COMP5349 – Cloud Computing
Assignment 2: Spark Machine Learning Application

Group: SIT 114 – 7

Introduction

This assignment requires analyses and performance tuning of a Spark application. This report contains three stages. All stages use the MNIST data set of handwritten digits. Stage one describes the implementation of a kNN classifier, stage two assess the performance of the classifier and stage three investigates two classifiers from the pyspark machine learning library, which are Decision Tree and Multi Layer Perceptron.

Stage One: Design

The kNN classifier is designed to distribute the calculation of the prediction of the 10,000 test images. This is carried out by using the *withColumn* method of the distributed dataframe with a user defined function to find the k nearest neighbours for each test set image. This code can be seen in Code snippet 1.

Code snippet 1: Parallel computation on test set

```
# for each sample of the test data, distribute the kNN prediction calculation
udf_func = udf(lambda row: kNN(row, bc_X, bc_y, k), IntegerType())
X = X.withColumn('prediction', udf_func(X.features))
```

Each node calculates the k nearest neighbours for the test set samples in the partitions assigned to that node. To facilitate this calculation the training set is broadcast to each node. This can be done because the training set fits in the memory on a single node. Broadcasting the training set means that it is sent once to each node and doesn't need to be re-sent with each task thereby reducing computation time. The code to broadcast the training set is shown in Code snippet 2.

Code snippet 2: Broadcast training set

```
# Broadcast the training data so that it will be available
#  for each distributed computation of the nearest neighbours
#  to the test set vectors.

bc_X = sc.broadcast(self.X)
bc_y = sc.broadcast(self.y)
```

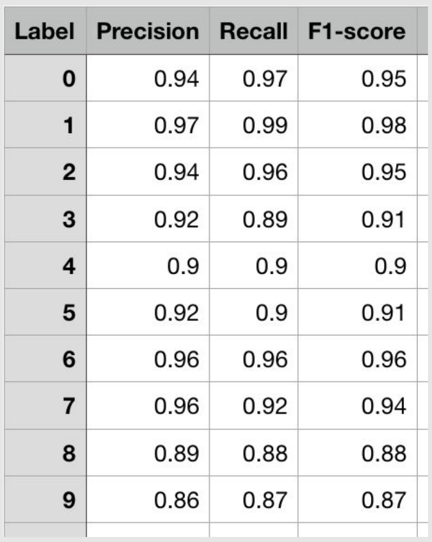
The user defined function applied in Code snippet 1 is shown in Code snippet 3. This function is computed on each node for different partitions of the test set. The function converts the test sample to an array and uses numpy to calculate the distance from the test sample to every training vector. Based on those distances, the indexes of the k nearest neighbours can be found using numpy's argpartition method. These indexes are used to find the labels of the k nearest neighbours and the mode of those classes is returned as the prediction.

Code snippet 3: User defined kNN function

```
# convert vector to array
sample_vector = sample_vector.toArray()

# calculate distance to training vectors
distances = np.linalg.norm(sample_vector-bc_X.value, axis=1)

# find indexes of nearest neighbours
idx = np.argpartition(distances, k)

# get the most common label of the k nearest neighbours
prediction = int(mode(bc_y.value[idx[:k]]))
```

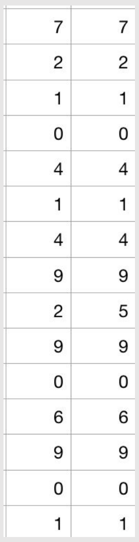
Example results from running PCA and the kNN classifier on the 10,000 test images is shown in Figure 1. The number of dimensions were reduced to 10 using PCA, followed by kNN classification using the 3 nearest neighbours. It can be seen that the classifier has the most difficulty with classifying images of eights and nines.

Figure 1: Example output



Label	Precision	Recall	F1-score
0	0.94	0.97	0.95
1	0.97	0.99	0.98
2	0.94	0.96	0.95
3	0.92	0.89	0.91
4	0.9	0.9	0.9
5	0.92	0.9	0.91
6	0.96	0.96	0.96
7	0.96	0.92	0.94
8	0.89	0.88	0.88
9	0.86	0.87	0.87

Figure 2: Performance Metrics



7	7
2	2
1	1
0	0
4	4
1	1
4	4
9	9
2	5
9	9
0	0
6	6
9	9
0	0
1	1

Figure 3: Predictions and ground truth

Stage Two: Performance Analysis

The performance of the kNN classifier is investigated with 12 different combinations of hyperparameters. These parameters are the PCA dimensionality reduction, d, the number of nearest neighbours, k, and the number of executors. The number of executor cores is set to 2 for each of the combinations.

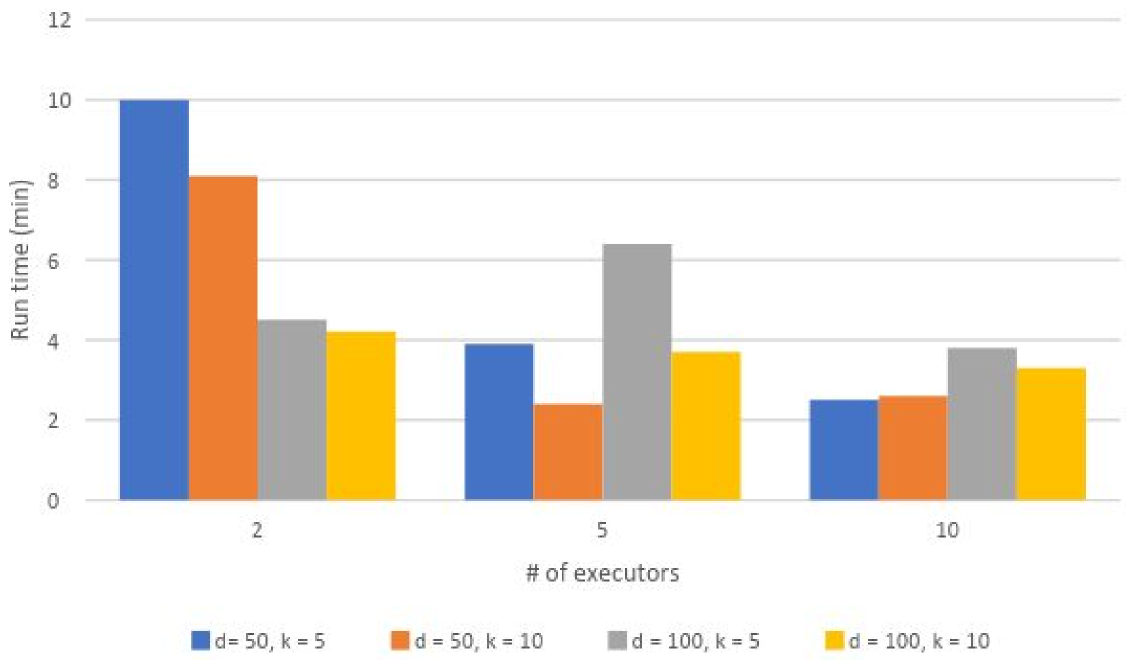
The 12 combinations were run on the Sydney University Spark cluster. The results are shown in Table 1. Screenshots of the Spark history server as shown in Appendix A.

Table 1: Performance Statistics for kNN classifier

Run	d	k	# of executors	Run time (min)	F1-score	Accuracy	Input (MB)	Shuffle Write (MB)
1	50	5	2	10	0.97	0.97	457.5	0
2	50	5	5	3.9	0.97	0.97	457.6	10.2
3	50	5	10	2.3	0.97	0.97	457.7	18.8
4	50	10	2	8.1	0.97	0.97	457.5	0
5	50	10	5	2.4	0.97	0.97	393.5	10.2
6	50	10	10	2.6	0.97	0.97	457.7	18.8
7	100	5	2	4.5	0.97	0.97	457.5	0
8	100	5	5	6.4	0.97	0.97	457.5	0
9	100	5	10	3.8	0.97	0.97	457.5	18.8
10	100	10	2	4.2	0.97	0.97	457.5	0
11	100	10	5	3.7	0.97	0.97	457.7	10.2
12	100	10	10	3.3	0.97	0.97	382.1	18.8

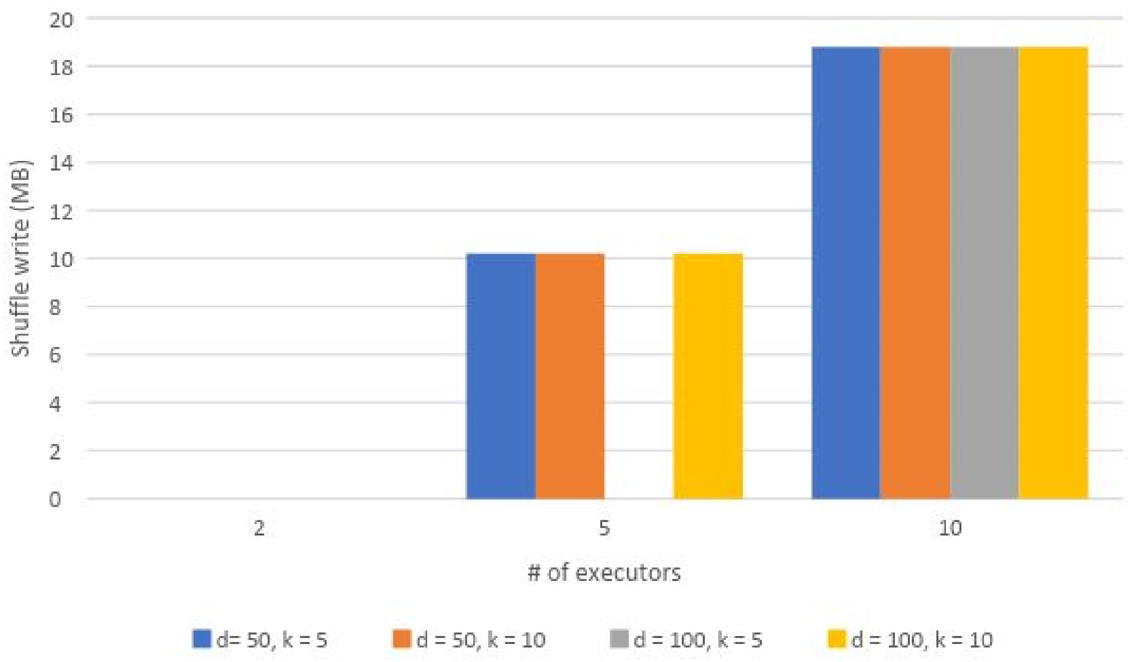
The run time for each of the 12 combinations is shown in Figure 4. Generally the run time reduces as the number of executors is increased from 2 up to 10. The reduction in run time is due to the increase in parallel computation. At some point we expect that the run time will not decrease if the number of executors is increased because it will no longer be beneficial to further distribute the calculation at the expense of needing to shuffle the data between more executors.

Figure 4: Spark Application Run Time



The shuffle write for each of the 12 combinations is shown in Figure 5. With more executors there is more data that is shuffled because each executor has less of the distributed data.

Figure 5: Spark Application Shuffle



The accuracy and F1-scores vary slightly between the 12 combinations but in Table 1 these values are shown to be the same when rounded to two decimal places. We believe this to be the case because the range of values for PCA dimensionality reduction and number of nearest neighbours is not wide enough to shown a significant difference in classifier accuracy and F1-score.

Stage Three: Spark Classifier Exploration

Applicable to both classifiers:

Data Preparation

The train and test csv is read in as a train dataframe and test dataframe. VectorAssembler() is used to convert the 748 features into a sparse vector feature representation.

Output Collection

The prediction results are written out as csv files, with the first column is the correct labels, and the second column is the predicted labels after transformation. P,R,F-scores and accuracy are printed out in terminal.

Execution Environment

Executor-cores is 2 and num-executors is 2.

Decision Tree Classifier

Decision Tree Classifier is widely used in Machine Learning Classification as well as Regression problems due to its easiness to understand, explore and implement. It predicts the class of a new coming data by creating a model that learns decision rules from a set of training data features.

We examine Decision Tree Classifier by changing the dimension of feature vector. The first job use the raw pixel value feature vector, without any dimensionality reduction process. The second, third and fourth jobs use PCA dimensionality reduction algorithm as a preprocessing step, with number of reduced features are 25, 50 and 100 respectively. After completing the PCA feature reduction, this reduced feature vector is used to train Decision Tree model, which is then used to predict the class of each new data in the test set.

The code to perform Decision Tree classifier without PCA reduction is shown in Code Snippet 4.

```
dt = DecisionTreeClassifier(impurity='entropy', maxDepth= 30, maxBins=32)
model = dt.fit(train_vectors)
predictions = model.transform(test_vectors)
```

A pipeline is created to perform PCA dimensionality reduction and Decision Tree transformation, which can be seen in Code Snippet 5.

```
    # Create pipeline and apply Decision Tree Classifier with PCA reduction
    pca = PCA(k= PCA_k, inputCol="features", outputCol="pca")
    decision_tree = DecisionTreeClassifier(featuresCol = "pca", impurity='gini', maxDepth= 30, maxBins=32)
    pipeline = Pipeline(stages = [pca, decision_tree])

    # Apply pipeline to training data & make prediction on test data
    model = pipeline.fit(train_vectors)
    predictions = model.transform(test_vectors)
```

The performance metric for each dimension value is tabulated in table 2.

	No PCA			k = 25			k = 50			k = 100		
Accuracy	0.88			0.86			0.85			0.85		
Training Time	73			37			69			65		
Execution Time	1.9			1.1			1.6			1.6		
Input	8GB			1.2GB			1.4GB			1.7GB		
Label	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
0	0.95	0.91	0.93	0.92	0.9	0.91	0.91	0.9	0.91	0.91	0.9	0.91
1	0.96	0.97	0.96	0.96	0.96	0.96	0.97	0.96	0.96	0.97	0.96	0.97
2	0.85	0.87	0.86	0.88	0.88	0.87	0.85	0.85	0.85	0.84	0.85	0.84
3	0.84	0.85	0.84	0.84	0.84	0.84	0.84	0.83	0.83	0.82	0.81	0.82
4	0.88	0.88	0.88	0.83	0.83	0.82	0.82	0.83	0.82	0.82	0.82	0.82
5	0.85	0.84	0.84	0.8	0.8	0.8	0.79	0.78	0.79	0.79	0.78	0.78
6	0.9	0.91	0.91	0.9	0.9	0.89	0.88	0.87	0.87	0.89	0.88	0.88
7	0.9	0.89	0.9	0.87	0.87	0.86	0.86	0.87	0.86	0.85	0.87	0.86
8	0.82	0.83	0.82	0.79	0.79	0.79	0.78	0.79	0.79	0.78	0.79	0.78
9	0.86	0.84	0.85	0.79	0.79	0.79	0.79	0.79	0.79	0.79	0.8	0.79
Avr	0.88	0.88	0.88	0.86	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.84

Table 2: Performance Statistics for Decision Tree Classifier

As shown in the table, the highest accuracy (88%) is achieved when no dimensionality feature reduction is used, which is reasonable because no feature information is lost. However, it is not efficient in terms of execution time and I/O cost. The I/O cost without PCA is 6 times higher than using PCA with k = 25. The execution time without PCA is also higher than the time needed to run both PCA reduction and Decision Tree classifier.

Comparing the performance between three PCA dimensional values, we can see there is no improvement in terms of accuracy and other metrics (precision, recall, f1-score) if the dimension is increased from 25 to 100. On the contrary, these metrics are even slightly reduced. Furthermore, the execution time and I/O cost are also increased with higher dimensional values, due to the fact that time and space complexity of Decision Tree classifier depend on the dimension of feature vector. Therefore, to choose between three values, 25, 50 and 100 for PCA dimensionality feature reduction, we can conclude that the value of 25 is sufficient for this Decision Tree classifier.

We further experiment different PCA dimensional values ranging from 5 to 100. The results can be seen in Figure 6.

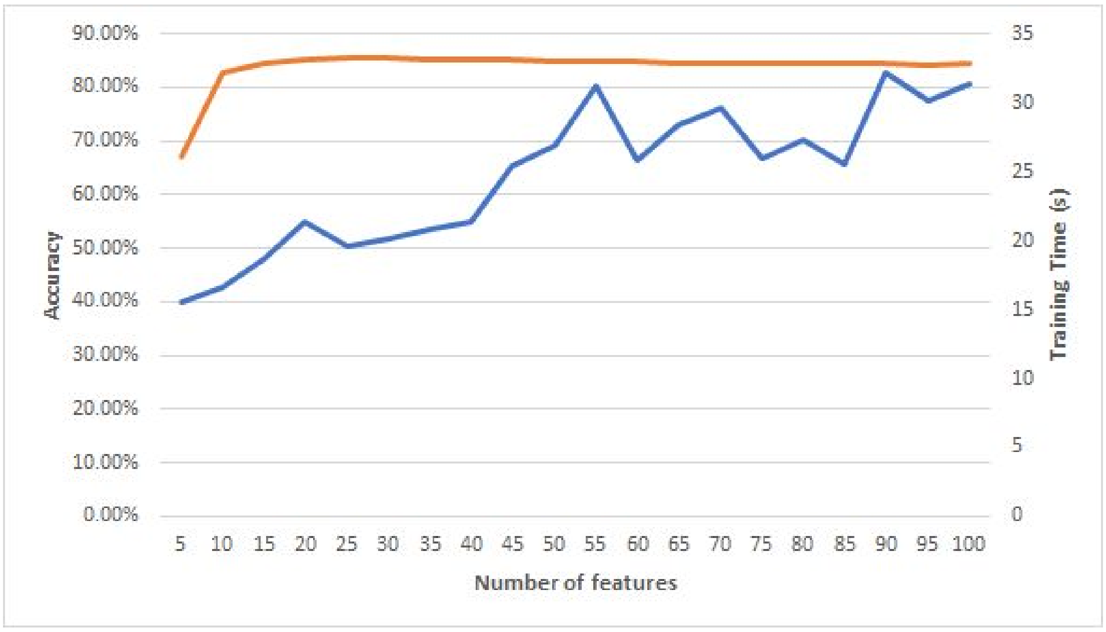


Figure 6: Accuracy and training time on different PCA dimensional values

From the figure, the accuracy is increased when the number of features increases from 5 to 25 and starts to reduce afterwards. Thus, we can conclude that higher PCA dimensional features does not always guarantee better accuracy performance. We suspect that for each dataset, there may exist a turning point on which the accuracy starts to reduce or stops getting better results with the increase of PCA dimensional values. Furthermore, in general, it takes more time and I/O cost to train higher dimensional features.

In conclusion, depending on the dataset and priorities, we should experiment various PCA dimensional values in order to obtain an efficient value in terms of execution time, accuracy and I/O cost.

Multilayer Perceptron (MLP) Classifier

The MLP Classifier is based on an artificial feed-forward neural network model. It maps input features to output labels using patterns learned through one or more hidden layers. For example, the input layer analyses the feature set of each image, then the hidden layer learns distinctive patterns of the image. Finally at the output layer stage, it classifies the image by label based on the patterns learned. Here we explore the effect of block size on the classifier's performance with block sizes of 30MB, 15MB and 1MB.

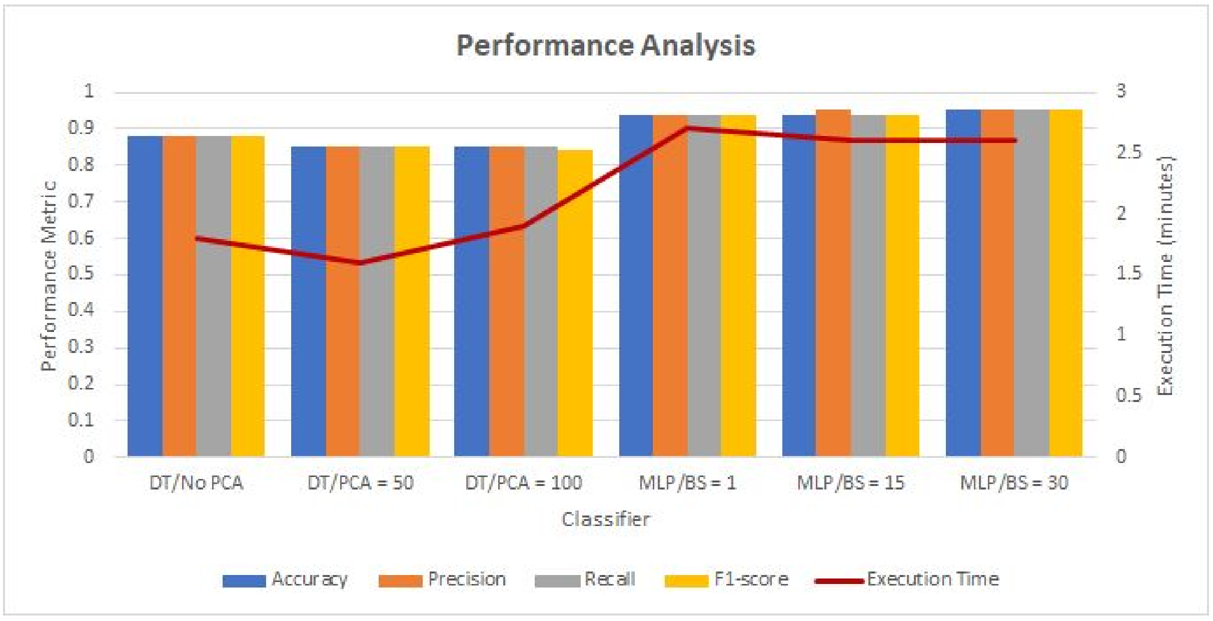
In total there are three layers involved:
1 - Input layer of size 784 as there are 784 different features
2 - A hidden layer of fixed size 100, (100 hidden neurons)
3 - Output layer of size 10 as there are 10 different labels (numbers 0 to 9)
Max iterations is set to 100.

	block size = 1			block size = 15			block size = 30		
Accuracy			0.94			0.94			0.95
Running time			2.7mins			2.6mins			2.6mins
Input			39.7GB			37.8GB			37.7GB
Label	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
0	0.98	0.95	0.97	0.99	0.96	0.97	0.98	0.96	0.97
1	0.98	0.97	0.98	0.98	0.97	0.98	0.98	0.98	0.98
2	0.94	0.94	0.94	0.95	0.94	0.94	0.94	0.95	0.95
3	0.93	0.92	0.92	0.93	0.92	0.93	0.94	0.93	0.93
4	0.94	0.94	0.94	0.94	0.94	0.94	0.95	0.95	0.95
5	0.92	0.93	0.93	0.92	0.94	0.93	0.91	0.95	0.93
6	0.96	0.95	0.95	0.96	0.95	0.96	0.96	0.95	0.96
7	0.94	0.95	0.94	0.94	0.95	0.94	0.94	0.95	0.94
8	0.92	0.94	0.93	0.93	0.93	0.93	0.94	0.93	0.93
9	0.92	0.93	0.92	0.92	0.93	0.92	0.93	0.94	0.93
Avr	0.94	0.94	0.94	0.95	0.94	0.94	0.95	0.95	0.95

Table 2: Performance Statistics for Multilayer Perceptron Classifier

As shown in Table 2, there are small variabilities in performance by label when block size is changed, however the average scores suggest no significant differences. According to the Spark documentation, this parameter is used to stack input data into matrices to "speed up computation". However no significant improvements in speed is obtained with a larger block size. At this point of exploration, the results suggest block size as a minor parameter for performance tuning for our given dataset and settings. Further testing should be done with different parameter tuning such as maxIterations, number of hidden layers and number of neurons within each layer.

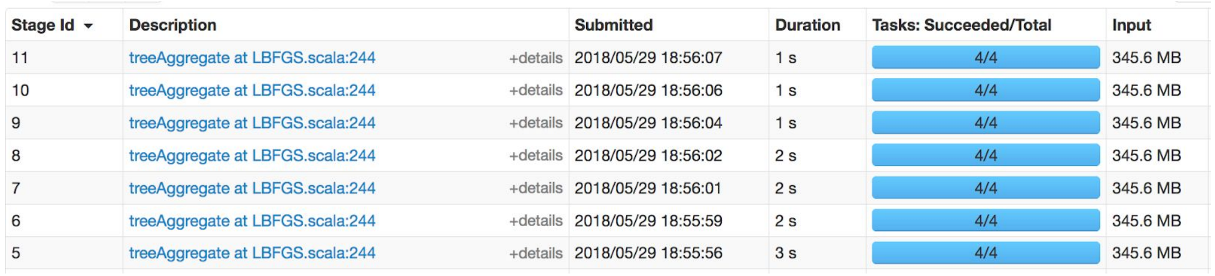
Performance Comparison



We compare the Decision Tree and MLP job with the highest accuracy, where MLP job = block size of 30 and Decision Tree job = no PCA reduction.

Trade-off between accuracy and execution speed is apparent, where the Decision Tree job achieved 0.88 overall accuracy which was still lower than all MLP jobs (block size = 1 & 15: acc = 0.94, block size = 30: acc = 0.95). We found overall, Decision Tree classifier involves less calculations and memory which reflects the quicker execution speed of 1.8mins compared to MLP (2.6 mins, no PCA). We suggest Decision Tree optimises "overall performance" when considering I/O cost and time. MLP outperforms Decision Tree in accuracy (0.95) at the expense of a slower execution speed due to more calculations and more memory-intensive processes.

Decision Tree also had an advantage of lower I/O cost (8.6GB) compared to MLP (37.7GB). The large I/O cost of MLP is hypothesised to be due to the large dimensionality (784 features) and its effect on training with L-BFGS and its associated history maintenance until convergence is achieved or reaches maxIter = 100. This process makes the bulk of the I/O cost. An increase in I/O is linked to larger dimensionality and larger number of iterations.



Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total	Input
11	treeAggregate at LBFGS.scala:244	+details	2018/05/29 18:56:07	1 s	4/4	345.6 MB
10	treeAggregate at LBFGS.scala:244	+details	2018/05/29 18:56:06	1 s	4/4	345.6 MB
9	treeAggregate at LBFGS.scala:244	+details	2018/05/29 18:56:04	1 s	4/4	345.6 MB
8	treeAggregate at LBFGS.scala:244	+details	2018/05/29 18:56:02	2 s	4/4	345.6 MB
7	treeAggregate at LBFGS.scala:244	+details	2018/05/29 18:56:01	2 s	4/4	345.6 MB
6	treeAggregate at LBFGS.scala:244	+details	2018/05/29 18:55:59	2 s	4/4	345.6 MB
5	treeAggregate at LBFGS.scala:244	+details	2018/05/29 18:55:56	3 s	4/4	345.6 MB

Table : Sample of treeAggregate at LBFGS stages. The number of LBFGS stages = maxIterations

<u>General Observations</u>

A key observation we had was the effect of busy servers on the execution speed of our applications. During busy periods, there were larger delays as a result of having to wait for available executors. This is a factor to consider when interpreting run time results.
In conclusion, we found KNN classifier of stage one to have the best "overall performance" due to its highest accuracy and lowest I/O cost for our dataset and classification problem.

This analysis shows that when designing Spark applications it is important to consider the amount of parallelisation possible and to tune both the application design and the number of executors. Proper tuning can provide great improvements in performance.