



COMP 5349 Cloud Computing

GROUP MEMBERS

UZAIR MAJID (STUDENT ID: - 470480768)

AHMED SABEEH SYED (STUDENT ID: - 460498652)

SHAKTI MALHOTRA (STUDENT ID: - 450611276)



Table of Contents

Introduction

Stage 1

KNN Classifier

- a) Introduction
- b) Implementation
- c) Execution and Machine Learning Stages

Stage 2

Execution and Performance Analysis

- a) Introduction
- b) Performance Analysis

Stage 3

Naïve Bayes Classifier

- a) Introduction
- b) Implementation
- c) Classifier

Multi-Layer Perceptron Classifier

- a) Introduction
- b) Implementation
- c) Execution Summary
- d) Evaluation

Analysis and comparison between Naïve Bayes and Multilayer Preceptor Spark ML algorithms

- a) Quality Metrics
- b) Performance metrics

REFERENCES

APPENDIX

Stage 1

Introduction

KNN is a non-parametric lazy learning algorithm has been evolved for classification and regression. In this terminology representation of different cases in dependent on the input consists of the k closest training examples in the data space. The algorithm truly dependent on the fact of observation weather the classification has been used or regression has been considered.

- In *k-NN classification*, the output is depicted as a class membership. An object is classified depending on the neighbors' majority votes, where the assignment of object to the particular class which is most common among its Knearest neighbors (k is a small positive integer). If $k = 1$, then the object is allocated to the class of its single nearest neighbor.
- In *k-NN regression*, the output achieved defines the object's property value. This value is mapped as the average of the values of its k nearest neighbors

Implementation

As per our assignment requirement we implemented KNN classification algorithm using spark framework & libraries.

We implemented Spark ML pipeline framework, SPARK SQL, Dataset & RDD APIs to implement KNN. In Following we have summarized important areas of our system's implementation

1- Pipeline Framework & Algorithm parameters

In our driver application we first initialize the program with configurable element coming from configuration namely K, D input files output & execution mode (cluster or yarn) .

Our driver reads the input data using parser function to parse the given file generally in this format (Row ID, Label, feature Vector) Then we reduced dimensions of given data using PCA and prepare the code for next step in this step each row of input data is assigned a row id bot for training & test this is used for referencing and further collections.

After PCA KNN is initialized with parameter K and placed in Pipeline

```
// Set KNN in pipeline
Pipeline pipeline = new Pipeline().setStages(new PipelineStage[] { new KNN().setK(KNearestNeighbours) });
```

PCA & KNN was not processed in same pipelines because of the nonlinear nature of processing ,this is shown in following code snippet :-

```
// read training & test dataset
Dataset<Row> trainDS = readTrainingDataSet();
Dataset<Row> testDS = readTestDataSet();

// use PCA to reduce dimention
PCA pcaTrain = new PCA().setInputCol("training_features").setOutputCol("pcaFeatures").setK(dimension);
PCA pcaTest = new PCA().setInputCol("test_features").setOutputCol("pcaFeatures").setK(dimension);

// reduce dimention dataset
trainDS = pcaTrain.fit(trainDS).transform(trainDS);
testDS = pcaTest.fit(testDS).transform(testDS);
log.info("PCA transformation completed ");
//.....
//.....
log.info("KNN ## fitting training data then learning labels for for test data.... ");
Dataset<Row> result = pipeline.fit(trainDS).transform(testDS); //KNN in thiis pipe line
```

Then fit & transformed pipeline containing KNN using Train DS & test data set respectively

```
// fit training data & learn on test data
log.info("KNN ## fitting training data then learning labels for for test data.... ");
Dataset<Row> result = pipeline.fit(trainDS).transform(testDS);
```

KNN Class extending spark. This class extends abstract class “org.apache.spark.ml.Estimator” override implementing methods to make KNN work in pipeline . The method fit creates KNN Model that’s our trained model and is described next.

KNNModel is the transformer returned from KNN. This class extends “org.apache.spark.ml.Model” that extends Transformer & Pipeline Stage from the API and implement other interfaces.

Methods were overridden from this class to implement & comply with pipeline framework.

```
KNNModel validateAndTrain(Dataset<?> trainDS) {
    // check for required columns
    HashSet<String> cols = new HashSet<>();
    cols.addAll(Arrays.asList(trainDS.columns()));

    if (!(cols.contains("training_id") && cols.contains("training_label"))) {
        // required columns missing throw exception
        throw new IllegalArgumentException(
            "Required columns missing : supplied columns in training data set ::" + cols.toString());
    }

    if (cols.contains(PCA_FEAT_COL_NAME) && cols.contains(TRAINING_FEAT_COL_NAME)) {
        trainDS.drop(TRAINING_FEAT_COL_NAME); // replace features with pcafeat
        trainDS.withColumnRenamed(PCA_FEAT_COL_NAME, TRAINING_FEAT_COL_NAME);
    }
    this.trainingDS = (Dataset<Row>) trainDS;
    return this;
}
```

How KNN Model working when the fit method called on this class by framework this class takes and validates training dataset as show in following code snippet.

Execution and Machine Learning Stages

Cartesian Product using partition by test_row_id

First step in transformation we partitioned test Dataset by test_id because next stage of processing requires test_id to be cross join Id and grouped & used frequently so we wanted it to be in partition for as less shuffling as possible. Currently no custom partitioning supported in dataset API (only for RDD is available) we experience some data skewing because of we created far more partition that available workers to then available executors

```
/**
 * Cartesian strategy
 *
 * @param testDS
 * @return
 */
private Dataset<Row> calculateCartesianOptimized(Dataset<?> testDS) {
    SparkSession spark = SparkSession.builder().getOrCreate();
    SparkConf conf = spark.sparkContext().env().conf();
    int exec = conf.getInt("spark.executor.num", 8);
    int cores = conf.getInt("spark.executor.cores", 5);
    testDS = testDS.repartition(exec * cores * 20, testDS.col("test_id"));
    testDS.cache();
    testDS.take(1); // force caching
    Dataset<Row> returnDS = testDS.crossJoin(trainingDS);
    testDS.unpersist(); // release cache to release disk/memory locks |
    return returnDS;
}
```

and cores combined to make sure maximum parallelism among nodes and workers have enough load. We also observed some skewed data that because of hashing scheme used by spark to partition. As per our observation good parallelism was achieved during Cartesian phase & next stages very few partition were skewed but were able to recover in reasonable time there is a potential for further fine tuning using lowered level RDD Api's and custom partitioning.

Distance Calculation on features vectors – UserDefinedFunction for Spark SQL. We have defined a User defined function to calculate distance between two vector columns in Cartesian row as showing in snippet .

```
// register UDF
spark.udf().register("calcVectorDistance", new CalcVectorDistanceUDF(), DataTypes.DoubleType);
calculateCartesianOptimized(testDS).createOrReplaceTempView("CartesianData");

// calcVectorDistance(..) is custom function
Dataset<Row> tmp = spark.sql(
    "select test_id, training_id, training_label, calcVectorDistance(test_features, training_features) "
    + "as distance from CartesianData");
```

Learning Label from K Nearest Neighbors by Vote/frequency

User defined Aggregate Function:- A user defined aggregate function was implementing in spark API by name of FindKNearestNeighboursLabelByFrequency this function was used as spark SQL later

```
// register UDF & UDAF
spark.udf().register("findKNearestNeighboursLabelByFrequency", new FindKNearestNeighboursLabelByFrequency(k));
Dataset<Row> result = spark.sql(
    "select test_id, findKNearestNeighboursLabelByFrequency(distance, training_label) as prediction FROM DistanceCalculated"
```

Major Performance improvement in FindKNearestNeighboursLabelByFrequency

During initial development we were using Spark SQL ORDER BY GROUP BY features in our analysis at that time we realized that the query is not optimized and there is no select Top N function available in spark so we enhanced our Aggregate function by implementing buffer in tree map structure that can have maximum of top K neighbors or elements using distance as key. This scales well in distributed fashion and reduced the unnecessary sorting of

all column (row count of larger table for each row) this improved the efficiency and overhead on following code snippets showcase this :-

```

    public void update(MutableAggregationBuffer buffer, Row input) {
        .....
        .....
    }

    /**
     * trim buffer to keep top N values
     */
    int size = internalBuffer.size();
    if (size > k) {
        // find & keep N only
        int rem = size - k;
        for (int x = size; x > k; x--) {
            internalBuffer.remove(internalBuffer.lastKey()); // trim the tree to keep top K or smallest K values
        }
    }
}

```

Merging two buffers in aggregate function

In final stage of this function we had tree of K nearest neighbor we then used hash map to find the frequency for each label

```

    int maxFreq = 0;
    for (Entry<Double, Integer> entry : c.entrySet()) {
        int freq = entry.getValue();
        if (freq > maxFreq) {
            maxFreq = freq;
            frequentLabel = entry.getKey();
        }
    }
    //log.info("computed prediction... " + frequentLabel);
    return frequentLabel;
}

```

```

@Override
public void merge(MutableAggregationBuffer buff1, Row buff2) {
    TreeMap<Double, Double> internalBuffer = new TreeMap<>();
    // update in internal buffer
    internalBuffer.putAll((Map<Double, Double>) buff1.<Double, Double>getJavaMap(0));
    internalBuffer.putAll((Map<Double, Double>) buff2.<Double, Double>getJavaMap(0));

    // trim buffer to top N only this works best in distribuited environment
    int bufferSize = internalBuffer.size();
    if (bufferSize > k) {
        for (int x = bufferSize; x > k; x--) {
            internalBuffer.remove(internalBuffer.lastKey()); // trim the tree to keep top
        }
    }
    //log.debug("merging...");
    buff1.update(0, internalBuffer);
}
}

```

Instrumentation Using Metrics API by Spark

We develop a class to record metrics from our processing in class Instrumentation that class also writes output to HDFS if configured.

Stage 2

Introduction

In this stage of work, we have analyzed different combination and set of hyper parameters which consists of d dimensions and K nearest neighbors. By using these parameters, we have executed 12 combinations by adding different environment properties such as number of executions and execution cores with these parameters. These combinations have allowed us to analyze the variance based on two types of metrics such as quality analysis: - In quality analysis we have considered the metrics values such as F1-score, recall, precision accuracy per each combination. Secondly, another important analysis is based on the performance of this application. In performance analysis we have used the changing values of the the num-exec and exec cores with different set values to analyze the performance of our algorithm and to check and highlight the maximum parallelism.

The combinations for both of our analysis along with the data, can be seen in the table below :

D & K combination	num exec	exec cores	execution time	Shuffle Read	Shuffle Write
51 , 6	5	3	3.2	17.6	19.6
51 , 6	7	3	1.4	12.8	17.8
51 , 6	9	4	2.1	16.8	20
51 , 9	5	3	10	1.3	20.4
51 , 9	7	3	2.3	21.7	20.4
51 , 9	9	4	2.9	17.4	20.4
95 , 6	5	3	5.4	7.4	21.1
95 ,6	7	3	1.8	17.8	21.1
95 ,6	9	4	7.6	1.1	21.7
95 ,9	5	3	2.5	21.1	21.1
95 ,9	7	3	2	21.7	21.7
95 ,9	9	4	7.3	17.4	21.2




The quality metrics were not included due to space constraints and are depicted in the graphs below.

The table shows that we have used two pairs of d and k with 4 combinations for these. Apart from this, different execution considerations and allowed the total combinations to 12. This raw data is further used in various plotted graphs to analyze the performance metrics. The aim of these executions was to depict the maximum parallelism in our application. After running these combinations, we came to the conclusion that our application is achieving the **maximum parallelism** up to 30 plus cores. The parallelism with can be seen in the image taken from one of our execution with the execution of 9 cores and 4 virtual cores.

Comp-5349 Cloud Computing

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
driver	172.16.176.243:43277	Active	4	67.3 KB / 384.1 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump
1	soit-hdp-pro-18.ucc.usyd.edu.au:36143	Active	3	65.3 KB / 956.6 MB	0.0 B	4	4	0	81	85	56 s (5 s)	136.9 MB	83 KB	1 MB	stdout stderr	Thread Dump
2	soit-hdp-pro-17.ucc.usyd.edu.au:34401	Active	3	65.3 KB / 956.6 MB	0.0 B	4	4	0	75	79	1.0 min (6 s)	125.9 MB	59.4 KB	1 MB	stdout stderr	Thread Dump
3	soit-hdp-pro-24.ucc.usyd.edu.au:36790	Active	3	65.3 KB / 956.6 MB	0.0 B	4	4	0	94	98	1.0 min (5 s)	160.9 MB	122 KB	1.1 MB	stdout stderr	Thread Dump
4	soit-hdp-pro-13.ucc.usyd.edu.au:39020	Active	4	67.3 KB / 956.6 MB	0.0 B	4	4	0	168	172	1.4 min (5 s)	283.9 MB	2.2 MB	6.2 MB	stdout stderr	Thread Dump
5	soit-hdp-pro-9.ucc.usyd.edu.au:50866	Active	3	65.3 KB / 956.6 MB	0.0 B	4	4	0	237	241	58 s (2 s)	417.7 MB	2.6 MB	1.8 MB	stdout stderr	Thread Dump
6	soit-hdp-pro-12.ucc.usyd.edu.au:53159	Active	3	65.3 KB / 956.6 MB	0.0 B	4	4	0	246	250	57 s (2 s)	441.2 MB	429.1 KB	1.1 MB	stdout stderr	Thread Dump
7	soit-hdp-pro-25.ucc.usyd.edu.au:53512	Active	3	65.3 KB / 956.6 MB	0.0 B	4	4	0	263	267	57 s (3 s)	468.8 MB	352.1 KB	1.1 MB	stdout stderr	Thread Dump
8	soit-hdp-pro-28.ucc.usyd.edu.au:53030	Active	3	65.3 KB / 956.6 MB	0.0 B	4	4	0	93	97	60 s (4 s)	159 MB	40.3 KB	1.1 MB	stdout stderr	Thread Dump
9	soit-hdp-pro-3.ucc.usyd.edu.au:50686	Active	3	65.3 KB / 956.6 MB	0.0 B	4	4	0	96	100	60 s (6 s)	166.2 MB	67.4 KB	1 MB	stdout stderr	Thread Dump

This image highlights the usage of all the assigned cores for our tasks hence depicting the perfect parallel application. To support our assumption, the next image shows usage of more than 20 cores (--num exec) but the amount of cores is not using more than 30 virtual cores although it has been assigned more containers which are sitting idle.

172.16.64.62:39570	Active	6	85 KB / 384.1 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
soit-hdp-pro-24.ucc.usyd.edu.au:35902	Active	2	31.2 KB / 2.1 GB	0.0 B	6	1	0	0	1	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
soit-hdp-pro-17.ucc.usyd.edu.au:38114	Active	0	0.0 B / 2.1 GB	0.0 B	6	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
soit-hdp-pro-28.ucc.usyd.edu.au:52153	Active	2	31.2 KB / 2.1 GB	0.0 B	6	1	0	0	1	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
soit-hdp-pro-25.ucc.usyd.edu.au:51435	Active	0	0.0 B / 2.1 GB	0.0 B	6	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
soit-hdp-pro-3.ucc.usyd.edu.au:58586	Active	0	0.0 B / 2.1 GB	0.0 B	6	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
soit-hdp-pro-9.ucc.usyd.edu.au:43919	Active	0	0.0 B / 2.1 GB	0.0 B	6	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
soit-hdp-pro-22.ucc.usyd.edu.au:34997	Active	6	85 KB / 2.1 GB	0.0 B	6	6	0	227	233	1.3 min (7 s)	291.5 MB	0.0 B	10.1 MB	stdout stderr	Thread Dump
soit-hdp-pro-14.ucc.usyd.edu.au:58454	Active	2	43.1 KB / 2.1 GB	0.0 B	6	6	0	226	232	1.0 min (2 s)	408.9 MB	0.0 B	2.5 MB	stdout stderr	Thread Dump
soit-hdp-pro-23.ucc.usyd.edu.au:54239	Active	0	0.0 B / 2.1 GB	0.0 B	6	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
soit-hdp-pro-13.ucc.usyd.edu.au:48352	Active	2	43.1 KB / 2.1 GB	0.0 B	6	6	0	253	259	1.1 min (2 s)	458.5 MB	0.0 B	2.5 MB	stdout stderr	Thread Dump
soit-hdp-pro-4.ucc.usyd.edu.au:59457	Active	0	0.0 B / 2.1 GB	0.0 B	6	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
soit-hdp-pro-15.ucc.usyd.edu.au:39550	Active	0	0.0 B / 2.1 GB	0.0 B	6	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
-D0000-78ff04...csv	...	Train-6000.csv	...		Test-1000.csv	...		Test-1000-data.csv	...		Test-1000-label.csv	...		Show All	

Performance Analysis:

To interpret the performance in detail, we used 2 major types of metrics such as execution time of the application and the IO costs. Further details are explained in the respective sections:

➤ Execution Time:

The above data from the table was constructed thoughtfully to indicate a clear pattern of time. The graph is as follows:



Series 1 : 5,3 (num exec , exec cores) Series 2 : 7,3 (num exec , exec cores)

Series 3 : 9,4 (num exec , exec cores)

The graph shows 3 types of num exec and exec cores and highlights the execution time for each of the D and K combination for these execution properties. The observations show that for all the combinations of D and K, which are based of 51 & 95 dimensions and 6 & 9 neighbors, The environment with 7 containers and 3 virtual cores is taking the least amount of time with compared to the other values . While this value contradicts from our parallel environment specs but there are certain things that need to be considered here. firstly, the containers/executor scheduling costs highly varied due to heavy load on the cluster at all times. Secondly, the more containers are added the more network costs and IO costs tend to increase due to which the time of the application could be adversely affected. Hence, adding more containers is providing more parallelism but the costs mentioned are also increasing the time 9 num exec and 4 exec cores.

➤ **I/O costs:**

We are considering the I/O costs based on the shuffle read and shuffle parameters because reading and writing are certainly an input and output operations. These costs are depicted in the graph below which highlights the shuffle read and write operations for all combinations of d and k(x axis) under various

execution parameters.



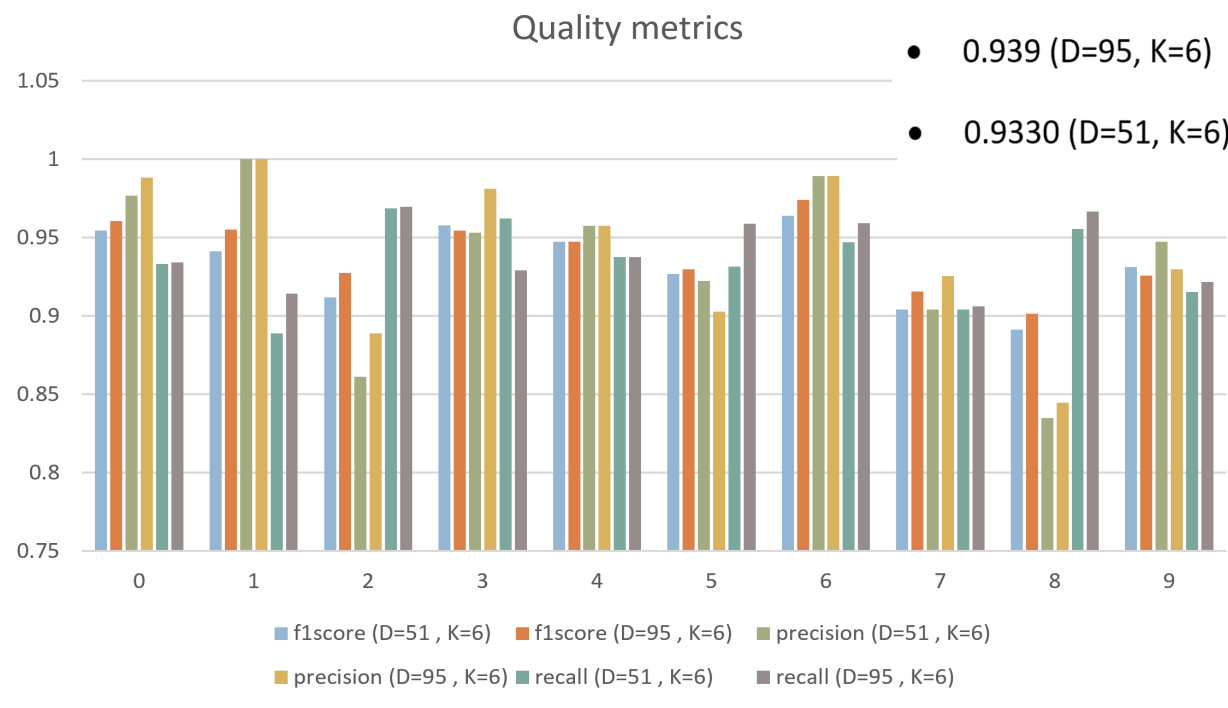
The IO factor with KNN classifier was really hard to depict due to the variance in the resources in the university cluster. Some assumptions however indicate that the growing number of dimensions along with the number of executors is increasing the IO of application. This can be seen with the 9,4 executors that as the dimensions and neighbors are increased a slight difference in the read and write operations is felt.

Quality Analysis:

For the analysis of the quality of the classifier we measured various measures such as accuracy, f1-score, precision and recall. Although we calculated these stats for all our combinations but to here we combine the metrics based on two such combinations of D and K as seen in the graph below:

The graph shows the metrics for all the labels i.e. from 0 to 9 based on two different hyper parameters of D and K i.e. 51, 6 and 95, 6. To better measure the quality we selected same number of neighbors and focused on increasing the dimensions. It can be seen from the graph that as we increase the neighbors all quality metrics tend to increase. The dots on top right give the accuracy for the 2 combinations and the accuracy also tends to increase with the increase in dimensions. The other sets which have not been shown here but have been a part of our executions also show a similar pattern which relates to the fact that with the increase in nearest neighbors

and feature dimensions the prediction tends to be more accurate.



Stage 3

Naïve Bayes Classifier

Introduction

A Naïve Bayes Classifier is a supervised machine-learning algorithm that uses the Bayes' Theorem, that specifies that features are statically independent. The theorem signifies on the naïve assumption that input variables are independent of each other, which means there is no way to know anything about other variables when a given which means it is not possible to know about other variables with a variable available.

It basically relies on the Bayes' Theorem, which is established on the basis of conditional probability or it can be simply explained as the likelihood of event (X) will happen as given that another event (Y) has already happened. Significantly, the theorem predicts and updates the hypothesis each time new evidence is introduced. The equation depicts the Bayes' Theorem in the language of probability.

$$P(A | B) = \frac{P(B | A)P(A)}{P(B)}$$

- "P" is the symbol to denote probability.
- $P(A | B)$ = The probability of event A (hypothesis) occurring given that B (evidence) has occurred.
- $P(B | A)$ = The probability of the event B (evidence) occurring given that A (hypothesis) has occurred.
- $P(A)$ = The probability of event B (hypothesis) occurring.
- $P(B)$ = The probability of event A (evidence) occurring.

Implementation

Data Processing:

In this phase of project, we imported the data in spark environment using Spark Session builder. We split the dataset into categories where first column is label and rest of the column is features. Then in next step we used Vector Assembler which is a transformer that combines a given list of columns into single vector column. The next step for PCA reduction was not used as the PCA returns a reduced features column with negative values and naïve Bayes only accepts positive values for its implementation.

Classifier:

The two columns namely label and features from the training and test dataset are passed on to the Naive Bayes classifier to train the model and predict the labels. The output is then fetched to the spark ml metrics class to calculate the confusion matrix, which is then used to calculate accuracy and other quality metrics.

```

NaiveBayesModel nbmodel = nb.fit(output);

Dataset<Row> predictions = nbmodel.transform(output2);
predictions = predictions.withColumn("labeltmp", predictions.col("label").cast(DoubleType))
    .drop("labeltmp")
    .withColumnRenamed("labeltmp", "label");
predictions.select("label", "prediction").show(2);

MulticlassMetrics metrics = new MulticlassMetrics(predictions.selectExpr("cast(label as double) label", "prediction"));

// Confusion matrix
org.apache.spark.mllib.linalg.Matrix confusion = metrics.confusionMatrix();
// System.out.println("Confusion matrix: \n" + confusion);

// Overall statistics
System.out.println("Accuracy = " + metrics.accuracy());

```

Multi-Layer Perceptron Classifier

Introduction

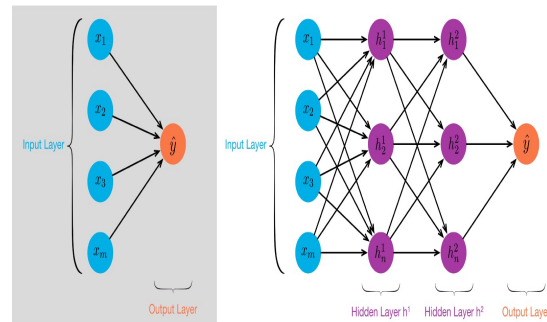
The Artificial Neural Network field is usually depicted as neural networks or multi-layer perceptron although they are called most useful type of neural network. A perceptron is considered a single neuron model that was a predecessor to larger neural networks. A multilayer perceptron (MLP) is a feedforward artificial neural network model maps the set of our input to create a set of appropriate outputs. It consists of multiple layers of nodes in a directed graph where each layer is mutually connected. Each node is a neuron connected with a nonlinear activation function. MLP uses a technique called backpropagation which utilizes a supervised learning technique for training the network.

Activation Function

If a multilayer perceptron has a linear activation function in all neurons, that means a linear function that maps the weighted inputs to the output of each neuron, so it can be easily verified with the use of linear algebra that the number of layers can be reduced to the standard two-layer input-output model. The two main activation functions used in current applications are both sigmoid, and are described by, in which the former function is a hyperbolic tangent which ranges from -1 to 1, and the latter, the logistic function, is similar in shape but ranges from 0 to 1.

Multilayer Perceptron: - A Multi-Layer Perceptron (MLP) accumulates one or more hidden layers (on top of one input and output layer). Although, a single layer perceptron can only learn or recognise linear functions, on the other hand non-linear functions are learned in multi-layer perceptron.

Input Layer: - Data in the input layer is labelled as x with subscript 1, 2, 3, ..., m . Neurons in the hidden layer are labelled as h with subscripts 1, 2, 3, ..., n . The hidden layer specified in algorithm implemented on dataset might differ from the number in input data. The hidden layer is superscripted with 1. This is done as we have more than one hidden layer where first is superscripted as 1 and similar pattern is followed. To further explain input data as (x_1, x_2, \dots, x_m) , is called as m features. On step further, we will multiply each of the m features with a weight (w_1, w_2, w_m) and procedurally sum them all together, that we call dot product. $W \cdot X = w_1x_1 + w_2x_2 + \dots + w_mx_m = \sum_{i=1}^m w_ix_i$. If we add bias value to it, further explained in image



Implementation

Data Processing

In this phase of project, we imported the data in spark environment using Spark Session builder. Spark read function is used to read the workload Training and Test Data set. We split the dataset into categories where first column is label and rest of the column is features. Then in next step we used Vector Assembler which is a transformer that combines a given list of columns into single vector column. It accepts the input columns which are then concatenated into vector in a specified order. In our implementation Vector assembler has defined input as “featurelist” and output as “Features 2”. After transformation trainset and test set data set are then saved into dataset output and output2 Dataset respectively. Then in our next step we considered 55 K-set features to fit into PCA model with considering trainset dataset as input. Transformation is done on both vector assembler. The output is then fetched by the spark ml metrics class to calculate the confusion matrix, which is then used to calculate accuracy and other quality metrics.

Classifier Implementation: -

Spark ML based multilayer perceptron classifier has following predefined parameter which are as follows: - Layers, Tolerance of iteration, Block size of the learning, Seed size, Max iteration. As per the property of algorithm the smaller the value of convergence tolerance will lead to higher accuracy with the cost of more iterations. The default block size parameter is 128 and maximum number of iteration is set as 100 as default value.

In this section we created two hidden layers and we select 55 input datasets. Then created the output after transforming the input one hidden layer neurons of size 40 at which we have received higher accuracy. As per our observation hidden layer neuron number plays an important role in getting the accuracy as we proceed to increase neuron number our accuracy increased significantly. On considering layer neuron number as 5 ,10 ,20 30 ,40 the accuracy has varied as follow with respective 0.4127044294152317, 0.7915960903449425, 0.8580760148693766, 0.9047044413925787, 0.9144671514816372. For the evaluation of accuracy, we used Multilayer Classifier core library train fit and test transform Train and Test Dataset respectively. We have also analyzed that if we reduce the hidden layers neuron corresponding to other the accuracy will vary.

Evaluation and observations: - As we consider to analyze accuracy with different input parameter k-set value , we have observed the accuracy gets reduced with very slight variation where decrease in accuracy is observed with increased in value of parameter input values. While considering different neuron number hidden layer values are as follows with respective accuracy. 55, 65, 85 to generate the Test accuracy, 0.9174, 0.9337333333333333, 0.9456833333333333 respectively. The concept of increase can be explained by the data distribution and computation done more accurately with more weighted dot combination to compute to show better result, but I will affect the efficiently of algorithm.

```

// input layer of size 4 (categories), the intermediate of size 5 and 4
// and output of size 3 (classes)
int[] layers = new int[] {55, 85, 95};

// create the trainer and set its parameters
MultilayerPerceptronClassifier trainer = new MultilayerPerceptronClassifier()
    .setLayers(layers)
    .setBlockSize(128)
    .setSeed(1234L)
    .setMaxIter(100);

// train the model
MultilayerPerceptronClassificationModel multimodel = trainer.fit(trainingset);

// compute accuracy on the test set
Dataset<Row> result = multimodel.transform(trainingset);
Dataset<Row> predictionAndLabels = result.select("prediction", "label");
predictionAndLabels.show(30);
MulticlassClassificationEvaluator evaluator = new MulticlassClassificationEvaluator()
    .setMetricName("accuracy");

System.out.println("Test set accuracy = " + evaluator.evaluate(predictionAndLabels));

}
// Load training data

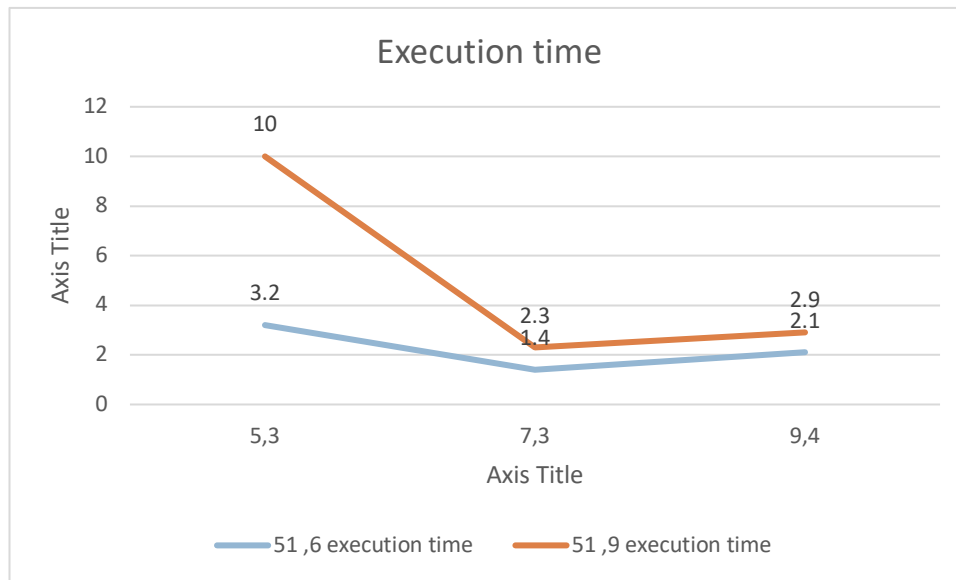
```

Analysis and comparison between Naïve Bayes and Multilayer Preceptor Spark ML algorithms

The comparison and analysis between these two algorithms can be divided in to 2 main categories i.e. quality metrics and performance metrics. The quality metrics consists of metrics that affect the quality of the algorithm and can be attributed to metrics such as accuracy, precision, recall and f1 score. A draw back in the case of Naïve Bayes is that these measures can not be evaluated by changing the feature vector dimensions due to non-usage of the PCA algorithm (non-negative values not accepted in naïve Bayes). While the multilayer preceptor algorithm does take values such as hidden layer neurons but the comparison between these two algorithms for quality metrics takes only one optimum value set for multipreceptor. The second category of analysis is based on the performance element where different environment properties such as executor cores and num executor are used to analyze patterns related to execution time and I/O costs. The provided university cluster was used for conducting these tests.

Quality Metrics:

To analyze the quality among these algorithms we used multipreceptor with dimension set to 55 and the neurons set to 40 and 40. For Naive Bayes no input values can be specified as discussed earlier. The graph produced from this execution for these algorithms is shown below:



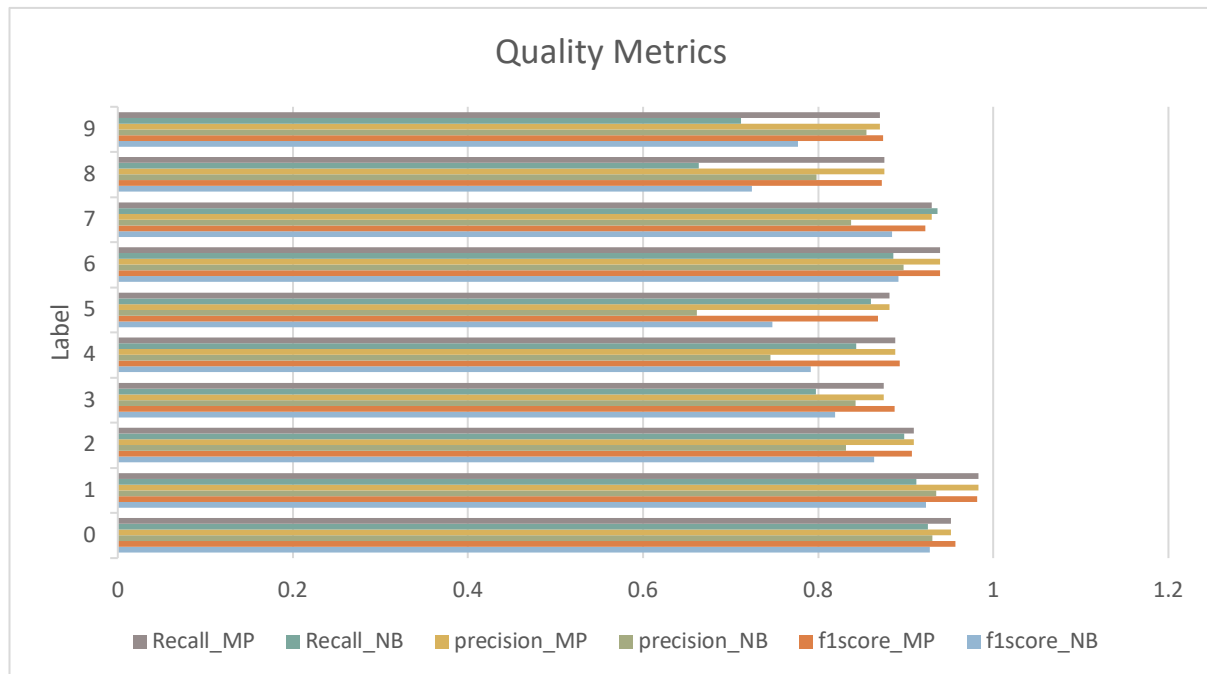
The results show that multilayer preceptor is taking more time to execute as compared to naïve Bayes. This could relate to two main assumptions.

- 1) More jobs and tasks are being used in Multilayer Preceptor algorithm due to the introduction of PCA reduction step which is not being used in naïve Bayes.
- 2) Secondly, the working of multilayer preceptor looks more complicated with usage of neuron layers that could potentially create more tasks for it as per its design, this could also be a vital factor to increase the execution time.

Apart from this it can also be seen that increasing the number of executors is having an adverse(negative) effect on the application due to the executors requested being given as per their availability which causes a delay. Also. For naïve Bayes, the executors being used are being utilized well for the tasks, but the maximum parallelism is only achieved with less number of executors. The same can be said about multilayer preceptor, the parallelism is achieved well with the combination of 3 and 3 but as the executors are increased the time also increases.

2) I/O costs:

To calculate I/O costs, we have considered two properties shuffle read and shuffle write . Both indicate about the amount of data being used for input and output operations per the jobs involved spark. To analyze the I/O we again used the same execution properties and calculated the read and write properties with respect to the two classifiers and plotted the following column graph:



The results show that in terms of accuracy multipeceptor is more accurate with a value of 91.15 % but this could be due to the fact that dimensions in Naïve Bayes could not be reduced, hence giving an accuracy of 83.6 %. To further evaluate the quality metrics, we have considered metrics such as f1 score, recall and precision which give insights on the quality of the prediction as per each label and the graph clearly indicates that all the values of these metrics is higher for the multilayer preceptor as compared to naïve Bayes which again supports the better overall accuracy for multilayer preceptor. The average value for these metrics in naïve Bayes is more than 80% whereas the metrics for multi-layer preceptor is more than 90%.

Performance Metrics:

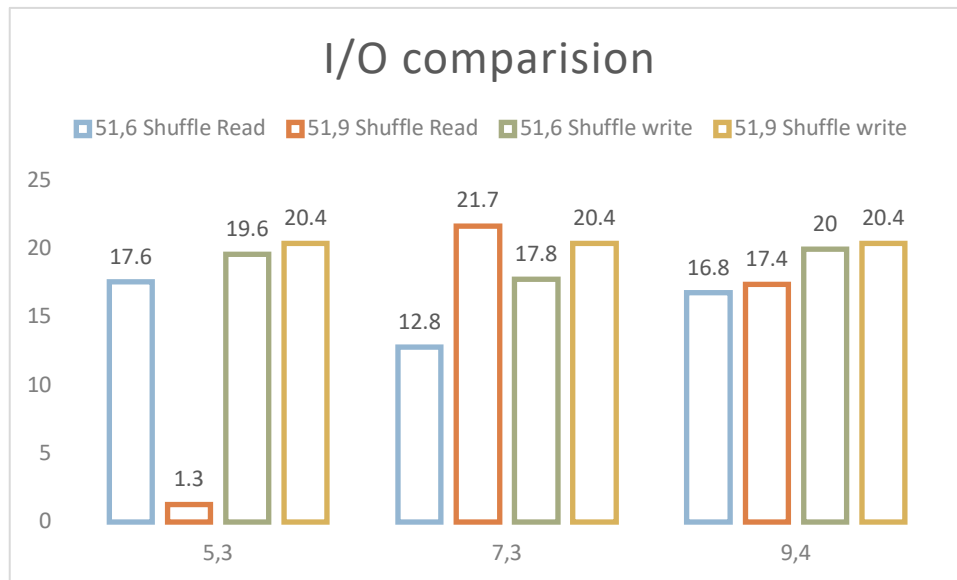
For measuring the performance of these algorithms, we have executed the spark application with three distinct pairs of the number of executors (--num exec) and the cores associated with each container (--exec cores). These values are a combination of 3, 5 and 7 number of executor and its cores. Both these algorithms were executed with the same execution values and the results per the three values can be divided per the execution time of the application and the its I/O costs.

1) Execution time:

The line chart shows execution times per each execution based on given parameters the analysis is discussed below:

The parameters are:

- Num exec – exec cores (3 , 3)
- Num exec – exec cores (5 , 5)
- Num exec – exec cores (7 , 7)



The results show that for each of the 3 executions for these classifiers, the amount of shuffle read and write, both are quite low for Naïve Bayes as compared to multipreceptor. This could be due to the less tasks for naïve Bayes which produces less read and write costs for the data. Another important point to note here is that as the executors are increased for these classifiers the cost of read and write tends to increase due to data being transferred among various containers. This impact can also be related to the increase in the execution time as more containers(executors) are added.

REFERENCING

- CRASH COURSE ON MULTI-LAYER PERCEPTRON NEURAL NETWORKS (2016)
[HTTPS://MACHINELEARNINGMASTERY.COM/NEURAL-NETWORKS-CRASH-COURSE/](https://machinelearningmastery.com/neural-networks-crash-course/)
- A QUICK INTRODUCTION TO NEURAL NETWORKS (2016) [HTTPS://UJJWALKARN.ME/2016/08/09/QUICK-INTRO-NEURAL-NETWORKS/](https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/)
- DEEP LEARNING VIA MULTILAYER PERCEPTRON CLASSIFIER
(2016)[HTTPS://DZONE.COM/ARTICLES/DEEP-LEARNING-VIA-MULTILAYER-PERCEPTRON-CLASSIFIER](https://dzone.com/articles/deep-learning-via-multilayer-perceptron-classifier)
- <https://en.wikipedia.org/wiki/Wikipedia>

APPENDIX

<u>application 1527467200287 1361</u>			without. Multipreceptor
<u>application 1527467200287 1396</u>	7	7	Multipreceptor
<u>application 1527467200287 2393</u>	3.	3.	Multipreceptor
<u>application 1527467200287 2405</u>	5	5.	Multipreceptor
<u>application 1527467200287 2433</u>	7	7.	Multipreceptor
<u>application 1527467200287 2481</u>	7.	7	Bayes
<u>application 1527467200287 2492</u>	5.	5.	Bayes
<u>application 1527467200287 2503</u>	3	3.	Bayes
<u>application 1527467200287 2582</u>	4.	4.	Multipreceptor
<u>application 1527467200287 2677</u>			without Multipreceptor
<u>application 1527467200287 2690</u>	2	2.	Multipreceptor
<u>application 1527467200287 3583</u>	7	7	Bayes
<u>application 1527467200287 3630</u>	2	4	Bayes
<u>application 1527467200287 3648</u>	3	4	Bayes
<u>application 1527467200287 3666</u>	8	8	Bayes
<u>application 1527467200287 3807</u>	5	5	Bayes
<u>application 1527467200287 5083</u>	5	5.	Bayes
<u>application 1527467200287 5110</u>	5.	5.	Bayes
<u>application 1527467200287 5858</u>	1	7	Bayes
<u>application 1527467200287 5893</u>	5	5.	Multipreceptor