

PREDICTION OF BREAST CANCER USING MACHINE LEARNING TECHNIQUES

by

DRISHTI JAIN	15BCE1014
HEMANTH TEJA	15BCE1144
MADHU SAMHITHA	15BCE1193
BHAGAVATULA AISWARYA	15BCE1235

A project report submitted to

Prof. DEIVANAI K

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

in partial fulfilment of the requirements for the course of

CSE4014- HIGH PERFORMANCE COMPUTING

in

B.Tech. (Computer Science and Engineering)



VIT UNIVERSITY, CHENNAI

Vandalur – Kelambakkam Road

Chennai – 600127

Introduction

Data Standards in cancer research have been evolving considerably. Technical Standards and data administration are the requirements of Diagnosis and Prognosis with enormous challenges. Cancer, a deadly disease can be analysed with the innumerable genomic data and ethical analysis with big data functionalities as its crux. Breast Cancer Wisconsin data set from the [UCI Machine learning repo](#) is used to conduct the analysis.

We have implemented the following machine learning algorithms using R and Apache Spark:

- PCA
- Decision trees
- Linear regression
- The Random Forest Technique
- Logistic Regression
- Naive Bayes
- Extreme Gradient Boosting

Data is visualized by manual work. The limitations are that time taken for predicting tumour is high. The correct stage of the breast cancer is not predicted completely accurately. The risk of prediction is high and data is not visualized properly.

Problem statement

Using this dataset of 32 variables measuring the size and shape of cell nuclei, goal is to create a model that will allow us to predict whether a breast cancer cell is benign or malignant.

DATASET description:

Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. A few of the images can be found at: <http://www.cs.wisc.edu/~street/images/>

Separating plane described above was obtained using Multisurface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree Construction Via Linear Programming." Proceedings of the 4th Midwest Artificial Intelligence and Cognitive Science Society, pp. 97-101, 1992], a classification method which uses linear programming to construct a decision tree. Relevant features were selected using an exhaustive search in the space of

- 1-4 features and 1-3 separating planes.
- Number of instances: 569
- Number of attributes: 32 (ID, diagnosis, 30 real-valued input features)
- Attribute information
 - 1) ID number
 - 2) Diagnosis (M = malignant, B = benign)

3-32)

Ten real-valued features are computed for each cell nucleus:

- a) radius (mean of distances from center to points on the perimeter)
- b) texture (standard deviation of gray-scale values)
- c) perimeter
- d) area
- e) smoothness (local variation in radius lengths)
- f) compactness ($\text{perimeter}^2 / \text{area} - 1.0$)
- g) concavity (severity of concave portions of the contour)
- h) concave points (number of concave portions of the contour)
- i) symmetry
- j) fractal dimension ("coastline approximation" – 1)

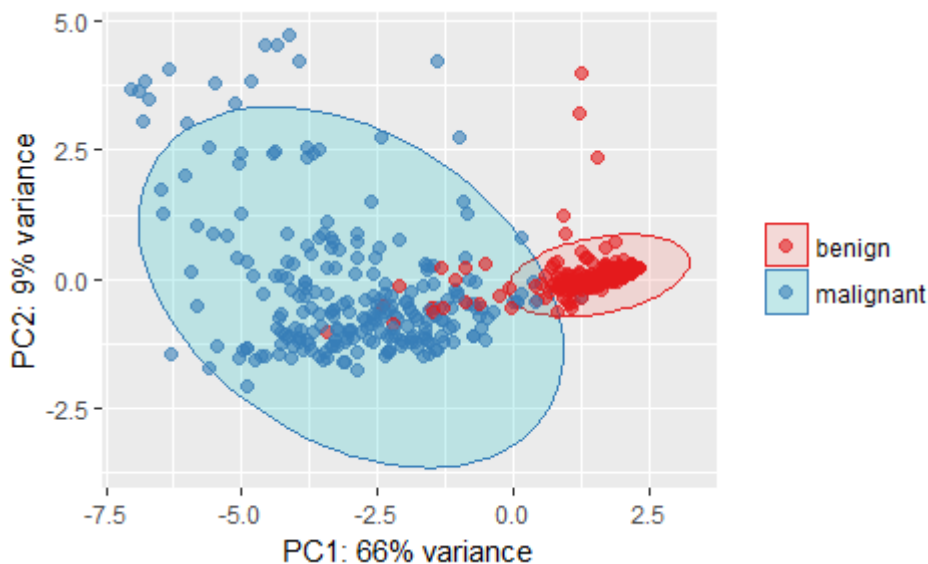
Several of the papers listed above contain detailed descriptions of how these features are computed. The mean, standard error, and "worst" or largest (mean of the three largest values) of these features were computed for each image, resulting in 30 features. For instance, field 3 is Mean Radius, field 13 is Radius SE, field 23 is Worst Radius. All feature values are recorded with four significant digits.

HANDLING MISSING DATA:

The MICE(Multivariate Imputation via Chained Equations) package is used in R to impute the data, that takes carefully of missing values in the dataset. The values for that attribute are taken and ran through a prediction algorithm like linear regression and the predicted values are imputed into the dataset.

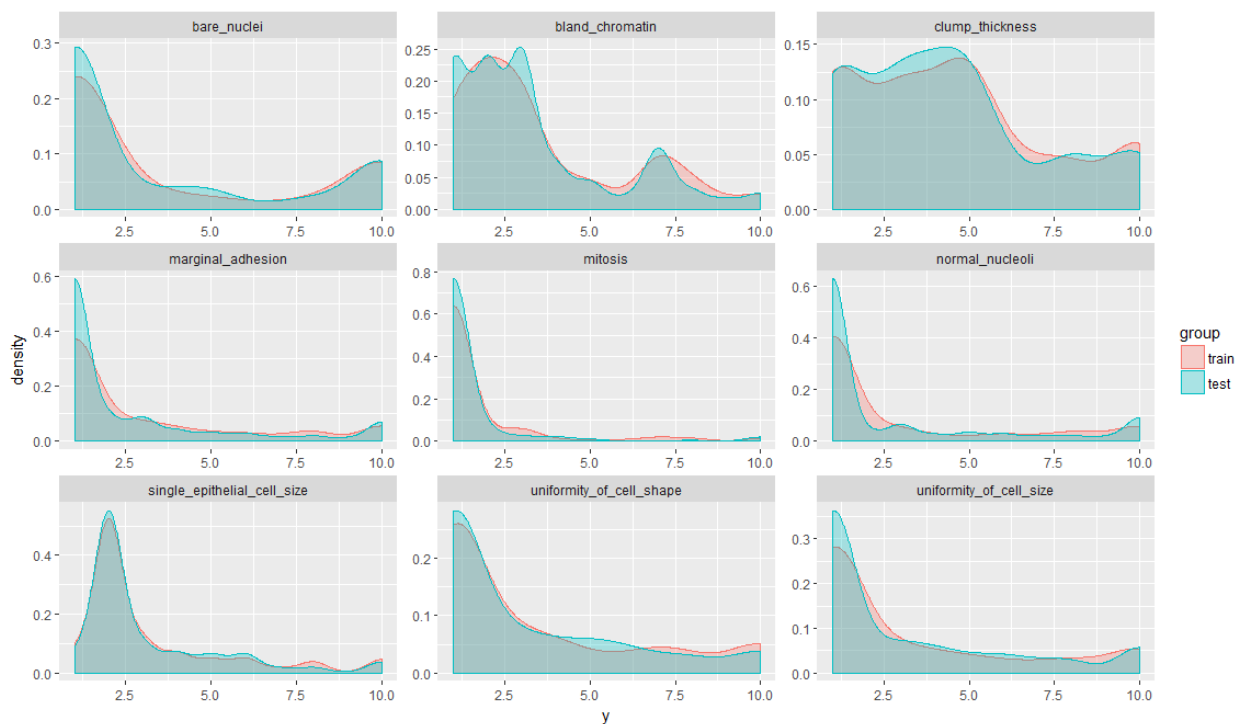
PRINCIPAL COMPONENT ANALYSIS:

We take PCA due to the number of variables in the model, we can try using a dimensionality reduction technique to unveil any patterns in the data. As mentioned in the Exploratory Data Analysis section, there are thirty variables that when combined can be used to model each patient's diagnosis. Using PCA we can combine our many variables into different linear combinations that each explain a part of the variance of the model. By proceeding with PCA we are assuming the linearity of the combinations of our variables within the dataset. By choosing only the linear combinations that provide a majority ($\geq 66\%$) of the co-variance, we can reduce the complexity of our model. We can then more easily see how the model works and provide meaningful graphs and representations of our complex dataset. The first step in doing a PCA, is to ask ourselves whether or not the data should be scaled to unit variance. That is, to bring all the numeric variables to the same scale. In other words, we are trying to determine whether we should use a correlation matrix or a covariance matrix in our calculations of eigen values and eigen vectors (aka principal components).



We have shown how dimensionality reduction technique like principal components analysis can be used to reduce a large number of highly correlated predictors to small set of linear combinations of those predictors. In doing so, we unveiled patterns in the data which led us to build a classification rule using decision trees and random forests.

Test and Train dataset:



REGRESSION

The caret package (short for classification and regression training) contains functions to streamline the model training process for complex regression and classification

problems. We use glm to fit generalized linear models, specified by giving a symbolic description of the linear predictor and a description of the error distribution. linear regression is a statistical method that allows us to summarize and study relationships between two continuous (quantitative) variables:

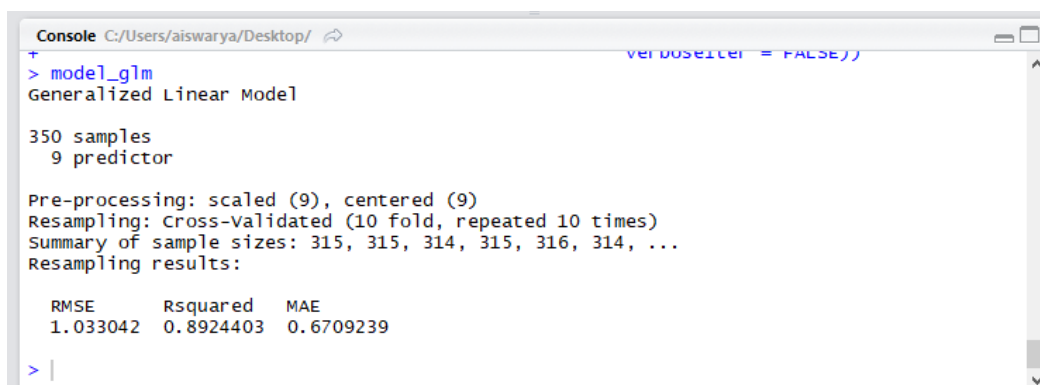
- One variable, denoted x , is regarded as the predictor, explanatory, or independent variable.
- The other variable, denoted y , is regarded as the response, outcome, or dependent variable.

It is important that the model include all relevant variables, it is also important the model does not start with more variables than are justified for the given number of observations. For the set of data, more variables generally produce a better model fit to the data. However, excessive variables will influence the coefficient in the model and contribute to the over-fitting model. A complicated model including many insignificant variables may result in less predictive power and it may often be difficult to interpret the results.

A line that fits the data "best" will be one for which the n prediction errors—one for each observed data point—are as small as possible in some overall sense. One way to achieve this goal is to invoke the "least squares criterion," which says to "minimize the sum of the squared prediction errors."

The validation analysis was performed so as to check whether the regression analysis is suitable or not. The prediction percentage of correct cases from the main samples must be greater than or equal to the validated samples. The validation is using other sample data but having the same coefficient values as the main data to calculate the percentage of correct cases. First, the data were divided into two. The first data containing 70% of the samples is used as the main data and used to find the coefficient values. The second data which contain 30% of the samples is used to validate the main data. Secondly, after obtaining the coefficient values from the main data, the probability of each sample from the validated data are calculated.

The model of linear regression can be seen as:
Considering Clump thickness as the attribute chosen:



```
Console C:/Users/aiswarya/Desktop/
> model_glm
Generalized Linear Model

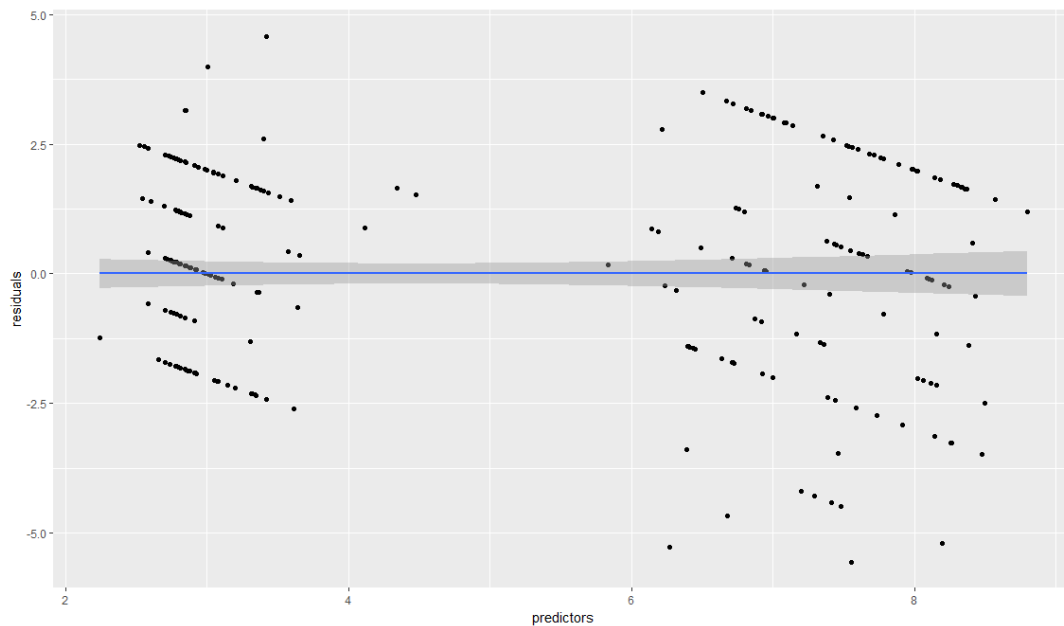
350 samples
9 predictor

Pre-processing: scaled (9), centered (9)
Resampling: Cross-Validated (10 fold, repeated 10 times)
Summary of sample sizes: 315, 315, 314, 315, 316, 314, ...
Resampling results:

RMSE      Rsquared    MAE
1.033042  0.8924403  0.6709239

> |
```

The plotting of the graph against Predictors and residuals:



Linear regression model for the attribute uniformity of cell size:

```

Console C:/Users/aiswarya/Desktop/
> model_glm
Generalized Linear Model

350 samples
 9 predictor

Pre-processing: scaled (9), centered (9)
Resampling: Cross-validated (10 fold, repeated 10 times)
Summary of sample sizes: 316, 315, 316, 314, 315, 315, ...
Resampling results:

    RMSE      Rsquared    MAE
 1.913722  0.5679379  1.599741
> |

```

Thus, we can see that the error rate is lesser by considering uniformity of cell size, as it is foundation to be the most important variable.

DECISION TREES

A decision tree is a tree in which each branch node represents a choice between a number of alternatives and each leaf node represents a decision. It is a type of supervised learning algorithm (with a predefined target variable) that is mostly used in classification problems and works for both categorical and continuous input and output variables. It is one of the most widely used and practical methods for inductive inference. (Inductive inference is the process of reaching a general conclusion from specific examples.)

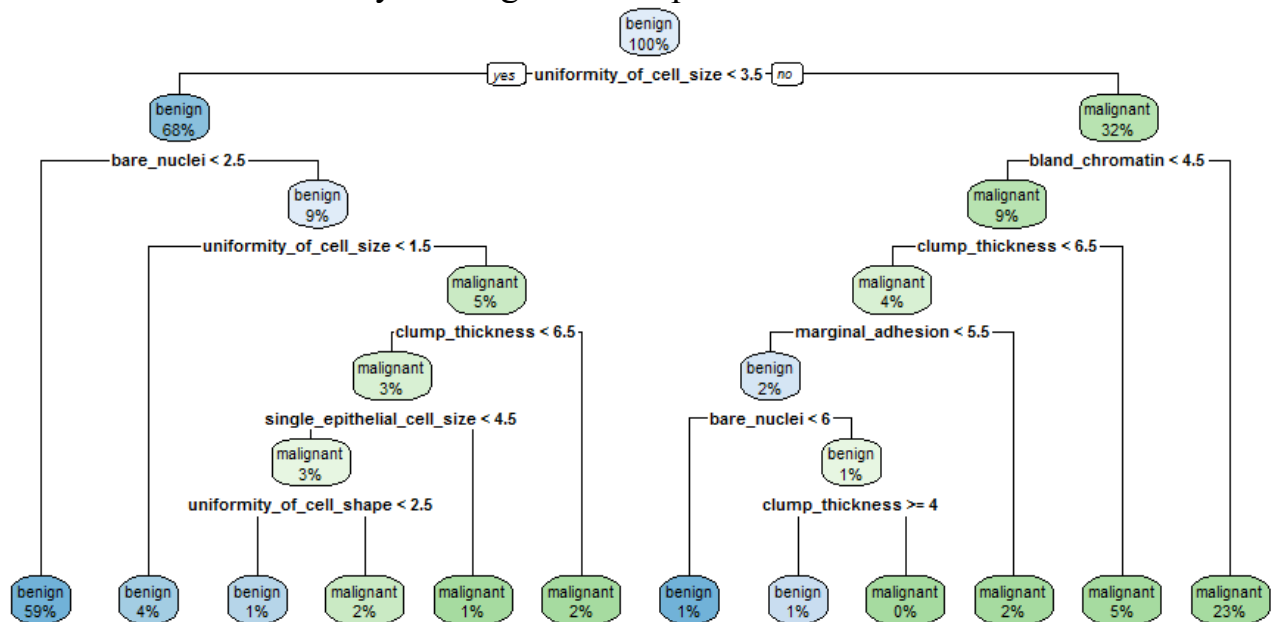
Tree-Based Models

- Recursive partitioning is a fundamental tool in data mining.
- It helps us explore the structure of a set of data, while developing easy to visualize decision rules for predicting a categorical (classification tree) or continuous (regression tree) outcome.

CART Modeling via rpart

- Classification and regression trees can be generated through the rpart package.

The decision tree obtained by running the R script:



RANDOM FORESTS

This concept is used to perform a preliminary screening of variables and to receive important ranks. Random forest predictions are based on the generation of multiple classification trees. To classify a new object from an input vector, put the input vector down each of the trees in the forest. Each tree gives a classification, and we say the tree "votes" for that class. The forest chooses the classification having the most votes (over all the trees in the forest). Random forest do not over fit, thus we can run as many trees as wanted. It is the most accurate and fastest algorithm. We use the `repeatedcv` method in R to train across many folds. The accuracy of the prediction is at 97.12%.

```

> model_rf
Random Forest

350 samples
 9 predictor
 2 classes: 'benign', 'malignant'

Pre-processing: scaled (9), centered (9)
Resampling: Cross-Validated (10 fold, repeated 10 times)
Summary of sample sizes: 316, 315, 315, 315, 315, 315, ...
Resampling results across tuning parameters:

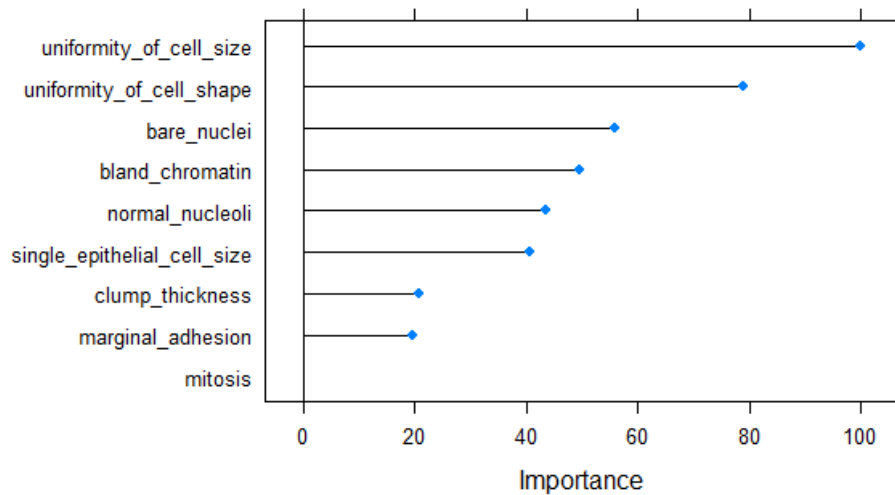
mtry  Accuracy  Kappa
2     0.9654402  0.9238429
5     0.9622969  0.9165943
9     0.9588511  0.9088841

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 2.
> |

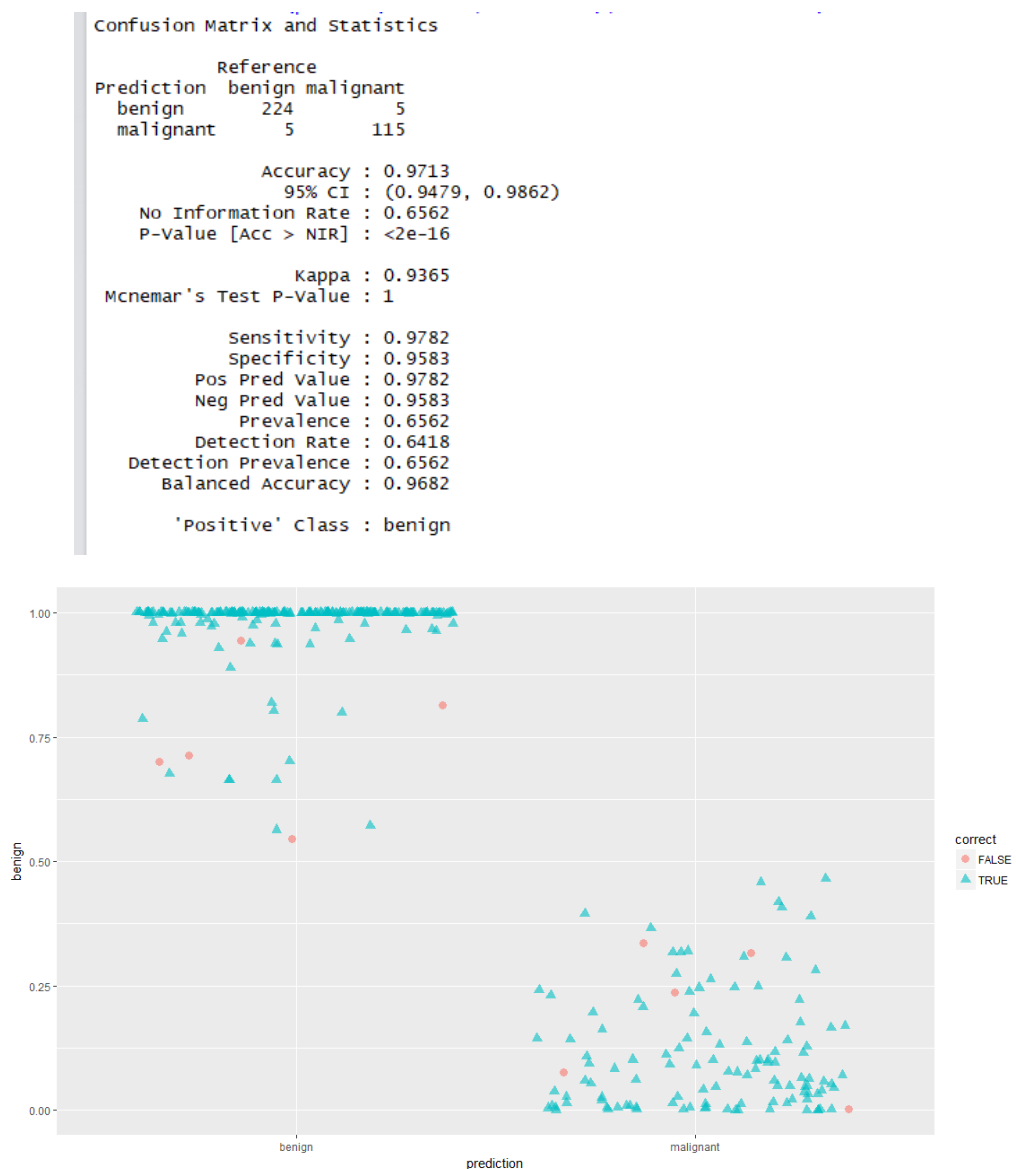
```

FEATURE IMPORTANCE

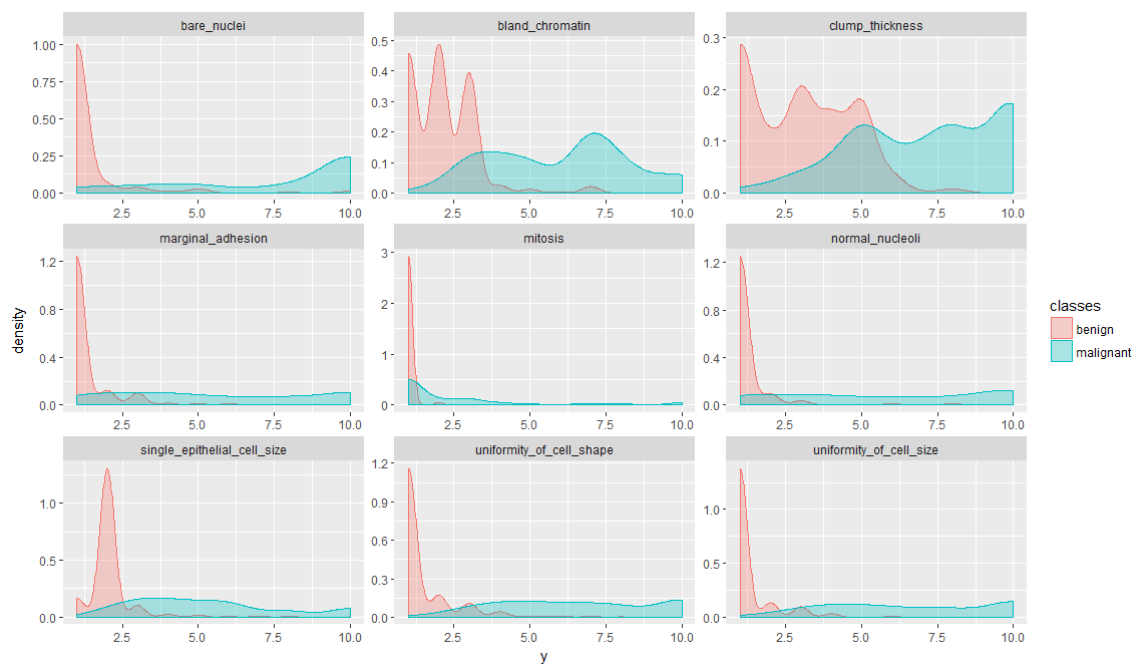
The features or columns are considered across the training data to consider different decision trees and the importance of each feature is calculated. It is found that the uniformity of cell size is the most important attribute, which we have considered again as the attribute for plotting linear regression models, to give lesser error rates.



The confusion matrix values for the dataset used:



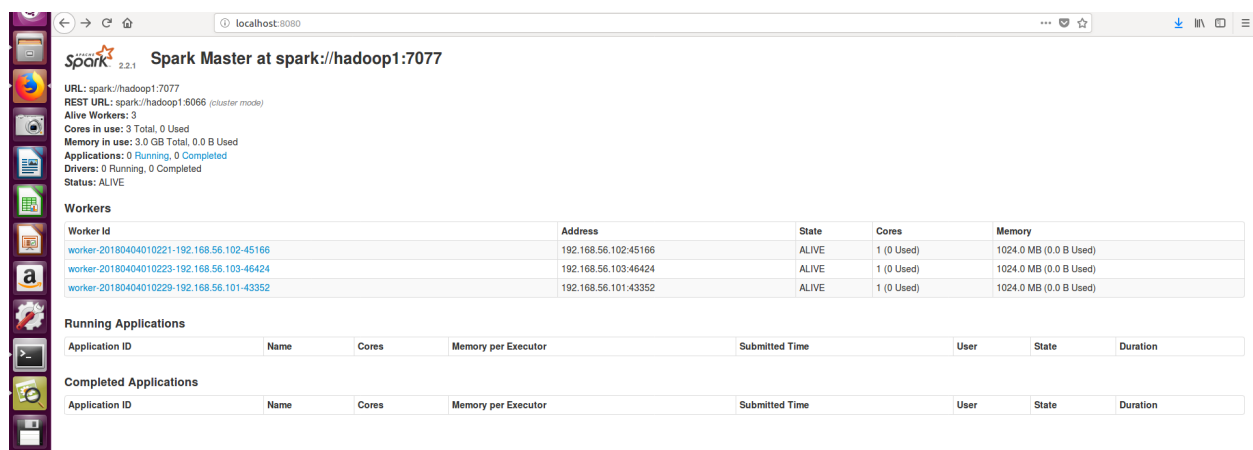
Final representation of the data set with respect to the classes is:



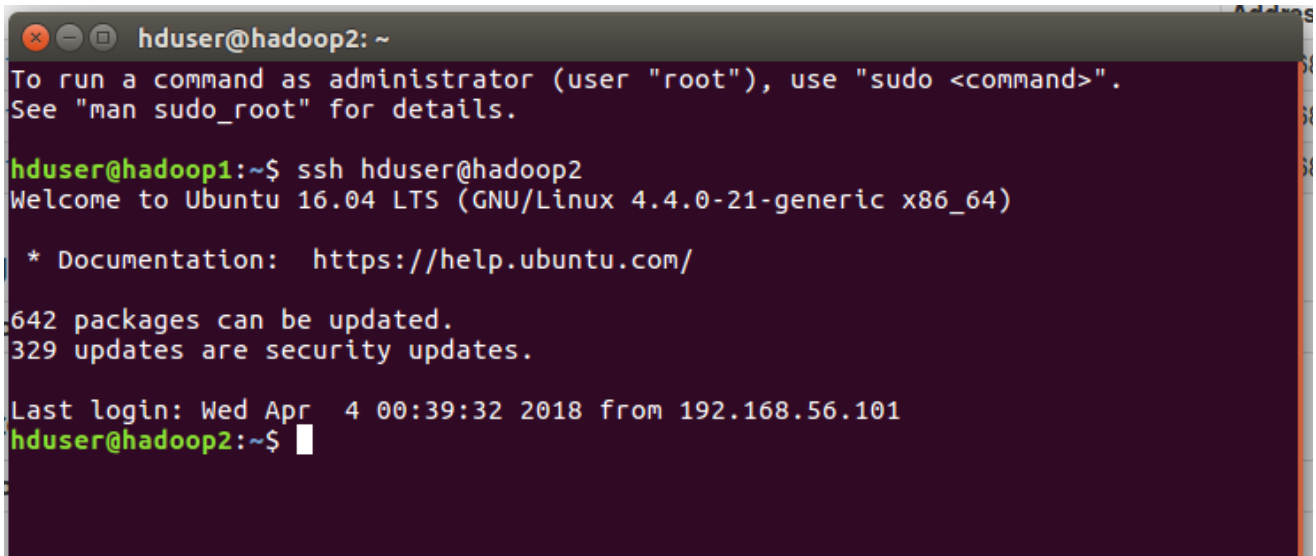
APACHE SPARK:

Apache Spark is a lightning-fast cluster computing technology, designed for fast computation. It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations like interactive queries and stream processing. The main feature of Spark is its in-memory cluster computing that increases the processing speed of an application. Apache Spark has as its architectural foundation the resilient distributed dataset (RDD), a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way. We make use of SCALA and Python programming to implement these algorithms in the Spark environment. We make use of this distributed system to enhance the results for our data set using the Machine Learning algorithms.

A cluster of three worker nodes has been set up. We use three virtual machines with the use of host only network with stable IP address from the host gateway.



The nodes can communicate amongst each other by using the secure shell protocol. For example, we have accessed the second worker node from the first one:

A terminal window titled 'hduser@hadoop2: ~' showing the output of an SSH command. The prompt is 'hduser@hadoop1:~\$' and the command is 'ssh hduser@hadoop2'. The output shows the Ubuntu 16.04 LTS login banner, including the version 'GNU/Linux 4.4.0-21-generic x86_64', documentation link 'https://help.ubuntu.com/', update information '642 packages can be updated. 329 updates are security updates.', and the last login time 'Wed Apr 4 00:39:32 2018 from 192.168.56.101'. The final prompt is 'hduser@hadoop2:~\$' with a cursor.

```
hduser@hadoop2: ~
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

hduser@hadoop1:~$ ssh hduser@hadoop2
Welcome to Ubuntu 16.04 LTS (GNU/Linux 4.4.0-21-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

642 packages can be updated.
329 updates are security updates.

Last login: Wed Apr  4 00:39:32 2018 from 192.168.56.101
hduser@hadoop2:~$
```

1. Logistic Regression

Logistic regression is a statistical method for analysing a dataset in which there are one or more independent variables that determine an outcome. The outcome is measured with a dichotomous variable (in which there are only two possible outcomes). In logistic regression, the dependent variable is binary or dichotomous, i.e. it only contains data coded as 1 (TRUE, success, pregnant, etc.) or 0 (FALSE, failure, non-pregnant, etc.). The goal of logistic regression is to find the best fitting (yet biologically reasonable) model to describe the relationship between the dichotomous characteristic of interest (dependent variable = response or outcome variable) and a set of independent (predictor or explanatory) variables. Logistic regression generates the coefficients (and its standard errors and significance levels) of a formula to predict a logistic transformation of the probability of presence of the characteristic of interest:

$$\text{logit}(p) = b_0 + b_1X_1 + b_2X_2 + b_3X_3 + \dots + b_kX_k$$

where p is the probability of presence of the characteristic of interest.

The accuracy obtained by implementing logistic regression is 99.26% by using SCALA.

```
scala> sqlContext.sql("SELECT clas, avg(thickness) as avgthickness, avg(size) as
  avgsize, avg(shape) as avgshape FROM obs GROUP BY clas ").show
+-----+-----+-----+-----+
|clas|    avgthickness|    avgsize|    avgshape|
+-----+-----+-----+-----+
| 0.0|2.963963963963964|1.3063063063063063|1.4144144144144144|
| 1.0|7.188284518828452| 6.577405857740586| 6.560669456066946|
+-----+-----+-----+-----+

scala> // compute avg thickness grouped by clas (malignant or not)

scala> obsDF.groupBy("clas").avg("thickness").show
+-----+-----+
|clas|    avg(thickness)|
+-----+-----+
| 0.0|2.963963963963964|
| 1.0|7.188284518828452|
+-----+-----+
```

```
scala> obsDF.printSchema
root
|-- clas: double (nullable = false)
|-- thickness: double (nullable = false)
|-- size: double (nullable = false)
|-- shape: double (nullable = false)
|-- madh: double (nullable = false)
|-- epsize: double (nullable = false)
|-- bnuc: double (nullable = false)
|-- bchrom: double (nullable = false)
|-- nNuc: double (nullable = false)
|-- mit: double (nullable = false)

scala>

scala> obsDF.describe("thickness").show
+-----+-----+
|summary|    thickness|
+-----+-----+
|   count|           683|
|   mean|  4.44216691068814|
| stddev|2.8207613188371288|
|    min|           1.0|
```

```
scala> val df2 = assembler.transform(obsDF)
df2: org.apache.spark.sql.DataFrame = [clas: double, thickness: double ... 9 more fields]

scala>

scala> //the transform method produced a new column: features.

scala>

scala> df2.show
+---+-----+---+-----+---+-----+---+-----+---+-----+---+-----+
|clas|thickness|size|shape|madh|epsize|bnuc|bchrom|nNuc|mit|features|
+---+-----+---+-----+---+-----+---+-----+---+-----+---+-----+
| 0.0|      5.0| 1.0|  1.0| 1.0|   2.0| 1.0|   3.0| 1.0|1.0|[5.0,1.0,1.0,1.0,...|
| 0.0|      5.0| 4.0|  4.0| 5.0|   7.0|10.0|   3.0| 2.0|1.0|[5.0,4.0,4.0,5.0,...|
| 0.0|      3.0| 1.0|  1.0| 1.0|   2.0| 2.0|   3.0| 1.0|1.0|[3.0,1.0,1.0,1.0,...|
```

```
scala> // split the dataframe into training and test data

scala>

scala> val splitSeed = 5043
splitSeed: Int = 5043

scala> val Array(trainingData, testData) = df3.randomSplit(Array(0.7, 0.3), splitSeed)
trainingData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [clas: double, thickness: double ... 10 more fields]
testData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [clas: double, thickness: double ... 10 more fields]

scala>

scala> // create the classifier, set parameters for training

scala>

scala> val lr = new LogisticRegression().setMaxIter(10).setRegParam(0.3).setElasticNetParam(0.8)
lr: org.apache.spark.ml.classification.LogisticRegression = logreg_e9d5f9164796
```

```
scala> predictions.show
```

clas	thickness	size	shape	madh	epsize	bnuc	bchrom	nNuc	mit	features
label	rawPrediction	probability	prediction							
0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	3.0	1.0	[1.0,1.0,1.0,1.0,...
0.0	[1.17923510971064...	[0.76481024658406...							0.0	
0.0	1.0	1.0	1.0	1.0	1.0	1.0	3.0	1.0	1.0	[1.0,1.0,1.0,1.0,...
0.0	[1.17670009823299...	[0.76435395397908...							0.0	
0.0	1.0	1.0	1.0	1.0	1.0	1.0	3.0	1.0	1.0	[1.0,1.0,1.0,1.0,...
0.0	[1.17670009823299...	[0.76435395397908...							0.0	
0.0	1.0	1.0	1.0	1.0	2.0	1.0	1.0	1.0	1.0	[1.0,1.0,1.0,1.0,...
0.0	[1.17923510971064...	[0.76481024658406...							0.0	
0.0	1.0	1.0	1.0	1.0	2.0	1.0	2.0	1.0	1.0	[1.0,1.0,1.0,1.0,...
0.0	[1.17796760397182...	[0.76458217679258...							0.0	
0.0	1.0	1.0	1.0	1.0	2.0	1.0	3.0	1.0	1.0	[1.0,1.0,1.0,1.0,...
0.0	[1.17670009823299...	[0.76435395397908...							0.0	
0.0	1.0	1.0	1.0	1.0	2.0	2.0	2.0	1.0	1.0	[1.0,1.0,1.0,1.0,...
0.0	[1.10212796544057...	[0.75065860985163...							0.0	
0.0	1.0	1.0	1.0	1.0	2.0	5.0	1.0	1.0	1.0	[1.0,1.0,1.0,1.0,...
0.0	[0.87587655558565...	[0.70596701397664...							0.0	

```
model: org.apache.spark.ml.classification.LogisticRegressionModel = logreg_e9d5f9164796

scala>

scala> // Print the coefficients and intercept for logistic regression

scala>

scala> println(s"Coefficients: ${model.coefficients} Intercept: ${model.intercept}")
Coefficients: [0.0,0.06503554553146378,0.07181362361391258,0.0,0.0,0.07583963853124671,0.0012675057388233403,0.0,0.0] Intercept: -1.39319142312609

scala>

scala> val predictions = model.transform(testData)
predictions: org.apache.spark.sql.DataFrame = [clas: double, thickness: double, ... 13 more fields]
```

```
scala> // Calculate Metrics

scala>

scala> val lp = predictions.select( "label", "prediction")
lp: org.apache.spark.sql.DataFrame = [label: double, prediction: double]

scala> val counttotal = predictions.count()
counttotal: Long = 199

scala> val correct = lp.filter($"label" === $"prediction").count()
correct: Long = 168

scala> val wrong = lp.filter(not($"label" === $"prediction")).count()
wrong: Long = 31

scala> val truep = lp.filter($"prediction" === 0.0).filter($"label" === $"prediction").count()
truep: Long = 128

scala> val falseN = lp.filter($"prediction" === 0.0).filter(not($"label" === $"prediction")).count()
falseN: Long = 30
```

```
scala>

scala> // Evaluates predictions and returns a scalar metric areaUnderROC(larger is better).

scala>

scala> val accuracy = evaluator.evaluate(predictions)
accuracy: Double = 0.9926910299003322

scala>

scala>
```

2. Naive bayes

Naive Bayes is a simple technique for constructing classifiers: models that assign class labels to problem instances, represented as vectors of feature values, where the class labels are drawn from some finite set. It is not a single algorithm for training such classifiers, but a family of algorithms based on a common principle: all naive Bayes classifiers assume that the value of a particular feature is independent of the value of any other feature, given the class variable. A naive Bayes classifier uses probability theory to classify data. Naive Bayes classifier algorithms make use of Bayes' theorem. The key insight of Bayes' theorem is that the probability of an event can be adjusted as new data is introduced. What makes a naive Bayes classifier naive is its assumption that all attributes of a data point under consideration are independent of each other.

CODE:

coding: utf-8

```

# In[1]:
import pyspark
import pandas as pd
from pyspark.sql.types import *
from pyspark.sql import Row
from pyspark.sql.functions import *
# In[2]:
sc=pyspark.SparkContext.getOrCreate()
sqlContext=SQLContext(sc)
#Reading Data
df=pd.read_csv('breastwisconsin.csv')
df
# In[3]:
schema= StructType([StructField('id',IntegerType(),True) ,
                      StructField('rd',IntegerType(),True) ,
                      StructField('tx',IntegerType(),True) ,
                      StructField('pm',IntegerType(),True) ,
                      StructField('ar',IntegerType(),True) ,
                      StructField('sm',IntegerType(),True) ,
                      StructField('cn',IntegerType(),True) ,
                      StructField('cc',IntegerType(),True) ,
                      StructField('cp',IntegerType(),True) ,
                      StructField('sym',IntegerType(),True) ,
                      StructField('fd',StringType(),True)])
# In[4]:
dfs=sqlContext.createDataFrame(df,schema)
dfs
# In[5]:
from pyspark.sql.functions import *
dfs.take(2)
dfs.printSchema()
# In[6]:
dfs.first()
dfs = dfs.select("rd",
                 "tx",
                 "pm",
                 "ar",
                 "sm",
                 "cn",
                 "cc",
                 "cp",
                 "sym",
                 "fd")
dfs.describe().show()
# In[7]:
from pyspark.ml.feature import StringIndexer,VectorAssembler,IndexToString
labelindexer = StringIndexer(inputCol = "fd", outputCol = "label").fit(dfs)

```

```

featureassembler = VectorAssembler(inputCols =
["rd","tx","pm","ar","sm","cn","cc","cp","sym"], outputCol = "features")
featureassembler
# In[8]:
train_data, test_data=dfs.randomSplit([.8,.2],seed=1234)
# In[10]:
from pyspark.ml.classification import NaiveBayes
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer
# In[11]:
nb=NaiveBayes(labelCol="label",featuresCol =
"features",smoothing=1.0,modelType="multinomial")
# In[12]:
pipeline = Pipeline(stages = [labelindexer,featureassembler,nb])
# In[13]:
model=pipeline.fit(train_data)
# In[14]:
predictions = model.transform(test_data)
predictions
# In[15]:
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
evaluator = MulticlassClassificationEvaluator(labelCol = "label",predictionCol =
"prediction",metricName = "accuracy")
# In[16]:
accuracy1 = evaluator.evaluate(predictions)
print(accuracy1)
sc.stop()

```

The screenshot shows a Jupyter Notebook titled "Naive Bayes" running on a local host. The notebook contains 16 input cells and one output cell. The code in the input cells matches the text provided in the first block. The output cell for In[7] shows the creation of the VectorAssembler object. The output cell for In[14] shows the predictions as a DataFrame with columns: rd, tx, pm, ar, sm, cn, cc, cp, sym, fd, string, label, double, features, vector, probability, vector, prediction, double. The final output cell for In[16] shows the accuracy1 value as 0.8688524590163934.

```

In [7]: from pyspark.ml.feature import StringIndexer, VectorAssembler, IndexToString
labelindexer = StringIndexer(inputCol = "fd", outputCol = "label").fit(dfs)

featureassembler = VectorAssembler(inputCols = ["rd","tx","pm","ar","sm","cn","cc","cp","sym"], outputCol = "features")
featureassembler

Out[7]: VectorAssembler_4fdd9d7b35aea12f4f80

In [8]: train_data, test_data=dfs.randomSplit([.8,.2],seed=1234)

In [10]: from pyspark.ml.classification import NaiveBayes
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer

In [11]: nb=NaiveBayes(labelCol="label",featuresCol = "features",smoothing=1.0,modelType="multinomial")

In [12]: pipeline = Pipeline(stages = [labelindexer,featureassembler,nb])

In [13]: model=pipeline.fit(train_data)

In [14]: predictions = model.transform(test_data)
predictions

Out[14]: DataFrame[rd: int, tx: int, pm: int, ar: int, sm: int, cn: int, cc: int, cp: int, sym: int, fd: string, label: double, fe
atures: vector, rawPrediction: vector, probability: vector, prediction: double]

In [15]: from pyspark.ml.evaluation import MulticlassClassificationEvaluator

evaluator = MulticlassClassificationEvaluator(labelCol = "label",predictionCol = "prediction",metricName = "accuracy")

In [16]: accuracy1 = evaluator.evaluate(predictions)
print(accuracy1)
sc.stop()

0.8688524590163934

```


The accuracy we have obtained for this algorithm is:

```
In [16]: accuracy1 = evaluator.evaluate(predictions)
          print(accuracy1)
          sc.stop()
          0.8688524590163934
```

3. Extreme Gradient Boost

XGBoost is an implementation of gradient boosted decision trees designed for speed and performance. This algorithm goes by lots of different names such as gradient boosting, multiple additive regression trees, stochastic gradient boosting or gradient boosting machines.

Boosting is an ensemble technique where new models are added to correct the errors made by existing models. Models are added sequentially until no further improvements can be made. A popular example is the AdaBoost algorithm that weights data points that are hard to predict.

Gradient boosting is an approach where new models are created that predict the residuals or errors of prior models and then added together to make the final prediction. It is called gradient boosting because it uses a gradient descent algorithm to minimize the loss when adding new models.

We need to define a so-called objective function, to measure the performance of the model given a certain set of parameters.

A very important fact about objective functions is they must always contain two parts: training loss and regularization.

$$\text{obj}(\theta) = L(\theta) + \Omega(\theta)$$

where L is the training loss function, and Ω is the regularization term. The training loss measures how predictive our model is on training data. For example, a commonly used training loss is mean squared error.

$$L(\theta) = \sum_i (y_i - \hat{y}_i)^2$$

CODE:

```
# coding: utf-8
# In[2]:
import pyspark
import pandas as pd
```

```

from pyspark.sql.types import *
from pyspark.sql import Row
from pyspark.sql.functions import *
# In[3]:
sc=pyspark.SparkContext.getOrCreate()
sqlContext=SQLContext(sc)
#Reading Data
df=pd.read_csv('breastwisconsin.csv')
df
# In[4]:
# rd = radius , tx = texture , pm = perimeter , ar = area , sm = smoothness , cn =
compactness, cc = concavity , cp = concave points , sym = symmetry , fd = fractal
dimension
schema= StructType([StructField('id',IntegerType(),True) ,
                      StructField('rd',IntegerType(),True) ,
                      StructField('tx',IntegerType(),True) ,
                      StructField('pm',IntegerType(),True) ,
                      StructField('ar',IntegerType(),True) ,
                      StructField('sm',IntegerType(),True) ,
                      StructField('cn',IntegerType(),True) ,
                      StructField('cc',IntegerType(),True) ,
                      StructField('cp',IntegerType(),True) ,
                      StructField('sym',IntegerType(),True) ,
                      StructField('fd',StringType(),True)])

# In[7]:
dfs=sqlContext.createDataFrame(df,schema)
dfs

# In[8]:
from pyspark.sql.functions import *
dfs.take(2)
dfs.printSchema()

# In[9]:
dfs.first()
dfs = dfs.select("rd",
                  "tx",
                  "pm",
                  "ar",
                  "sm",
                  "cn",
                  "cc",
                  "cp",
                  "sym",
                  "fd")

```

```
dfs.describe().show()
```

```
# In[10]:
```

```
from pyspark.ml.feature import StringIndexer, VectorAssembler, IndexToString  
labelindexer = StringIndexer(inputCol = "fd", outputCol = "label").fit(dfs)
```

```
featureassembler = VectorAssembler(inputCols =  
["rd", "tx", "pm", "ar", "sm", "cn", "cc", "cp", "sym"], outputCol = "features")
```

```
featureassembler
```

```
# In[11]:
```

```
train_data, test_data=dfs.randomSplit([.8,.2],seed=1234)
```

```
# In[12]:
```

```
#from pyspark.ml.regression import LinearRegression  
#from pyspark.ml.classification import MultilayerPerceptronClassifier  
from pyspark.ml.classification import GBTClassifier  
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
```

```
train_data.printSchema()
```

```
train_data.show()
```

```
#nb=NaiveBayes(smoothing=1.0,modelType="multinomial")
```

```
#lr=LinearRegression(labelCol="label",featuresCol="features",maxIter=10,regParam=0.3,  
lasticNetParam=0.8)
```

```
# In[14]:
```

```
#mlp = MultilayerPerceptronClassifier(maxIter=50,featuresCol = "features",labelCol  
="label", layers=[10,5,2], blockSize=128, seed=None)  
from pyspark.ml import Pipeline  
from pyspark.ml.feature import StringIndexer  
#stringIndexer = StringIndexer(inputCol="label", outputCol="indexed")  
gbt = GBTClassifier(maxIter = 10 ,labelCol="label",featuresCol = "features")
```

```
# In[15]:
```

```
pipeline = Pipeline(stages = [labelindexer,featureassembler,gbt])
```

```
# In[21]:
```

```
model=pipeline.fit(train_data)
```

```
# In[22]:
```

```
predictions = model.transform(test_data)  
predictions
```

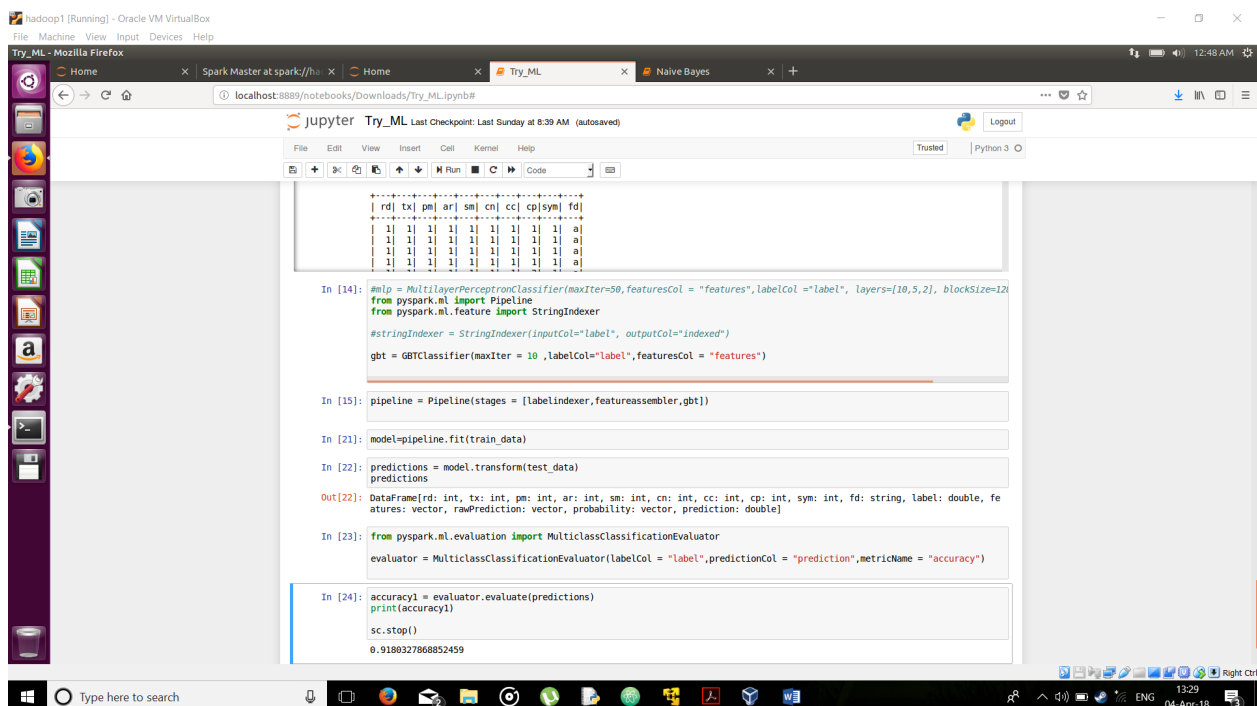
```
# In[23]:
```

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
evaluator = MulticlassClassificationEvaluator(labelCol = "label",predictionCol =
"prediction",metricName = "accuracy")
```

```
# In[24]:
```

```
accuracy1 = evaluator.evaluate(predictions)
print(accuracy1)
```

```
sc.stop()
```



```
rd| tx| pm| ar| sm| cn| cc| cp| sym| fd|
1| 1| 1| 1| 1| 1| 1| 1| 1| a|
1| 1| 1| 1| 1| 1| 1| 1| 1| a|
1| 1| 1| 1| 1| 1| 1| 1| 1| a|
1| 1| 1| 1| 1| 1| 1| 1| 1| a|

In [14]: #mlp = MultilayerPerceptronClassifier(maxIter=50,featuresCol = "features",labelCol ="label", layers=[10,5,2], blockSize=128)
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer
#stringIndexer = StringIndexer(inputCol="label", outputCol="indexed")
gbt = GBTClassifier(maxIter = 10 ,labelCol="label",featuresCol = "features")

In [15]: pipeline = Pipeline(stages = [labelIndexer,featureassembler,gbt])

In [21]: model=pipeline.fit(train_data)

In [22]: predictions = model.transform(test_data)
predictions
Out[22]: DataFrame[rd: int, tx: int, pm: int, ar: int, sm: int, cn: int, cc: int, cp: int, sym: int, fd: string, label: double, features: vector, rawPrediction: vector, probability: vector, prediction: double]

In [23]: from pyspark.ml.evaluation import MulticlassClassificationEvaluator
evaluator = MulticlassClassificationEvaluator(labelCol = "label",predictionCol = "prediction",metricName = "accuracy")

In [24]: accuracy1 = evaluator.evaluate(predictions)
print(accuracy1)
sc.stop()
0.9180327868852459
```

The accuracy arrived at for XGBoost using python in Spark is:

```
In [24]: accuracy1 = evaluator.evaluate(predictions)
print(accuracy1)

sc.stop()

0.9180327868852459
```