Project 3: Classification Algorithms

CSE 601: Data Mining and Bioinformatics

AVIJEET MISHRA
(AVIJEETM@BUFFALO.EDU)
UB#:50169242

PRITHVI GOLLU INDRAKUMAR
(PGOLLUIN@BUFFALO.EDU)
UB#:50169089

DEPT OF COMPUTER SCIENCE,
UNIVERSITY AT BUFFALO

Contents

Contents

Introduction

Classification is a data mining function that assigns items in a collection to target categories or classes. The goal of classification is to accurately predict the target class for each case in the data. For example, a classification model could be used to identify patients as low, medium, or highly prone to cancer. Another example of classification model would be that it could be used by a marketing manager at a company who needs to analyze a customer with a given profile, who will buy a new computer.

A classification task begins with a data set in which the class assignments are known. Each record contains a set of attributes, out of which one of the attributes is the class. For example, a classification model that predicts cancer risk could be developed based on observed data for many patients over a period of time. In addition to the normal disease, the data might track employment history, home ownership or rental, years of residence, number and type of investments, and so on. Credit rating would be the target, the other attributes would be the predictors, and the data for each customer would constitute a case.

A test set is used to determine the accuracy of the model. Usually, the given data set is divided into training and test sets, with training set used to build the model and test set used to validate it. So we have to find a model for class attribute as a function of the values of other attributes. Major goal of classification is to assign new records to a class as accurately as possible using previously unseen records.

Classifications are discrete and do not imply order. Continuous, floating-point values would indicate a numerical, rather than a categorical, target. A predictive model with a numerical target uses a regression algorithm, not a classification algorithm. The simplest type of classification problem is binary classification. In binary classification, the target attribute has only two possible values: for example, high credit rating or low credit rating. Multiclass targets have more than two values: for example, low, medium, high, or unknown credit rating.

The given 4 datasets represent data with their various attribute values which are the genes experiment conditions and last column contains their classification values. We have implemented K-Nearest Neighbor, Naïve Bayes, and Decision Tree (with Random forest and Boosting) for classification in this project. In the above mentioned algorithms, we are classifying records based on their attribute values after training a part of data and classifying or testing on rest of the data. K-fold Cross Validation is used in all the algorithms as a method of estimation.

Implementation

All the 5 algorithms are implemented in C# using Microsoft Visual Studio as a platform. The data is read from the Excel files of the datasets. In all the algorithms, the continuous and discrete features are handled differently. We choose binary split as a way to split discrete features after analyzing the complexity of implementation and efficiency of the algorithm in case of each type of discrete feature splitting on the given datasets. This was especially in the case of decision tree classification.

As mentioned before K-fold Cross Validation is used as a method of estimation in all the algorithms. In Cross Validation, K-fold means the data is partitioned into k disjoint subsets. In the first iteration training is performed on k-1 partitions and remaining 1 partition is used for testing. And in the second iteration some other subset is considered for testing the other subsets are used to train the data. This is done until all the subsets are used for testing. The overall Accuracy, Precision, Recall and F-Measure of an algorithm is the mean or average of the same at each iteration.

Code Snippet: K-Fold Cross Validation

```
int k_fold = 10;
int rows_counter = 0;
int whole_counter = 1;
while (rows_counter < k_fold && whole_counter < rows)// k fold iteration
{
       string ind = ",";
       int t = rows / k_fold;
       int k = 0;
       for (int i = 1; i <= t + 1 && whole_counter <= rows; i++)// partitioning data
       {
             k++;
             ind = ind + "," + whole_counter + ",";
             whole_counter++;
       }
       rows_counter++;
       for (int row = 1; row <= rows; ++row)
       {
           if (!ind.Contains("," + row + ","))
           {
               for (int col = 1; col <= cols - 1; ++col)
               {
                 fv[r_rows, col] = Convert.ToDouble(valueArray[row, col]);//training data

               }
               cl[r_rows] = Convert.ToInt32(valueArray[row, cols]);//class of training data
               r_rows++;
           }
           else
           {
           for (int col = 1; col <= cols - 1; ++col)
           {
               //access each cell
               test_fv[r_rows, col] = Convert.ToDouble(valueArray[row, col]);//test data
           }
           test_cl[t_rows] = Convert.ToInt32(valueArray[row, cols]); //class of test data
           t_rows++;
      }
}
```

K-Nearest Neighbor Classification Algorithm

In k-NN classification, the output is a class membership. An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If k = 1, then the object is simply assigned to the class of that single nearest neighbor. k-NN is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until classification. The k-NN algorithm is among the simplest of all machine learning algorithms.

The naive version of the algorithm is easy to implement by computing the distances from the test example to all stored examples, but it is computationally intensive for large training sets. Using an appropriate nearest neighbor search algorithm makes k-NN computationally tractable even for large data sets. Many nearest neighbor search algorithms have been proposed over the years; these generally seek to reduce the number of distance evaluations actually performed.

Selection of k is a very critical factor as if k is very small then it is too sensitive to noise points but if it is too high then neighborhood may include points from other classes. In our implementation we have selected value of K as 9 as it is giving optimal results for all the datasets. As a dataset could comprise of continuous as well as nominal attributes so both of the scenarios are handled in our implementation.

In KNN algorithm, the nearest neighbor is calculated using Euclidean distance which is as follows:

$$\boxed{dist(X,Y)=\sqrt{(x_1-y_1)^2+\cdots+(x_n-y_n)^2}}$$

In our implementation, initially distance is calculated for each possible pair of testing and training data and then it is sorted in ascending manner. After that weight (1/distance)^2) is calculated for each pair and cumulative weight is calculated for the nearest k-neighbors. Assignment of that record is done to the appropriate cluster on the basis of voting. This process is repeated for all the records so that a final assignment of class is achieved. Once classification is completed, it is compared with the ground truth values of the training data.

In the following Code snippet:1 & 2, above mentioned algorithm is implemented.

Code Snippet 1: Training the classifier: Distance Matrix Calculation

```
for (int i = 1; i < t_rows; i++) // each row of test data
{
  Dictionary<string, double> dm = new Dictionary<string, double>();
  for (int j = 1; j < r_rows; j++) // each row of training data
  {
     double sum = 0;
     int entry;
     for (int m = 1; m <= cols; m++)  // for loop on column
     {
        // checking if the current featuen column is distrecte or not

       if (col_string_dic.TryGetValue(m, out entry))
       {
         // if discrete then the distance between different values is 1 else 0
         if (test_fv[i, m] != fv[j, m])
         {
            sum = sum + 1;
         }
       }
       else
       {// Euclidian distance
        sum=sum +Math.Pow(Convert.ToDouble(test_fv[i, m])-Convert.ToDouble(fv[j, m]), 2);
       }
    }
    dm.Add(i + "," + j, sum); //distance matrix
}
```

In the above code snippet the code for calculating the distance matrix is shown. We also see how features with discrete values are handled. “col_string_dic” is a hashmap dictionary where all the discrete feature columns are stored. In the case of the discrete feature, if the value of the training and testing data are same then the distance is 0 else its 1.

In the next Code snippet: 2, the code shows how exactly the weights of the k nearest neighbors are used to classify the new testing data. The weights of K neighbors are normalized and added based on their class. Then the class with the maximum weight is selected as the class of the new testing data.

Code Snippet 2: Training the classifier: K nearest neighbor

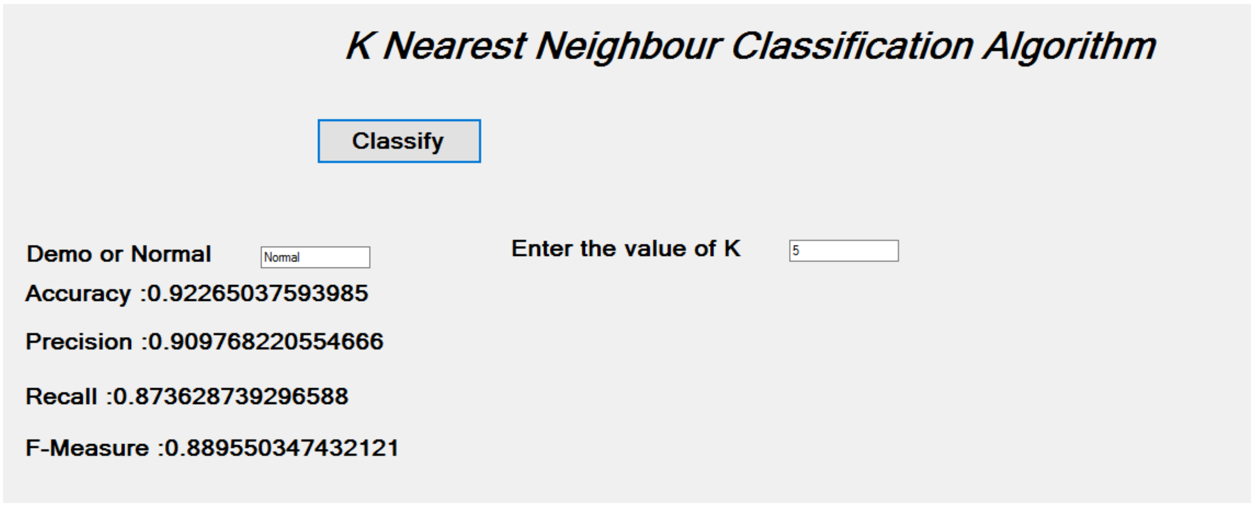
```
var items = from pair in dm orderby pair.Value ascending select pair; //ascending order
double wt0 = 0;
double wt1 = 0;
int count = 0;
foreach (KeyValuePair<string, double> pair in items)
{
    string[] p = pair.Key.Split(',');
```

```
        int cls = cl[Convert.ToInt32(p[1])];
        double wt = 1.0 / (pair.Value * pair.Value); //normalizing
        //adding the weights of k nearest neighbor’s belonging to the same class
        if (cls == 0)
        {
            wt0 = wt0 + wt;
        }
        else
        {
            wt1 = wt1 + wt;
        }
        count++;
        if (count == k) //K nearest neighbors
        {
            break;
        }
    }
    if (wt0 > wt1) //classify based on the class with more weight
    {
       assign[i] = 0;
    }
    else
    {
       assign[i] = 1;
    }
}
```

Output:

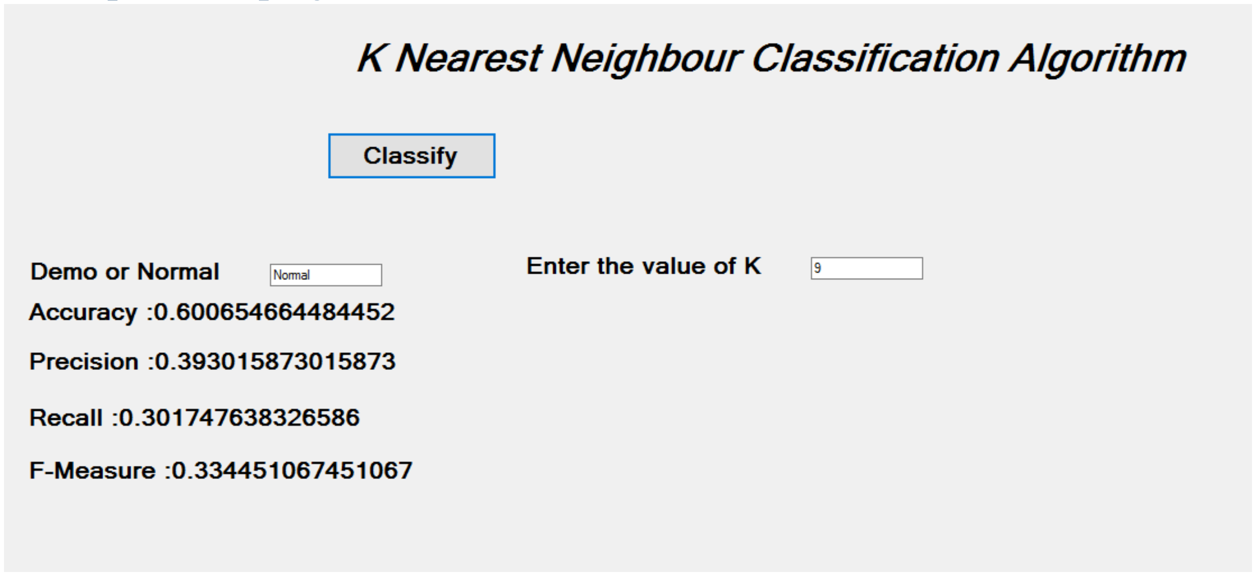
In this case the value of K was taken as 9 as it was giving the optimal results for all the datasets

For Input file:” project3_dataset1.xlsx”



```
Accuracy :0.92265037593985
Precision :0.909768220554666
Recall :0.873628739296588
F-Measure :0.889550347432121
```

For Input file:" project3_dataset2.xlsx"



Accuracy :0.600654664484452

Precision :0.393015873015873

Recall :0.301747638326586

F-Measure :0.334451067451067

Decision Tree Classification Algorithm

Decision tree learning uses a decision tree as a predictive model which maps observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves). It is one of the predictive modelling approaches used in statistics, data mining and machine learning. Tree models where the target variable can take a finite set of values are called classification trees. In these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels.

It breaks down a dataset into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed. The final result is a tree with decision nodes and leaf nodes. A decision node has two or more branches (e.g., Sunny, Overcast and Rainy). Leaf node represents a classification or decision. The topmost decision node in a tree which corresponds to the best predictor is called root node. Decision trees can handle both categorical and continuous data. In our implementation we have developed decision tress with two way split i.e. two branches.

Tree Induction starts with creating a null tree and then stopping condition is checked for the training data. If it is met then node is classified as leaf node. After that best split is determined for each of the feature. Among all the features, find the best feature to split and split S into S1 and S2. Then recursively repeat above mentioned steps. There are chances of overfitting in this algorithm so it is handled by using pre-pruning technique. So above mentioned algorithm stops as soon as it finds that all the nodes left are of same class or all the remaining attribute values are same.

Code Snippet 1: Decision Tree Creation

```
private void partition(m_tree S, int[] cl, double[,] fv, Dictionary<int, int>
col_string_dic)
        {

            if (S.index_name.Length == 2) ))//Leaf Node condition
            {
                S.leaf_node = true;
                S.value = cl[S.index_name[1]];
                return;
            }
            int p = 2;
            for (int i = 2; i < S.index_name.Length; i++)
            {
                if (cl[S.index_name[i]] != (cl[S.index_name[i - 1]]))
                {
                    break;
                }
                else p++;
            }
            if (p == S.index_name.Length) //If the class of all the sub records are the
same then return
            {
                S.leaf_node = true;
                S.value = cl[S.index_name[1]];
                return;
            }
            S1 = best_split(S.index_name, cl, fv, col_string_dic);
            int entry;
            if (col_string_dic.TryGetValue(Convert.ToInt32(S1[0]), out entry))//Diccrete
Attribute condition
            {
                for (int i = 1; i < S.index_name.Length; i++)
                {

                    if (fv[S.index_name[i], Convert.ToInt32(S1[0])] != S1[1])
                    {
                        S_left[count_left] = S.index_name[i];
                        count_left++;
                    }
                    else
                    {
                        S_right[count_right] = S.index_name[i];
                        count_right++;
                    }
                }
            }
            Else      //Continuous attribute condiation
            {
                for (int i = 1; i < S.index_name.Length; i++)
                {

                    if (fv[S.index_name[i], Convert.ToInt32(S1[0])] <= S1[1])
                    {
                        S_left[count_left] = S.index_name[i];
                        count_left++;
                    }
                    else
                    {
                        S_right[count_right] = S.index_name[i];
                        count_right++;
                    }
                }
            }

            S.index_right = S_right;
            S.column = Convert.ToInt32(S1[0]);
            S.value = S1[1];
            m_tree s_tree_left = new m_tree(0, 0, S.index_left);
            m_tree s_tree_right = new m_tree(0, 0, S.index_right);
            S.lnode = s_tree_left;
            S.rnode = s_tree_right;
            if (S.lnode != null)
            {
                partition(S.lnode, cl, fv, col_string_dic);//Partition on left child node
            }
            if (S.lnode != null)
            {
              partition(S.rnode, cl, fv, col_string_dic); //Partition on Right child node
            }
            return;
        }
```

There are many measures that can be used to determine the best way to split the records like Entropy, GINI etc. We have used GINI in our implementation. As there are multiple attributes corresponding to data, data is tested using GINI at each attribute and split at minimum value of GINI is taken into consideration. As there are continuous and nominal data present in the datasets, we have to use different approach to calculate GINI for both attributes.

For Continuous attributes, initially all the records present is taken into consideration for a particular attribute. Then for that attribute we have to find a value to split which would be either "Less than Equal to" (<=) or "Greater than" (>) a particular value. It would take a huge amount of time to calculate it using brute-force approach, so we sorted data in ascending manner and considered the average of those two adjacent records which differs in value. It significantly reduced the runtime complexity of algorithm.

Code Snippet 2: Function Get_Gini To Get GINI For Continuous Feature

```
private double[] get_gini(int ind, int[] B, int[] cl, double[,] fv)
      {
          for (int i = 1; i < B.Length; i++)
          {
              fv_temp[i, 0] = fv[B[i], ind];
              fv_temp[i, 1] = B[i];
              fv_temp[i, 2] = cl[B[i]];
          }

          System.Data.DataTable dt = new System.Data.DataTable();
          // assumes first row contains column names:

          // load data from string array to data table:
          for (int rowindex = 1; rowindex < fv_temp.GetLength(0); rowindex++)
          {
              DataRow row = dt.NewRow();
              for (int col = 0; col < fv_temp.GetLength(1); col++)
              {
                  row[col] = fv_temp[rowindex, col];
              }
              dt.Rows.Add(row);
          }
          // sort by first column:
          DataRow[] sortedrows = dt.Select("", "RowValue");

          for (int i = 1; i < B.Length; i++)
          {
              //check previous value and proceed only if it is different
              if (i > 0 && sort_arr[i, 0] != sort_arr[i - 1, 0])
              {
                  double v = (sort_arr[i, 0] + sort_arr[i - 1, 0]) / 2;
                  double l0 = 0, l1 = 0, g0 = 0, g1 = 0;
                  for (int j = 1; j < B.Length; j++)
                  {
                      if (sort_arr[j, 0] <= v)
                      {
                          if (sort_arr[j, 2] == 0)
                          {
                              l0++;
                          }
                          else
                          {
                              l1++;
                          }
                      }
                      else
                      {
                          if (sort_arr[j, 2] == 0)
                          {
                              g0++;
                          }
                          else
                          {
                              g1++;
                          }
                      }
```

```
                    }
                    double ginil = 1 - ((l0 / (l0 + l1)) * (l0 / (l0 + l1))) - ((l1 / (l0
+ l1)) * (l1 / (l0 + l1)));
                    double ginig = 1 - ((g0 / (g0 + g1)) * (g0 / (g0 + g1))) - ((g1 / (g0
+ g1)) * (g1 / (g0 + g1)));
                    double gini = (((l0 + l1) / (l0 + l1 + g0 + g1)) * ginil) + (((g0 +
g1) / (l0 + l1 + g0 + g1)) * ginig);
                    if (gini < min)
                    {
                        min = gini;
                        split = v;
                    }
                }

            }

            ret_arr[0] = min;
            ret_arr[1] = split;
            return ret_arr;
        }
```

For Nominal attributes, initially all the records present is taken into consideration for a particular attribute. Then for that attribute we have to find a value to split which would be either "Equal to" (==) or "Not Equal to" (!=) a particular value. So this process is repeated for all the possible nominal values.

Code Snippet 3: Function Get_Gini_String To Get GINI Foe A Discrete Feature

```
private double[] get_gini_string(int ind, int[] B, int[] cl, double[,] fv)
        {
            double[] ret_arr = new double[2];
            double min = 999999999;
            double split = 0;
            double[,] fv_temp = new double[B.Length, 3];

            for (int i = 1; i < B.Length; i++)
            {
                fv_temp[i, 0] = fv[B[i], ind];
                fv_temp[i, 1] = B[i];
                fv_temp[i, 2] = cl[B[i]];
            }


            string vstr = ";";
            for (int i = 1; i < B.Length; i++)
            {
                //check previous value and proceed only if it is different

                double v = fv_temp[i, 0];
                if (!(vstr.Contains(Convert.ToString(v))))
                {
                    vstr = vstr + ";" + Convert.ToString(v);

                    double l0 = 0, l1 = 0, g0 = 0, g1 = 0;
                    for (int j = 1; j < B.Length; j++)
                    {
                        if (fv_temp[j, 0] == fv_temp[i, 0])
                        {
                            if (fv_temp[j, 2] == 0)
                            {
                                l0++;
                            }
                            else
                            {
                                l1++;
                            }
                        }
                        else
                        {
                            if (fv_temp[j, 2] == 0)
                            {
                                g0++;
                            }
                            else
                            {
                                g1++;
                            }
                        }
                    }
//calculating Gini for Binary split
double ginil = 1 - ((l0 / (l0 + l1)) * (l0 / (l0 + l1))) - ((l1 / (l0 + l1)) * (l1 / (l0
+ l1)));

double ginig = 1 - ((g0 / (g0 + g1)) * (g0 / (g0 + g1))) - ((g1 / (g0 + g1)) * (g1 / (g0
+ g1)));

double gini = (((l0 + l1) / (l0 + l1 + g0 + g1)) * ginil) + (((g0 + g1) / (l0 + l1 + g0 +
g1)) * ginig);            //Gini  of the parent
                    if (gini < min)
                    {
                        min = gini;
                        split = v;
                    }
                }
            }

            ret_arr[0] = min;
            ret_arr[1] = split;
            return ret_arr;
        }
```

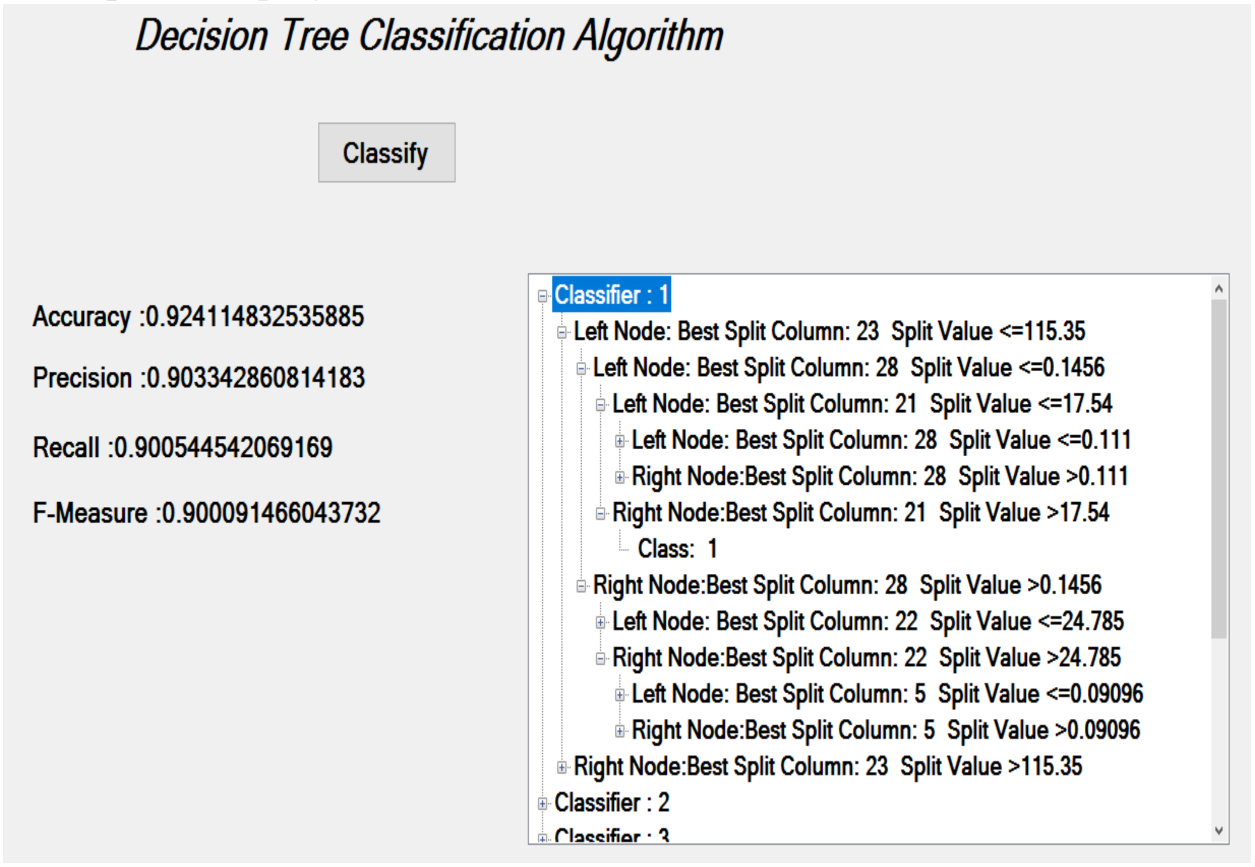
After GINI is calculated for all the attributes, value of best split and best feature is obtained for the minimum GINI.

Code Snippet 4: For Finding The Best Split

```
private double[] best_split(int[] A, int[] cl, double[,] fv, Dictionary<int, int>
col_string_dic)
        {
            double min = 999999999999;
            int col_best = 0;
            double split_value = 0;
            double[] ret_arr = new double[2];
            for (int i = 1; i < cols; i++)
            {

                int entry;
                double[] res;

                if (col_string_dic.TryGetValue(i, out entry))//// continues attr codn
                {
                    res = get_gini_string(i, A, cl, fv);//column number, list of index
                }
                else
                {
                    res = get_gini(i, A, cl, fv);//column number, list of index
                }

                    //0 index GINI value
                    //1 index split value

                if (res[0] < min)
                {
                    min = res[0];
                    col_best = i;

                    split_value = res[1];
                }
            }
            //0 column number on which split is taking place
            //1 value on which split is taking place
            ret_arr[0] = col_best;
            ret_arr[1] = split_value;
            return ret_arr;
        }
```

Output:

For Input file:" project3_dataset1.xlsx"



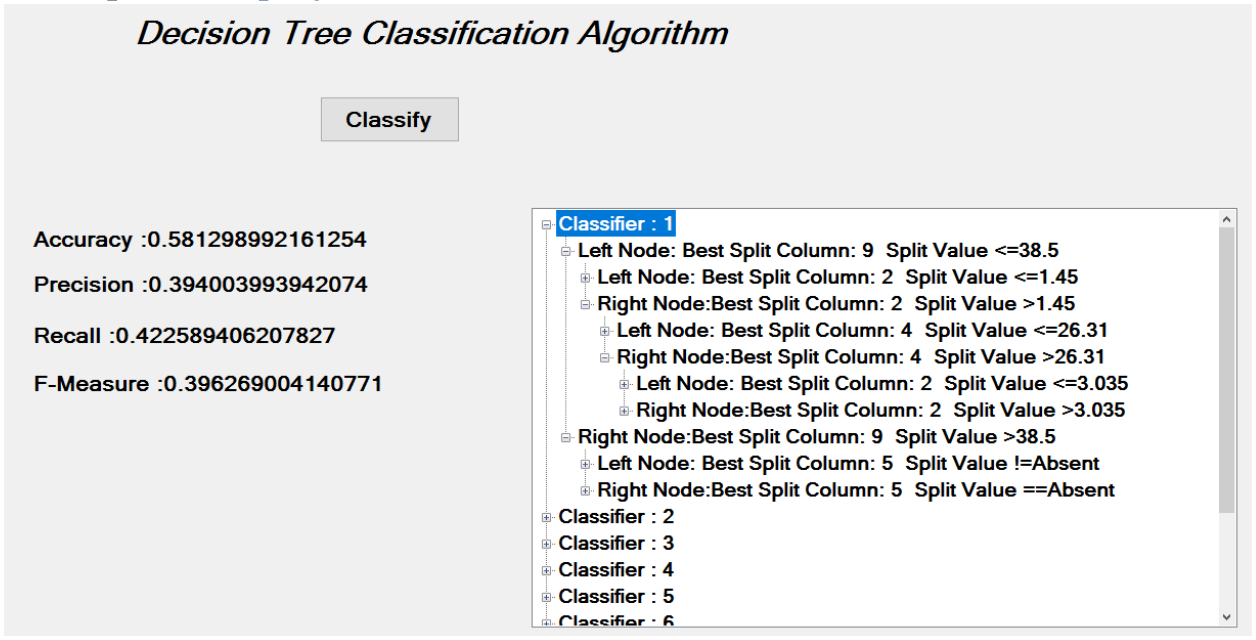
Decision Tree Classification Algorithm

```
Accuracy :0.924114832535885

Precision :0.903342860814183

Recall :0.900544542069169

F-Measure :0.900091466043732
```

```
Classifier : 1
  Left Node: Best Split Column: 23  Split Value <=115.35
    Left Node: Best Split Column: 28  Split Value <=0.1456
      Left Node: Best Split Column: 21  Split Value <=17.54
        Left Node: Best Split Column: 28  Split Value <=0.111
        Right Node:Best Split Column: 28  Split Value >0.111
      Right Node:Best Split Column: 21  Split Value >17.54
        Class:  1
    Right Node:Best Split Column: 28  Split Value >0.1456
      Left Node: Best Split Column: 22  Split Value <=24.785
      Right Node:Best Split Column: 22  Split Value >24.785
        Left Node: Best Split Column: 5  Split Value <=0.09096
        Right Node:Best Split Column: 5  Split Value >0.09096
  Right Node:Best Split Column: 23  Split Value >115.35
Classifier : 2
Classifier : 3
```

The best feature of a tree can be analyzed by simply looking at the tree as in the above scenario, best feature is 23th based on which 1st split occurs. Tree is designed in a way to display the continuous features as well as categorical features. As per our analysis of the data, continuous data generates much better results in the case of accuracy, recall and precision in comparison to categorical data.

For Input file:" project3_dataset2.xlsx"


Decision Tree Classification Algorithm

```
Accuracy :0.581298992161254
Precision :0.394003993942074
Recall :0.422589406207827
F-Measure :0.396269004140771
```

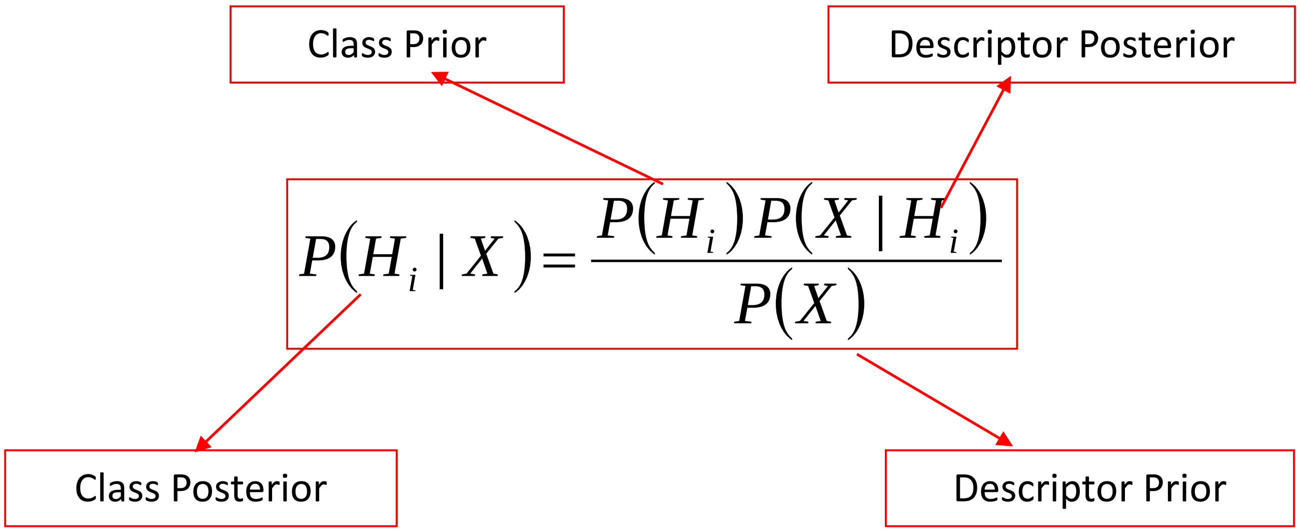
```
Classifier : 1
  Left Node: Best Split Column: 9  Split Value <=38.5
    Left Node: Best Split Column: 2  Split Value <=1.45
    Right Node:Best Split Column: 2  Split Value >1.45
      Left Node: Best Split Column: 4  Split Value <=26.31
      Right Node:Best Split Column: 4  Split Value >26.31
        Left Node: Best Split Column: 2  Split Value <=3.035
        Right Node:Best Split Column: 2  Split Value >3.035
  Right Node:Best Split Column: 9  Split Value >38.5
    Left Node: Best Split Column: 5  Split Value !=Absent
    Right Node:Best Split Column: 5  Split Value ==Absent
Classifier : 2
Classifier : 3
Classifier : 4
Classifier : 5
Classifier : 6
```

The best feature in this dataset is feature 9.

Naïve Bayes Classification Algorithm

Naïve Bayes classification is based on Bayes Theorem which works on estimating posterior probability. It works with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as 'Naive'.

In our implementation of Naïve Bayes Algorithm, Posterior probability is calculated using following.



$$P\left(H_i \mid X\right) = \frac{P\left(H_i\right) P\left(X \mid H_i\right)}{P\left(X\right)}$$

Where, -P(Hi|X) is posteriori probability of H conditioned on X i.e. the probability that dataset X belongs to class Ci given the attribute values of X.

-P(Hi) is class prior probability that X belongs to a particular class Ci.

-P(X) is prior probability of X i.e. the probability that observe the attribute values of X.

-P(X|Hi) is posterior probability of X given Hi is the Probability that observe X in class Ci.

P(Hi) class prior probability is calculated by dividing n -the total number of training data samples by ni -the number of training data samples of class Ci. This is handled on the main function of the code.

Code snippet 1 shows the calculation of P(X|Hi) in Descriptor Posterior Probability function for continuous and discrete features. In the case of continuous features Gaussian distribution to find the probability based on the mean and variance for the feature and use them in Gaussian distribution to find the class prior. Also the Zero-Probability Problem is handled using Laplacian correction.

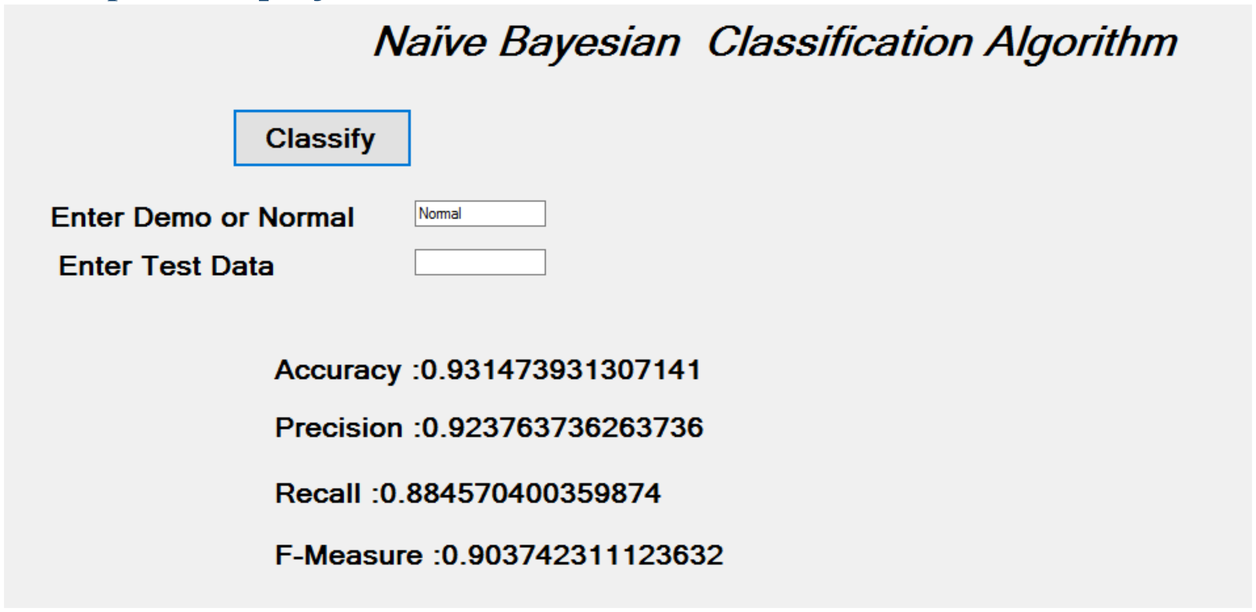
Code Snippet 1: Calculate Descriptor Posterior Probability

```
public double Descriptor_Posterior_Probability(double[,] fv, int[] cl, double[,] test_fv,
int r, int cls, Dictionary<string, int> val_string, Dictionary<int, int> col_string_dic,
double[,] gauss)
{
   double prob=1;
   for (int col = 1; col <= cols - 1; ++col)
   {
       int flag0 = 0;
       int entry;
       //Checking if the feature is continuous or not
       if (!(col_string_dic.TryGetValue(col, out entry)))
       {
          //in the case of continuous features use Gaussian distribution
          //to find the probability
                    double p=1;
                    p = test_fv[r, col] - gauss[col, 1];
                    p = Math.Pow(p, 2) / 2;
                    p = p / (Math.Pow(gauss[col, 2], 2));
                    p = Math.Exp(-p);
                    p = p / ((Math.Pow(Math.PI, 0.5)) * gauss[col, 2]);
                    if (p == 0)
                    {
                        flag0 = 1;
                    }
                    else
                    {
                        prob = prob * p;
                    }

                    p = 1;
       }
    else if (col_string_dic.TryGetValue(col, out entry))
    {
      /// In the case of discrete features

                    double p = 1;
                    string key = col + "_" + test_fv[r, col];
                    double s = 0, scls = 0; ;
                    for (int row = 1; row <= r_rows; ++row)
                    {
                       if (cl[row] == cls)
                       {
                           scls++;
                           if (test_fv[r, col] == fv[row, col])
                           {
                                s++;
                           }
                        }
                    }
                    p = s / scls;
                    if (p == 0)
                    {
                        flag0 = 1;
                    }
                    else
                    {
                         prob = prob * p;
                    }
                    p =1;
       }
       /// the Zero-Probability Problem is handled using Laplacian correction.
                if (flag0 == 1)
                {
                    string key = col + "_" + test_fv[r, col];
                    string k = ";";
                    int[] val = new int[50];
                    double v = 0;
                    foreach (string key_1 in val_string.Keys)
                    {
                        if (key_1.Contains(col + "_") && (!k.Contains(key_1)))
                        {
                            k = k + "_" + key + 1;
                            v++;

                        }
                    }

                        prob = prob * (1 / (r_rows + v));
                    flag0 = 0;
                }
            }
            return prob;

        }
```

Since, P(X) is prior probability of X is the same for both the classes, and comparison of posteriori probability P(Hi|X) between the 2 classes by dividing the product of P(X|Hi) * P(Hi) with P(X) is the same as comparing P(X|Hi) * P(Hi) without dividing it by P(X).

Output:

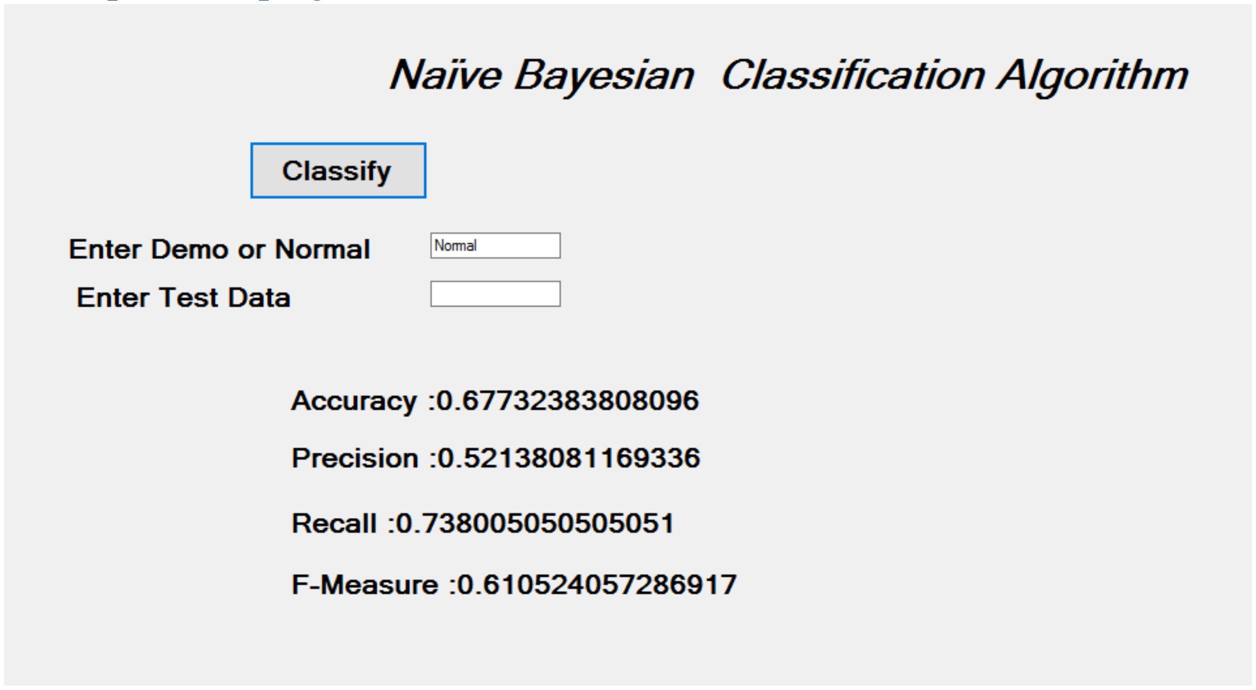
For Input file:" project3_dataset1.xlsx"



Enter Demo or Normal: Normal
Enter Test Data:

```
Accuracy :0.931473931307141

Precision :0.923763736263736

Recall :0.884570400359874

F-Measure :0.903742311123632
```

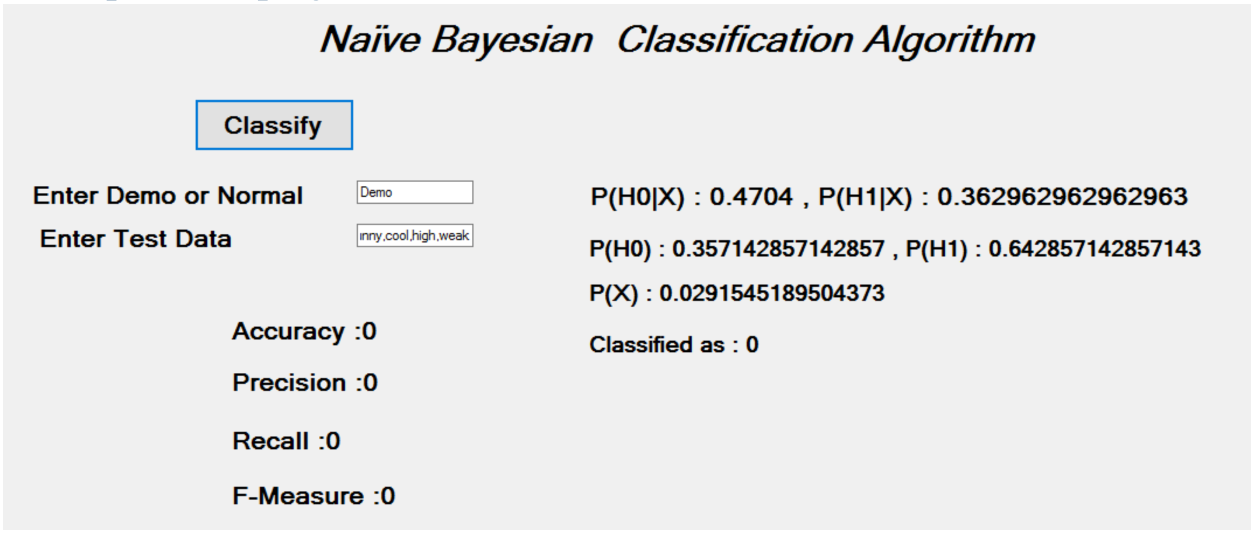
For Input file:" project3_dataset2.xlsx"



Enter Demo or Normal: Normal
Enter Test Data:

```
Accuracy :0.67732383808096

Precision :0.52138081169336

Recall :0.738005050505051

F-Measure :0.610524057286917
```

For Input file:" project3_dataset4.xlsx"


Naïve Bayesian Classification Algorithm

Enter Demo or Normal: Demo
Enter Test Data: inny,cool,high,weak

Accuracy :0

Precision :0

Recall :0

F-Measure :0

P(H0|X) : 0.4704 , P(H1|X) : 0.362962962962963

P(H0) : 0.357142857142857 , P(H1) : 0.642857142857143

P(X) : 0.0291545189504373

Classified as : 0

Decision Tree Classification Algorithm with Bagging and Random Forests

In this Implementation Decision Tree Classification is the same as the one explained before. The difference is that the training dataset is sampled in a different way using Bagging and Random Forest Predictive Modeling methods. They increase the accuracy and precision of the Classification algorithm they are inculcated in. The Bagging ensemble algorithm and the Random Forest algorithm support predictive modeling. The Bagging algorithm creates multiple different models from a single training dataset.

The Random Forest algorithm that makes a small tweak to Bagging and results in a very powerful classifier.

Code snippet 1 shows how we have implemented Bagging. While reading the sample of input data with randomness was considered as training data. Usually about %63.2 was kept as the original data. The rest of the data was used in testing datasets.

This is repeated untill the “Bag” iterations are completed. In each iteration, the sample testing datasets are different. This is the same for training datasets.

Code Snippet 1: Bagging

```
while (rows_counter_2 < bag)
                {
                    int t_2 = Convert.ToInt32(.632 * data_r_rows);
                    int[] ind_2 = new int[t_2+1];
                    Random r = new Random();
                    int cluster_count;
                    for (int i = 0; i <= t_2; i++)
                    {
                        cluster_count = r.Next(1, data_r_rows);
                        ind_2[i] = cluster_count;

                    }
```

Here, we randomly selected indexes from the input dataset, stored the indexes in an array and using it created a sample of as our training dataset. The indexes that weren’t there randomly selected were used to make the testing dataset.

In random forests, we sample on the features of the dataset. As done in Bagging, here instead of the sampling the records, we sample based on the features. The same logic as done in bagging is applied here. Random feature indexes are selected in each bagging iteration for classification. The same can be observed in code snippet 2.

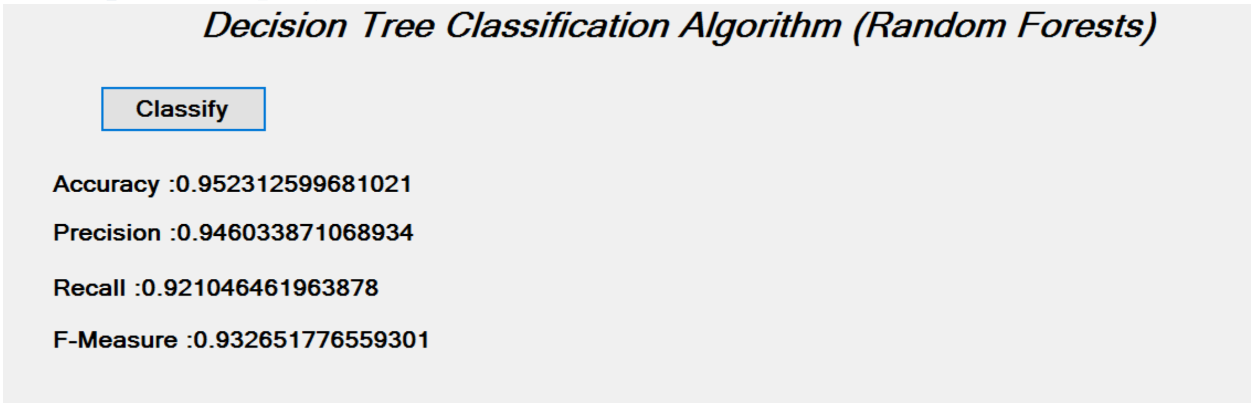
Code Snippet 2: Random Forests

```
string ind_3 = ",";
            Random n = new Random();
            int col_count;
            for (int i = 0; i <= r_cols; i++)
            {
                col_count = n.Next(1, cols);
                if (!(ind_3.Contains(Convert.ToString(col_count))))
                {
                    ind_3 = ind_3 + "," + col_count + ",";
                }
                else
                {
                    i--;
                }

if (ind_3.Contains("," + i + ","))
                {
                    int entry;
                    double[] res;

                    if (col_string_dic.TryGetValue(i, out entry))//// continues attr codn
                    {
                        res = get_gini_string(i, A, cl, fv);//column number, list of
index
                    }
                    else
                    {
                        res = get_gini(i, A, cl, fv);//column number, list of index
                    }
```

The main aim here is to apply a combination of classifiers in order to increase the accuracy, precision and decrease the prediction error.

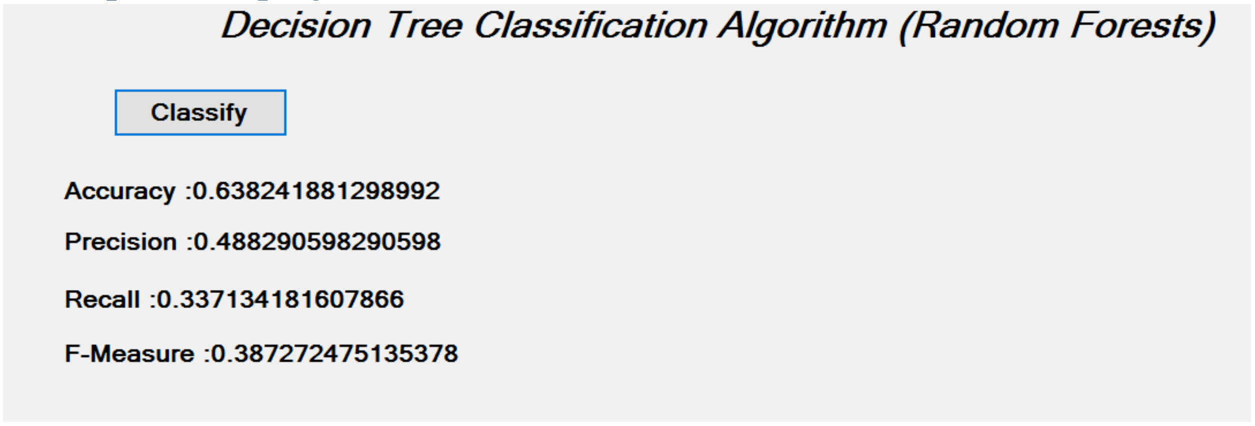
Output:

For Input file:" project3_dataset1.xlsx"



```
Accuracy :0.952312599681021
Precision :0.946033871068934
Recall :0.921046461963878
F-Measure :0.932651776559301
```

For Input file:" project3_dataset2.xlsx"

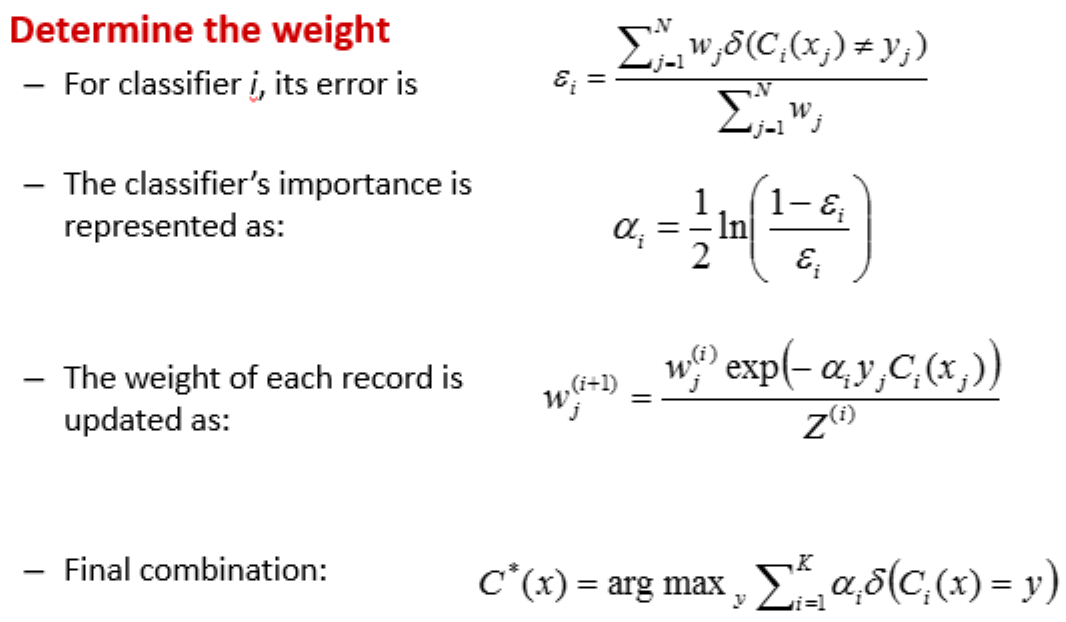


```
Accuracy :0.638241881298992
Precision :0.488290598290598
Recall :0.337134181607866
F-Measure :0.387272475135378
```

Decision Tree Classification Algorithm with Boosting

Similar to the previous Random Forest Technique, we implement boosting by using decision tree as the classifier. The working of the classifier is the same except in the case of boosting using weights, the records with higher weights are given higher preference. The main functionality of boosting algorithms is to transform weak learning classifiers to a strong learners. In our implementation, Adaboost algorithm as it could adapt to weak learners faster. In Adaboost the weak learners are tweaked to favor misclassified records. There by making the subsequent classifiers to concentrate more on the misclassified ones.

Code snippet 1 shows our implementation of Adaboost. We have calculated the weights as shown below.



Determine the weight

– For classifier *i*, its error is

$$\varepsilon_i = \frac{\sum_{j=1}^{N} w_j \delta(C_i(x_j) \neq y_j)}{\sum_{j=1}^{N} w_j}$$

– The classifier's importance is represented as:

$$\alpha_i = \frac{1}{2} \ln\left(\frac{1-\varepsilon_i}{\varepsilon_i}\right)$$

– The weight of each record is updated as:

$$w_j^{(i+1)} = \frac{w_j^{(i)} \exp\left(-\alpha_i y_j C_i(x_j)\right)}{Z^{(i)}}$$

– Final combination:

$$C^*(x) = \arg\max_y \sum_{i=1}^{K} \alpha_i \delta\left(C_i(x) = y\right)$$

Code Snippet 1: Boosting- Error Calculation

```
not_same = not_same/(not_same + same);
if (not_same >= 0.5)
{
    break;
}
if (not_same == 0)
{
      break;
}
```

The above code was for error calculation and error handling based on the classification of the classifier.

Code Snippet 2: Boosting- Reassigning of Weights.

```
alpha[rows_counter_2] =  (Math.Log((1 - not_same) / not_same))/2;
double c_correct = Math.Exp((-1) * alpha[rows_counter_2]);
double c_incorrect = Math.Exp((1) * alpha[rows_counter_2]);
//reassign appropriate weight
  for (int i = 1; i < t_rows; i++)
  {
     if (assign[i] == test_cl[i])
     {
data[Convert.ToInt32(test_fv[i,0]),cols]=data[Convert.ToInt32(test_fv[i,0]),cols]*c_corre
ct;
     }
     else
     {
     data[Convert.ToInt32(test_fv[i, 0]), cols] = data[Convert.ToInt32(test_fv[i, 0]),
cols] * c_incorrect;
     }
  }
```

The above code was the weights were reassigned based on the error calculated. “Alpha” vector the prediction value is stored. The weights for misclassified records are increased, and weights of correctly classified records are decreased.

Code Snippet 2: Boosting- Normalization of Weights.

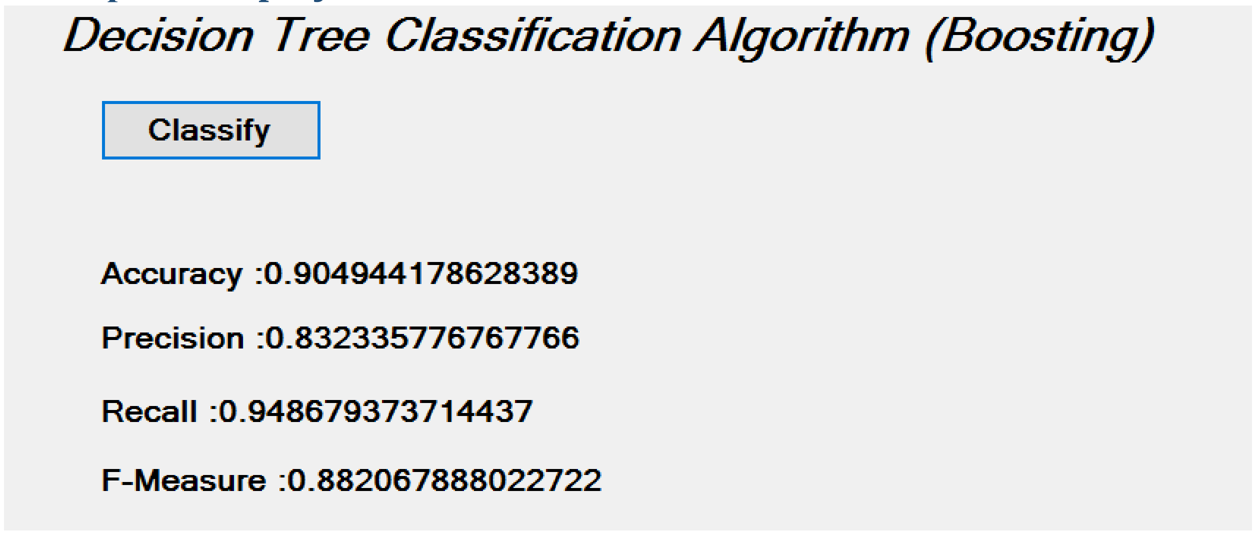
```
                    //normalize weight
                    double norm_wt = 0;
                    for (int i = 1; i < test_fv.GetLength(0); i++)
                    {
                        norm_wt += data[Convert.ToInt32(test_fv[i, 0]), cols];
                    }
                    for (int i = 1; i < test_fv.GetLength(0); i++)
                    {
                        data[Convert.ToInt32(test_fv[i, 0]), cols] =
data[Convert.ToInt32(test_fv[i, 0]), cols]/norm_wt;
                    }
```

Since the weights were changed, it needs to be classified.

Once the specified number of iteration are carried out, after which the classifier has become a better learner, then the last classifier predicts the data by taking the average of the all the classifier outputs, based on their weights.

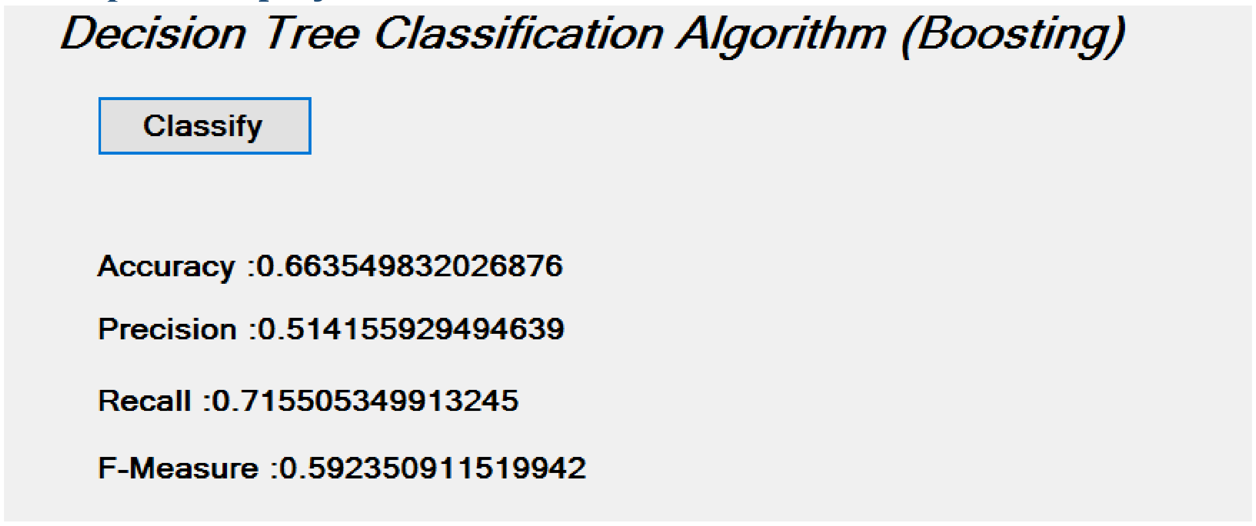
Output:

For Input file:” project3_dataset1.xlsx”



```
Accuracy :0.904944178628389

Precision :0.832335776767766

Recall :0.948679373714437

F-Measure :0.882067888022722
```

For Input file:” project3_dataset2.xlsx”



```
Accuracy :0.663549832026876

Precision :0.514155929494639

Recall :0.715505349913245

F-Measure :0.592350911519942
```

Validation based performance measures

We have done validation on the testing dataset. Based on their classification results, we calculated Accuracy, Precision, Recall and F-Measure. On the basis of classification, confusion matrix is created by calculating following terms.

- **True positives (TP)**: These are cases in which we predicted yes, and prediction is correct.
- **True negatives (TN)**: We predicted no, and prediction is correct.
- **False positives (FP)**: We predicted yes, but prediction is wrong. (Also known as a "Type I error.")
- **False negatives (FN)**: We predicted no, but prediction is wrong. (Also known as a "Type II error.")

Using this Accuracy, Precision, Recall and F-Measure are calculated as described following.

- **Accuracy**: Overall, how often is the classifier correct?
  (TP+TN)/ (TP+TN+FP+FN)
- **Precision**: When it predicts yes, how often is it correct.
  TP/ (TP+FP)
- **Recall**: When it's actually yes, how often does it predict yes. Also known as TPR (True Positive Rate).
  TP/ (TP+FN)
- **F-Measure**: This is a weighted average of the true positive rate (recall) and precision
  2*Recall*Precision/ (Recall + Precision)

In the following Code snippet:3, the Comparison of the actual class of the testing data with the classification result is shown. Based on this the Accuracy, Precision, Recall and F-Measure are calculated.

Code Snippet : Comparing the actual class of the testing data with the classification result

```
double tp = 0, tn = 0, fp = 0, fn = 0;
for (int i = 1; i < t_rows; i++)
{
   if (test_cl[i] == 1)
   {
      if (assign[i] == 1)
      {
          tp++;
      }
      else if (assign[i] == 0)
      {
          fn++;
      }
   }
   else if (test_cl[i] == 0)
   {
       if (assign[i] == 1)
       {
           fp++;
       }
       else if (assign[i] == 0)
       {
           tn++;
       }
   }
 }
 double accuracy = 0, precision = 0, recall = 0, f_measure = 0;
 accuracy = (tp + tn) / (tp + tn + fp + fn);
 precision = (tp) / (tp + fp);
 recall = (tp) / (tp + fn);
 f_measure = (2 * recall * precision) / (recall + precision);
}
```

When the Jaccard Coefficient is 1, then it means the clustering was an exact match to the ground truth. Where as in the case of Rand Index 0 indicates that the two data clusters do not agree on any pair of points and 1 indicates that the data clusters are exactly the same.

Dataset1	Accuracy	Precision	Recall	F-Measure
KNN	0.922	0.9097	0.8736	0.8895
Decision Tree	0.924	0.903	0.905	0.9009
Naïve Based	0.9314	0.9237	0.8845	0.9037
Random Forest	0.9523	0.9460	0.9210	0.9326
Boosting	0.9049	0.8323	0.9486	0.8820

In Dataset1 all the features are continuous. Therefore we are getting high accuracy. By observing the above table, the best classification algorithm is Random Forest on the basis of accuracy. But boosting has the higher recall. Since the primary functionality of random forest and boosting is to increase the accuracy, recall and precision, it can be observed that in the case of only continuous attributes, accuracy is being increased in random forest while recall is increased in boosting. Naïve Based also works significantly high over here since a large number of data is here to process the probabilities and similarly this data is also suitable for KNN as it is giving more than average results.

Dataset2	Accuracy	Precision	Recall	F-Measure
KNN	0.600	0.3930	0.3017	0.3045
Decision Tree	0.5512	0.394	0.4225	0.3962
Naïve Based	0.6773	0.521	0.738	0.6105
Random Forest	0.6382	0.4883	0.3371	0.3872
Boosting	0.6635	0.5141	0.7155	0.5923

In the table for the second dataset which has a discrete feature, the above can be observed. Since the primary functionality of random forest and boosting is to increase the accuracy and precision, it can be observed both the tables. Since the primary functionality of random forest and boosting is to increase the accuracy, recall and precision, it can be observed that in the case of continuous and nominal attributes, accuracy is being increased in random forest while recall is increased in boosting. There is also a significant drop in the measured performance values in comparison to dataset 1 only because of the fact that dataset 2 has nominal attribute also.

Naïve Based also works significantly high over here since a large number of data is here to process the probabilities and similarly this data is also suitable for KNN as it is giving more than average results.

Analysis of Algorithms

KNN: Pros of KNN is that it is very simple to implement, flexible to feature choices and can do well in practice with enough representative data but some of its cons are that it takes a large amount of storage data, large search problem to find nearest neighbors, has scaling issues and are lazy learners.

Decision tree: Pros of decision tree classification is that it is inexpensive to construct, extremely fast for classifying unknown records and accuracy is very much comparable to other classification algorithms but its cons are that it is hard to interpret large sized trees and issues of over fitting and under fitting which can be sometimes dealt by pre-pruning and post-pruning.

Decision tree (Random Forest): It is faster and accurate than decision tree for almost all of the cases.

Decision tree (Boosting): It works accurately in case of recall but sometimes fails in accuracy when data is already classified properly as it involves more complex process and in the process hurts the accuracy.

Naïve Bayes: It is easier to implement and has less model complexity but it has no variable dependency, which is not much justified as real life data is dependent on each other.