted to both “_c0” and “features”
- ⇨ Using Reduction algorithm such as PCA to reduce the features dimension which is the hyper parameter
- ⇨ Showing the vectors of both train and test file

Collection of the Output

- ⇨ Perform the prediction by producing the data into two columns which is both default column and the features
- ⇨ Check the accuracy by looping within the range of 10000 columns and return the value in the form of percentage.

Brief description of Logistic Regression Model

Logistic regression is a popular method to predict a categoric response. It is a special case of Generalized Linear models that predicts the probability of the outcomes. The spark.ml logistic regression can be utilized to predict a binary result by using binomial logistic regression, or it can be utilized to predict a multi-class result by using multinomial logistic regression. By using family parameter to choose between these two algorithms, or leave it unset and Spark will infer the correct variant.

Investigation

The logistic regression model also takes parameters such as dimension, number of cores, and number of executors when running it on cluster node. The initial test case was conducted on a cluster node is indicated in table 2 in logistic regression performance analysis. Based on this test case it can be concluded that the Logistic regression model tend to predict accurately as the size of dimension is increased. The initial was conducted on cluster node with dimension size of five(please refer to table 2 in Logistic regression performance analysis), the algorithm predicts sixty seven percent, and when it increases to one hundred of dimension size in second test case, it almost predicts accurately with the score of ninety percent. By looking at the two test cases, it can be said that the algorithm is likely works well with smaller datasets and able to provide accurate precision.

Preparation of Data

The preparation of data is similar to the previous classifier. Please refer to the first classifier.

Collection of the Output

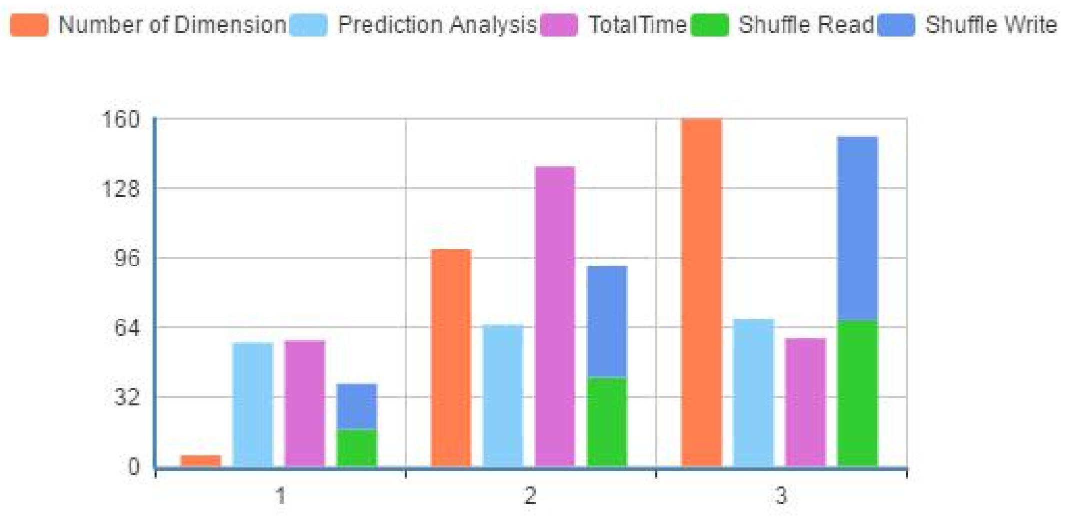
The collection of the output remains the same as the first classifier.

Performance Analysis

Decision Tree

The model is tested in the cluster by accepting two parameters which are number of executions and number of cores required to run the code. Below is the statistical analysis of the Decision Tree model performance based on number of cores, executors and also the size of the dimension when testing in cluster node.

Test #	Total Cores (5 executors x 4 Cores)	Number of Dimension	Prediction Analysis	Total time in seconds	Total I/O in MB(Shuffle read + Shuffle write)
1	20	5	57%	58	38 (16.9 + 21.1)
2	20	100	65%	138	92.1 (40.8 + 51.3)
3	20	748	67.9%	59	151.8 (67.2 + 84.6)

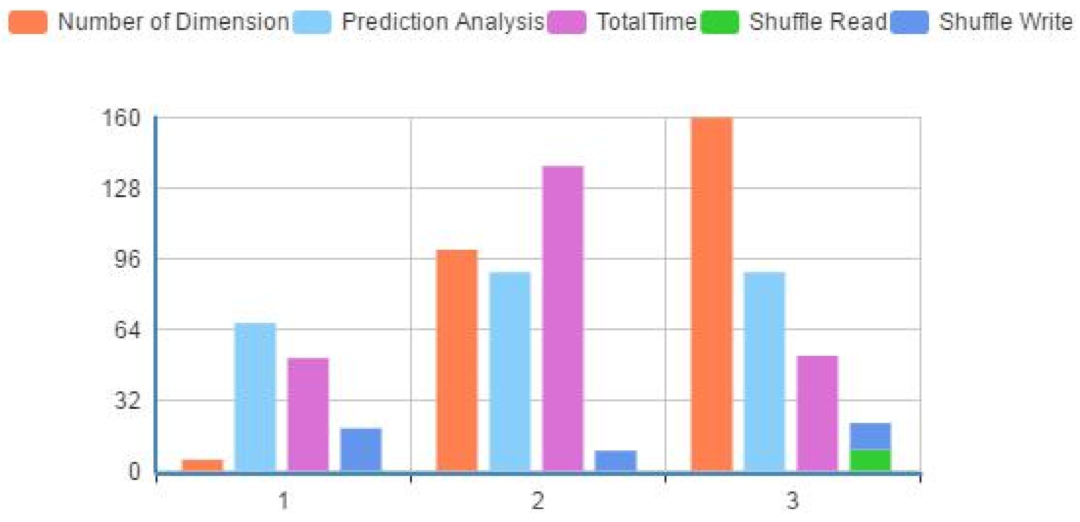


The graph shows that as the size of dimension is increased, the time is also increased when using PCA algorithm (Dimensionality reduction). In addition, the prediction is also slightly increased. On the other hand the total time execution is dramatically decreased when executing raw (unreduced) dimension in cluster and the accuracy of prediction is also slightly increased.

Logistic Regression

The same set up as decision tree is established for logistic regression algorithm when testing it in cluster node. The graph below indicates the performance analysis for this model.

Test #	Total Cores (5 executors x 4 Cores)	Number of Dimension	Prediction Analysis	Total time in seconds	Total I/O in MB(Shuffle read + Shuffle write)
1	20	5	67%	51 seconds	19.34 (0.14 + 19.2)
2	20	100	90 %	138 seconds	9.2 (0 + 9.2)
3	20	748	90 %	52 seconds	21.7 (9.6 + 12.1)



The table shows that using PCA model will increase the accuracy of prediction as the size of dimension is increased. In addition, the time will also be increased as the dimension is increased. On the contrary the time will be dramatically decreased when running raw (unreduced) dimension in cluster node while the accuracy of prediction almost accurate.

Interesting Finding

Both Algorithms work well with the data set provided, however interestingly Logistic Regression model predicts accurately as opposed to Decision Tree model. This is shown in table 1 and table 2 as part of the performance analysis. The impact of the dimension size for decision tree model is not significant to the predictions it produces for both test case one and case two. There is only slight increase to the accuracy of prediction when the dimension is increased. On the other hand, Logistic regression makes almost an accurate prediction regardless of small size of dimension. This is shown in table table 2, it predicts up to 90% when the dimension is increased from 100. In terms of performance, both algorithms execute the task based on the size of the dimension. If the dimension is increased then time it requires to execute the task is increased. Finally, when making comparison with raw (unreduced) dimension, it appears that both classifiers have dramatic increase in performance because it only requires 55 seconds in average to finish all tasks.