CSE 601
Data Mining and Bioinformatics

Project 3
Classification Algorithms

Part 1: Nearest Neighbor
Part 2: Decision Tree
Part 3: Naïve Bayes
Part 4: Random Forest
Part 5: Kaggle Competition

Submitted By:
Karan Manchandia
UB Name: karanman
Person # 50290755

Divya Srivastava
UB Name: divyasri
Person # 50290383

Varsha Lakshman
UB Name: varshala
Person # 50288138

Objective:

The objective of this project is to perform the following four classification algorithms on the two datasets, “project3_dataset1.txt” and “project3_dataset2.txt”:

- Nearest Neighbor
- Decision Tree
- Naïve Bayes
- Random Forest
- The fifth part involves implementing any improvement over a combination of these algorithms, to classify the records in the dataset given on Kaggle.

Adopt 10-fold cross-validation to evaluate the performance of all methods on the provided two datasets. The performance measures to be evaluated are:

- Accuracy
- Precision
- Recall
- F-1 Measure

Input file details are as follows:

- Each line represents one sample data.
- The last column of each line is a class, either 0 or 1.
- The rest columns are feature values. Each of them can be a real value.

Definitions of Performance Measures (Common to all Algorithms):

Consider the table shown below to understand what is accuracy, precision, recall and f1-measure (here Yes/No resembles 1/0 for the given datasets):

	Predicted Classes		
Actual Classes		Class = Yes	Class = No
	Class = Yes	a (TP)	b (FN)
	Class = No	c (FP)	d (TN)

- True Positive(a) is the number of records for which the predicted class is “Yes” and the actual class was also “Yes”.
- False Negative(b) is the number of records for which the predicted class is “No” while the actual class was “Yes”.
- False Positive(c) is the number of records for which predicted class is “Yes” while the actual class was “No”.
- True Negative(d) is the number of records for which the predicted class is “No” and the actual class was also “No”.

The various performance measures are defined as follows:

- Accuracy(A): Number of correctly classified test records divided by the total number of test records.
  A = (a+d)/(A+b+c+d)

- Precision(P): Of the records classified as “Yes” how many records are actually “Yes”
  Precision(P): a/(a+c)

- Recall(R): Of the records that are actually “Yes”, how many are classified as “Yes”.
  Recall (R): a/(a+b)

- F1-measure(FM): Weighted Harmonic mean of precision and recall.
  F1-measure: 2RP/(R+P) = 2a/(2a+b+c)

Algorithm 1: K-Nearest Neighbor(KNN):

Introduction:

- The K-Nearest Neighbor algorithm is defined as a nonparametric lazy learning algorithm that is used to predict the class label of the test record in question based on feature similarity. In other words, this algorithm uses k – nearest neighbors to the test record and classify test records to the class to which maximum of its neighbors belong to.
- Since the algorithm does not use the training data to do any generalization, as most of the other classification algorithms do, there is no explicit training phase in the KNN algorithm. So it requires most of the training data when it needs to predict the class labels in the testing phase.
- To classify a test record, KNN algorithm finds the k nearest training points to the test point in the n-dimensional space. In our implementation, the distance used to measure k nearest training points to the test point is Euclidean distance. The test point is assigned the class label corresponding to the majority class labels among the k nearest training points.

Algorithm:

- The first step of the KNN algorithm is to measure the distance of every record in the test dataset to every record in the training dataset.
  - We use Euclidean Distance to measure these distances.
  - For using Euclidean Distance we have handled the categorical attributes. A simple method to do this is to assign 0, in case the value for attribute exactly match and 1 when they don't. (For Example: Abesnt/Present = 0/1). More granular distance schemes can also be used to handle categorical values.
- Sort the distance matrix and identify the k nearest neighbors from the training dataset for each record in the test dataset.
- Then, use class labels of the k-nearest training records for each test record to predict the class for a particular test record. For this take the maximum of the count of classes to which k-closest neighbors belong to. The predicted class for a particular test data is the class with maximum votes.

Implementation:

1. The first step involves importing libraries. The following libraries were imported:
   Numpy, Math, Operator, Scipy.spacial.Distance.
   (Note: In this Implementation description we go by defining the functions in the order in which they are executed. In the actual code you will find that the functions are defined first and are called in the last two cells.
2. The next step is to take input from the user for the following:
   - K (the number of nearest neighbor to be considered)
   - The number of cross-validation folds.
   - The data file name

3. In this step the function ‘data_read_process’ is called. The input for this function is just the file name this function takes the datafile, open it and create a matrix. This function also takes care of continuous and categorical values in the dataset. This function converts all the continuous values into float values and assigns integers for similar categorical values in the dataset. For example, for dataset2 this function assigns 0/1 in place of absent/present.
4. In this step we call the ‘knn_cvfolds’ function. The input for this function is the processed data that is returned by ‘data_read_process’ function, the number of cross validation folds and K. This function divides the processed data as per the number of cross validation folds into training and testing data sets and training and testing class labels. ‘Knn_cvfolds function also calls the ‘knn_algorithm’ function which returns accuracy, precision, recall and F1-measure. The ‘knn_cvfold’ function finally calculates the average of all accuracy, precision, recall and f1-measure coming from each fold.
5. The ‘knn_algorithm’ function is responsible for normalizing all the datasets and finding the euclidean distance between all the records in the test dataset to all the records in the training dataset, sorting the distances and finding the class label for the test records based on class labels of k-nearest training records. This function finally predicts the test class labels and calls the ‘aprf_performance’ function to calculate accuracy, precision, recall and F1-measure.
6. The ‘aprf_performance’ function takes the original test class labels and predicted test class labels as input and calculates accuracy, precision, recall and f1-measure for each fold.
7. There is an option for providing different training and testing data files as input from the user. This code is required for the demo data files. Here the train and test files are separately given as input to the ‘data_read_process’ function to get the processed_training_dataset and processed_test_dataset. Then, processed training dataset and processed test dataset and k is given as input to the ‘knn_algorithm’. The ‘knn_algorithm’ function returns accuracy, precision, recall and F1-measure.

Choice Description:

K:

In order to select K that is right for our data we ran the KNN algorithm several times with different values of K and choose the K value that is perfect for our data. The parameter tuning can be found in the result analysis section. Here are some important points about choosing K:

- If we decrease K to 1, our predictions become less stable because the classification is affected by noise.
- Also, it should be tried to not choose an even value of K because it may be a possibility the the K-nearest neighbors have the exactly same number (K/2) of records labeled oppositely. We usually take K an odd number to have a tiebreaker.
- As we increase the value of K our predictions become more stable due to majority voting and thus more likely to make accurate predictions(upto a certain point). Eventually, we begin to witness overfitting. It is at this point we know we have pushed the value of K too far.
- In the tuning section of the report we have plotted K V/S accuracy, precision, recall and F1- measure. We varied K from 2 to half the number of records in the increments of two.

Finally we choose K = 9 for dataset 1 and K = 25 for dataset 2 because we were getting most accurate and balanced performance indicators values for these K.

Distance:

- We used Euclidean distance as the distance metric. Since, Euclidean distance is used, so for continuous attributes after normalizing the test and training data, we just take the Euclidean distance.
- For categorical attributes, we want to take the dissimilarity which corresponds to assigning different numerical values for different string categorical values (For Example assigning 0/1 for Absent/Present). In case there are more then 2 categorical values for an attribute, for example high, low and medium for wind speed then the code will assign 1,0 and 0.5 respectively. We replace the categorical values in a similar way and then calculate Euclidean Distance between records.

Continuous Attributes:

- For handling the continuous attributes that attributes with continuous data are identified. In the next step we normalize all the columns in the training data that are having continuous attributes. For normalizing a column we calculate mean and standard deviation of a column, we then subtract mean from all the elements oin the column and divide each value by standard deviation of the column. After normalization, all of the attributes contribute the same while measuring distance.
- In case of processing the test data we first identify the continuous attribute then normalize the data, but in this case we consider the same mean and standard deviation that we considered for normalizing the training data.

Categorical Attributes:

To handle categorical values in the dataset we first identify the categorical attributes in the dataset by checking that the values represent a digit ot not. Once an attribute is market as categorical, we find the number of different values among all the records for that particular attribute. We then assign particular float values between 0 and 1 based on the attribute values. For example, if the attribute values are Absent/Present, we assign it 0/1. Other more granular schemes may also be used which would assign different weights among different values. For example, if the attribute values are high, low and medium, it may be assigned 1, 0 and 0.5 respectively.

Results:

(Note: results for some parameters can be found in this section, if you want to view the complete parameter tuning result you can find it in parameter tuning section of the report.)

<u>Project 3_dataset1.txt:</u>

The KNN results for K = 2, folds = 10 and dataset = project3_dataset1.txt are shown below:
Accuracy = 0.9472431077694236
Precision = 0.9519805214913912

```
Recall = 0.904963822978013
F-measure = 0.9253185376769523

The KNN results for K = 3, folds = 10 and dataset = project3_dataset1.txt are shown below:
Accuracy = 0.9613408521303256
Precision = 0.9734126984126983
Recall = 0.9207386490374694
F-measure = 0.9446110300644828

The KNN results for K = 5, folds = 10 and dataset = project3_dataset1.txt are shown below:
Accuracy = 0.9666353383458647
Precision = 0.9782791377420536
Recall = 0.9280553645799913
F-measure = 0.9506992132434624

The KNN results for K = 7, folds = 10 and dataset = project3_dataset1.txt are shown below:
Accuracy = 0.9613408521303256
Precision = 0.9780815092835555
Recall = 0.9150226565472833
F-measure = 0.9432906692034985

The KNN results for K = 9, folds = 10 and dataset = project3_dataset1.txt are shown below:
Accuracy = 0.9666353383458647
Precision = 0.9854219948849104
Recall = 0.9258877591865794
F-measure = 0.952928174937622

The KNN results for K = 11, folds = 10 and dataset = project3_dataset1.txt are shown below:
Accuracy = 0.9683897243107771
Precision = 0.9854219948849104
Recall = 0.9291135656381924
F-measure = 0.9547399808990482

The KNN results for K = 13, folds = 10 and dataset = project3_dataset1.txt are shown below:
Accuracy = 0.9683897243107769
Precision = 0.989769820971867
Recall = 0.9276619527349664
F-measure = 0.9563530038765361

The KNN results for K = 15, folds = 10 and dataset = project3_dataset1.txt are shown below:
Accuracy = 0.9648182957393482
Precision = 0.9893557422969188
Recall = 0.9188091388821527
F-measure = 0.9520405093059587
```

The KNN results for K = 17, folds = 10 and dataset = project3_dataset1.txt are shown below:
Accuracy = 0.9613095238095237
Precision = 0.9889880952380953
Recall = 0.9079267859409761
F-measure = 0.9458289501532231

Project3_dataset2.txt:
The KNN results for K = 5, folds = 10 and dataset = project3_dataset2.txt are shown below:
Accuracy = 0.6496453900709219
Precision = 0.49662393162393165
Recall = 0.3708340562945826
F-measure = 0.41295499853453366

The KNN results for K = 11, folds = 10 and dataset = project3_dataset2.txt are shown below:
Accuracy = 0.6785597381342063
Precision = 0.5474206349206349
Recall = 0.35459104491999227
F-measure = 0.4154160799423957

The KNN results for K = 17, folds = 10 and dataset = project3_dataset2.txt are shown below:
Accuracy = 0.7083469721767595
Precision = 0.6113141025641026
Recall = 0.37679776363986883
F-measure = 0.44969037423339848

The KNN results for K = 23, folds = 10 and dataset = project3_dataset2.txt are shown below:
Accuracy = 0.7186033824331697
Precision = 0.6642640692640693
Recall = 0.3705865625602468
F-measure = 0.4667525558601072

The KNN results for K = 25, folds = 10 and dataset = project3_dataset2.txt are shown below:
Accuracy = 0.7254228041462084
Precision = 0.7029761904761905
Recall = 0.3956692211297474
F-measure = 0.49170512112364867

The KNN results for K = 27, folds = 10 and dataset = project3_dataset2.txt are shown below:
Accuracy = 0.7190398254228042
Precision = 0.6782359307359307
Recall = 0.3913490456911509
F-measure = 0.4853234501630303

The KNN results for K = 29, folds = 10 and dataset = project3_dataset2.txt are shown below:
Accuracy = 0.7152209492635024
Precision = 0.67260101010101
Recall = 0.3689912280701755
F-measure = 0.4648548842461885

The KNN results for K = 31, folds = 10 and dataset = project3_dataset2.txt are shown below:
Accuracy = 0.7147845062738679
Precision = 0.6587770562770563
Recall = 0.3742543859649123
F-measure = 0.46659294674831636

The KNN results for K = 37, folds = 10 and dataset = project3_dataset2.txt are shown below:
Accuracy = 0.7186033824331697
Precision = 0.6893956043956044
Recall = 0.3665620782726046
F-measure = 0.46616191250058525

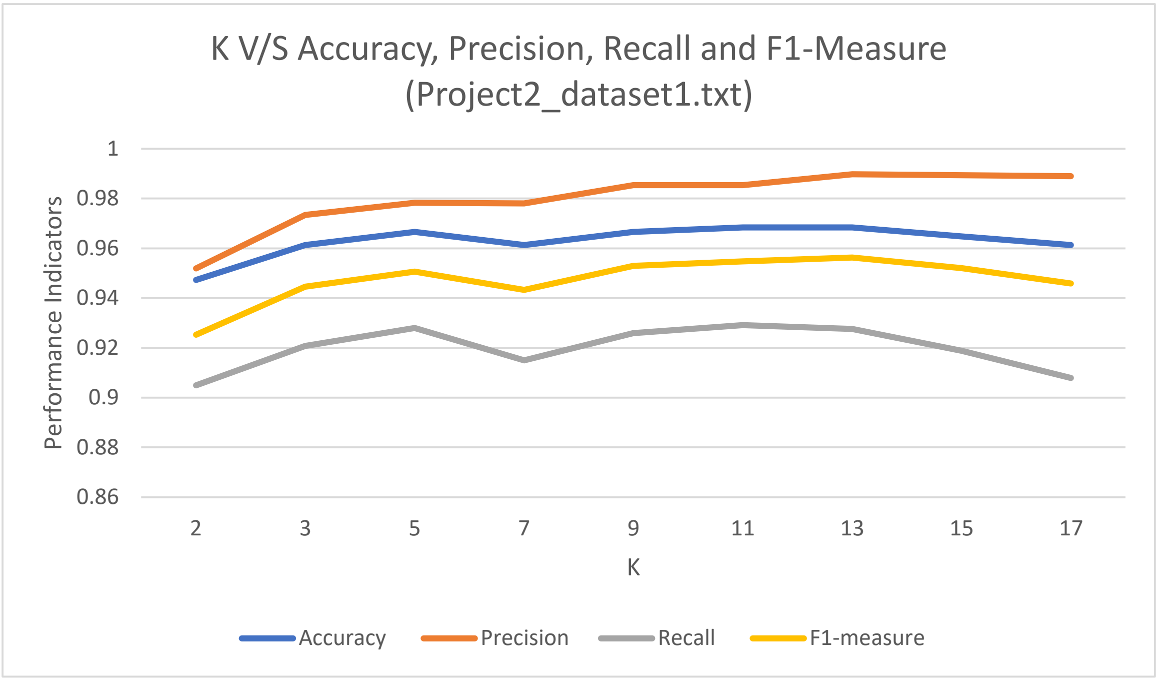
Result Analysis and Parameter Tuning:

<u>Parameter Tuning:</u>
We have changed the values of K and checked which value give the most balanced values of Accuracy, Precision, Recall and F1 Measure.

<u>Project3_dataset1.txt:</u>

K	Accuracy	Precision	Recall	F1-measure
2	0.9472431077694236	0.9519805214913912	0.904963822978013	0.9253185376769523
3	0.9613408521303256	0.9734126984126983	0.9207386490374694	0.9446110300644828
5	0.9666353383458647	0.9782791377420536	0.9280553645799913	0.9506992132434624
7	0.9613408521303256	0.9780815092835555	0.9150226565472833	0.9432906692034985
9	0.9666353383458647	0.9854219948849104	0.9258877591865794	0.952928174937622
11	0.9683897243107771	0.9854219948849104	0.9291135656381924	0.9547399808990482
13	0.9683897243107769	0.989769820971867	0.9276619527349664	0.9563530038765361
15	0.9648182957393482	0.9893557422969188	0.9188091388821527	0.9520405093059587
17	0.9613095238095237	0.9889880952380953	0.9079267859409761	0.9458289501532231

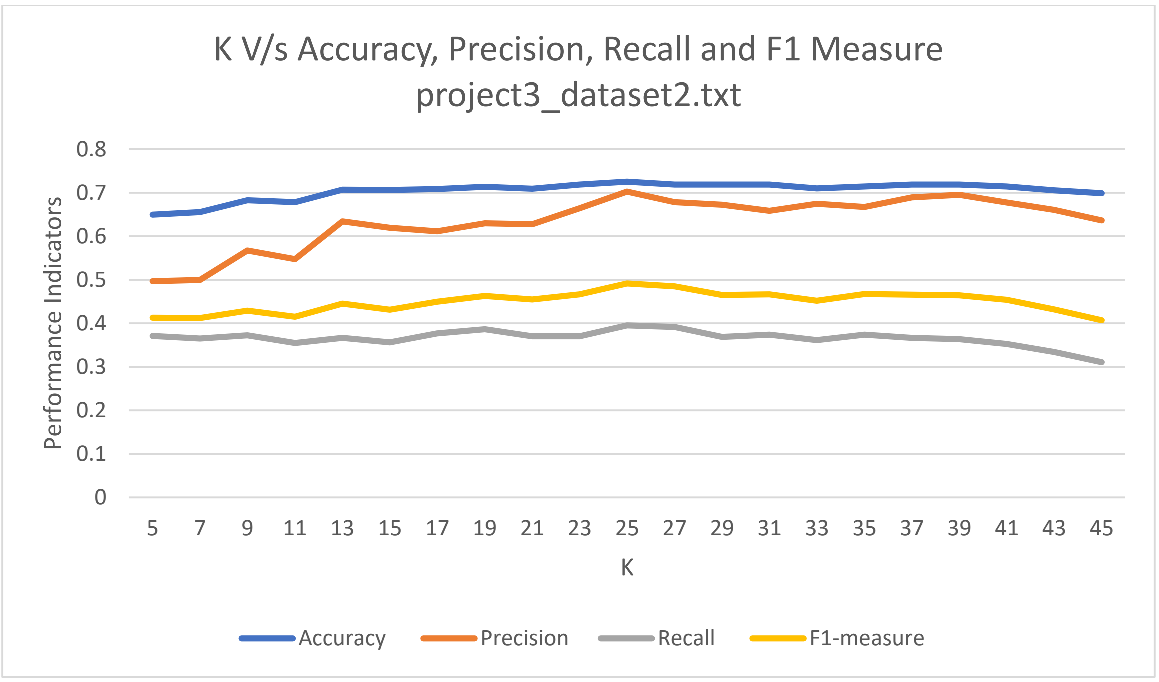
The graph plotted from the parameter tuning table is shown below:



<u>Project3_dataset2.txt:</u>

K	Accuracy	Precision	Recall	F1-measure
5	0.6496453900709219	0.49662393162393165	0.3708340562945826	0.41295499853453366
7	0.6555919258046918	0.5	0.364822151532677	0.4118722344723369
9	0.6828150572831424	0.5672960372960373	0.37241179872758823	0.42934535823149417
11	0.6785597381342063	0.5474206349206349	0.35459104491999227	0.4154160799423957
13	0.7070921985815604	0.6344172494172493	0.3668666859456333	0.44522176632662563
15	0.7066557555919258	0.619543512043512	0.35624638519375357	0.43170628154643503
17	0.7083469721767595	0.6113141025641026	0.37679776363986883	0.4496903742339848
19	0.713911620294599	0.6302192252192251	0.38619553691922104	0.46263470795817
21	0.7096563011456629	0.6275252525252525	0.37014483323693853	0.4548880381303643
23	0.7186033824331697	0.6642640692640693	0.3705865625602468	0.4667525558601072
25	0.7254228041462084	0.7029761904761905	0.3956692211297474	0.49170512112364867
27	0.7190398254228042	0.6782359307359307	0.3913490456911509	0.4853234501630303
29	0.7190398254228042	0.67260101010101	0.3689912280701755	0.4648548842461885
31	0.7190398254228042	0.6587770562770563	0.3742543859649123	0.46659294674831636
33	0.7100927441352973	0.6747835497835498	0.3612989203778677	0.45211457751503514
35	0.714784506273868	0.6672743922743922	0.3742543859649123	0.46747995118704494
37	0.7186033824331697	0.6893956043956044	0.3665620782726046	0.46616191250058525
39	0.7190398254228042	0.6951190476190476	0.3637280701754385	0.4646663359523771
41	0.7147845062738679	0.6773701298701298	0.3528007518796993	0.45421174436454564
43	0.7058374249863612	0.6606060606060605	0.3345821283979179	0.4324480616534946
45	0.6990180032733224	0.6367099567099566	0.3106260844418739	0.4072705091755435

The graph plotted from the parameter tuning table is shown below:



Result Analysis:

- From the graphs above we see that as we increase the value of K , our accuracy and balance between different performace indicators becomes stable upto we reach a certain point and then it starts decreasing.
- By seeing the graphs we can say that, the value of K for which we are getting good accuracy and most stable other performance indicators are:
    - K = 9, for project3_dataset1.txt
    - K = 25, for project3_dataset2.txt
- We can also conclude that the dataset with all continuous attributes is more suitable for the KNN algorithm. This is because we get a higher accuracy for dataset 1 than for dataset 2.
- We Don't train the KNN Classification model, so it is a lazy learning algorithm.
- KNN algorithm works best for smaller datasets then for larger datasets.

Cross Validation:

10-Fold Cross Validation is implemented for getting good estimates of performance measures of the algorithm. Below are the steps on how Cross Validation is implemented in the code:

- The first step is to find the size of the test dataset that is required for the ten fold cross-validation. We also randomly shuffle input dataset to avoid overfitting issues.
- As per the size of test set, we extract the test dataset. The remaining records other then test dataset are put in another numpy matrix and are called train dataset.
- Then we compute the performance metrics by performing KNN algorithm on this train and test datasets.

- In the next step we take the next fold of test data records and then the remaining records are considered as the training dataset and KNN is performed.
- The same precess is repeated 10 times until all the 10 segments of dataset are considered as the test dataset.
- In the final step, we calculate average of all 10 performance metrics calculate during each fold to get the final performance metrics.

Pros and Cons:

Pros:

- The k-nearest neighbor algorithm is simple and easy to implement.
- The algorithm requires only a little change to handle more than 2 final class labels.
- There is no need to build a separate model for training, tune several parameters or make additional assumptions.
- KNN allows us to choose the distance function that we want for each feature. For example, KNN gives the opportunity to consider euclidean distance in case of continuous features and hamming distance in case of categorical features.

Cons:

- KNN Algorithm gets significantly slower as the size of the dataset becomes large.
- KNN is computationally expensive as we need to calculate the distance for every test record with all the training records. Additionally, we also need to parse through all these distances to find the minimum once, in order to identify the k-nearest neighbors.
- Since we may be dealing with mixed types of attributes in the dataset, we must need to define proper distance function in order to get correct results.
- The performance of KNN classifier can drop significantly in case dataset contains noisy attributes. This is because KNN use distance as its defining metric and gives equal weights to each attribute.
- If k is chosen to be too small then the model may be sensitive to noise points and if k is chosen too large model may include points from other classes.

Algorithm 2: Decision Tree:

Objective:

Our task was to implement decision tree classification on project3_dataset1 and project3_dataset2 and adopt 10-fold cross validation scheme to evaluate performance on the two provided datasets in terms of accuracy, precision, recall and f1 measure.

Introduction:

- Decision tree learning is a method commonly used in data mining.
- A decision tree is a simple representation for classifying examples. The goal is to create a model that predicts the value of a target variable based on several input variables.
- We have a training set on which we build our model i.e. decision tree.
- We have a test set on which we test the accuracy of our model to predict the class for each record.
    - when the subset at a node has all the same values of the target variable, or when splitting no longer adds value to the predictions.

Algorithm and Implementation:

Step1: Take the filename as input from the user. Open the tab delimited text file and remove the last column from the file and store it as a numpy matrix named ‘data_file’. Create another file ‘out_file’ with the last column which contains the labels.

Step2: Now detect the **categorical variables** in the dataset and give them numerical values. We iterate through all the columns and check if any attribute is string using function ‘string_check’, if yes then append the column indices to a list ‘s_indices’. Now we iterate through all such columns in ‘s_indices’ and assign numerical values to the categorical attributes in that column.

Step-3: Adopted **10-fold cross validation** to evaluate the performance where the data is divided into 10 parts and we execute the algorithm 10 times such that each segment is treated as test and other segments as training set in successive iterations.

Step4: Now we find the **best split** i.e. the split node to be treated as our root node from the training set. For this we use ‘find_node’ function. Here we iterate through the rows and columns and split the data to ‘left_list’ and ‘right_list’. For every element of data, we find the left and right lists such that all the data less than that element lies in left_list and if greater than that element, then lies in right_list.

Step-5: Then we calculate the gini-index corresponding to that element and its left and right lists. If the gini index for this element is less that the minimum gini index calculated yet, then the element corresponding to the least gini index becomes the best node for split. This is checked for every element in the data and the lowest gini index found will correspond to the best node for split.

Step-6: This best node is considered as the root for the tree which consists of the following attributes- row number of the element, the element itself, column number and corresponding left and right lists.

Step-7: Now we build the decision tree using ‘decision_tree’ function. Here, if left list is not empty, we check if all the elements in left list belong to same class and find that class using ‘find_class’ function where the class is determined by the maximum count that is available for any class. The same way, we check the class of right list, if all the elements belong to the same class.

Step-8: Now if the elements does not belong to the same class, then we do a recursion on tree and pass the root as the best node available in the left and right trees respectively. We repeat the steps 7 and 8 until the stopping condition.

Step-9: We **stop** when all the elements are either in left list or all on right list and then find the corresponding class.

Step-10: Once the tree is made, we now use our test dataset to classify the labels using ‘test_classification’ function where we iterate through our tree and find the label for each record of our test dataset and append it to ‘test_class’ list which is our predicted_class.

Step-11: We now compare our predicted_class with the last column of our test_set i.e. actual_class and form the performance matrix through the function ‘performanceMetrics’ based on which, we calculate the accuracy, precision, recall and f1_measure which will be an average reading for all the 10 iterations of our 10-fold validation algorithm.

Choice Description:

Handling Categorical Features:

To detect the categorical features and process them, we follow step2 of our implementation. The details are as follows. We first iterate through all the columns of the first row of our dataset and apply ‘string_check’ function on each value which checks if the value at a given column in row 0 is string or not. If it is string, then we append its index to ‘s_indices’ list. Now for each of the columns whose index are in s_indices, we check the number of unique values in that column and assign them with numerical values, one numerical value for each unique categorical value and we write these original values and replacement values in dictionary. So, if the column has categorical values ‘present’ and ‘absent’, then they will be replaced with ‘0’ and ‘1’ respectively. Now when we check for the best split node, we check for these columns separately as since these are not continuous values, we cannot check for less than or greater than values to compose our left and right lists. Hence, if the value exactly matches with the element, then we put them in left list or else in right list. Then we calculate the gini index for each element in the dataset and find the minimum gini which corresponds to the node value for the best split. The same procedure is applied for classification of labels for test data where we calculate the labels for categorical data separately as we cannot use less than and greater than comparison operators here as well.

Handling Continuous Features:

To handle the continuous features, for each value in the dataset, we iterate through all the values of the dataset one by one and if the values are less than that particular value, then we append it to the left list and if greater, then we append it to the right list. Gini index for both the lists are calculated separately and then a combined gini index is calculated corresponding to that node. Likewise, gini index for each data element is calculated and the data for which gini is minimum, is considered as the node for the best split.

Best Feature:

For every data point in the dataset, after handling the continuous and categorical data separately, we find the gini index. Then out of all the gini indices calculated, we find the minimum gini index and the corresponding node is considered as the split node with least impurity as the gini index for this node is the minimum. This way the entire tree is constructed by repeating the above process such that at each level, we find the node with least gini index. Hence, we always choose the best feature which gives the least impurity at each node in our constructed decision tree.

Stop-Critera:

We continue finding the best split and building the tree until any of the following two conditions are met:

- If len(left_list) == 0 or len(right_list) == 0
    - This corresponds to the condition where all the elements are either in left list or all the elements are in right list and the other list becomes empty.
- If len(left_list) > 0 and if len(np.unique(left_list[:,-1])) == 1 and
    - if len(right_list) > 0 and if len(np.unique(right_list[:,-1])) == 1
    - This corresponds to a condition when all the elements of left list belong to the same class and all the elements of the right list belongs to the same class.

If any of the above two conditions are met, we stop building our tree further at that instance.

Pros and Cons:

<u>Pros:</u>

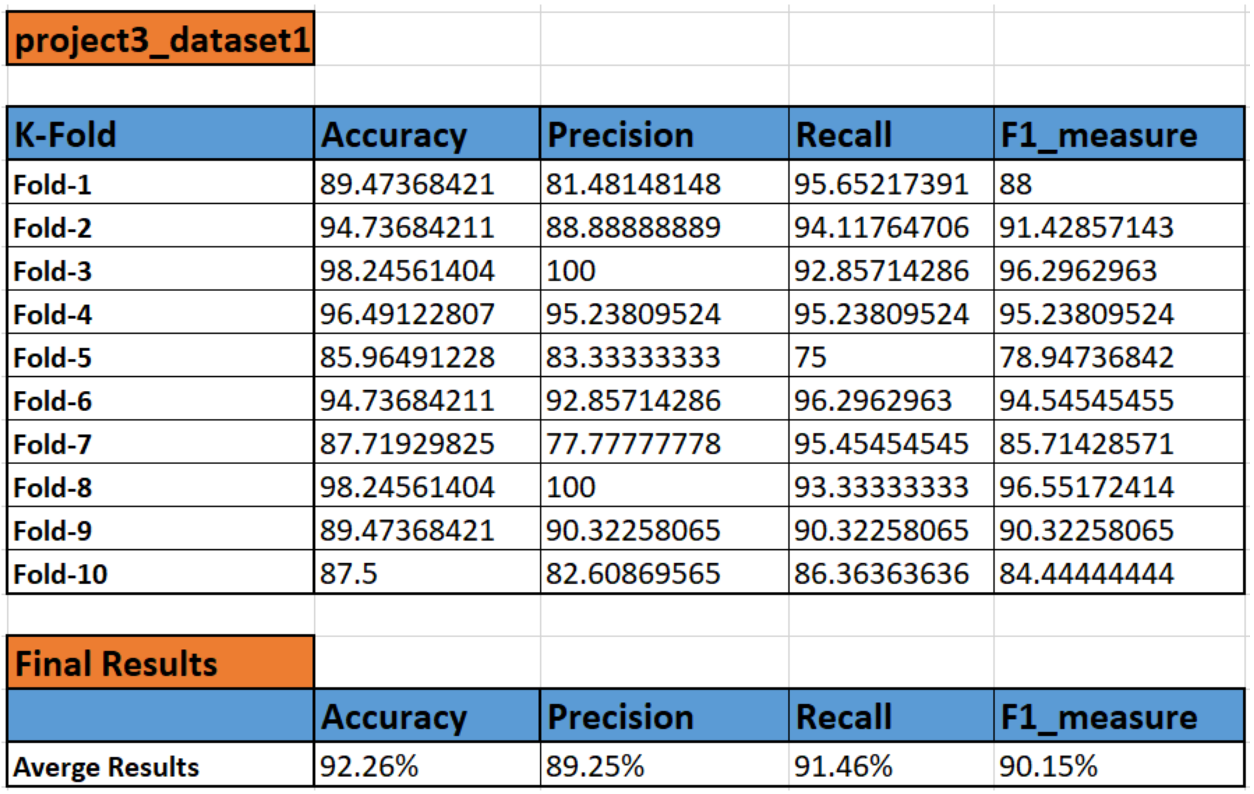
- Simple to understand and interpret.
- Able to handle both numerical and categorical data.
- Requires little data preparation.
- Uses a white box model i.e. the results can easily be derived and explained.
- Possible to validate a model using statistical tests.
- Performs well with large datasets.

<u>Cons:</u>

- Trees can be very non-robust. A small change in the training data can result in a large change in the tree and consequently the final predictions.
- Decision-tree learners can create over-complex trees that do not generalize well from the training data. This is known as overfitting.
- For data including categorical variables with different numbers of levels, information gain in decision trees is biased in favor of attributes with more levels.

Cross-validation:

We have implemented 10-fold cross validation in our implementation. This way we improve the efficiency of our algorithm as we train our dataset 10 times. We segment our dataset in 10 parts and keeping each segment as test and remaining dataset as train set for each iteration, we calculate the accuracy, precision, recall and f1_measure. We then take average of all the readings and those averaged values are our final accuracy, precision, recall and f1_measure.
Following are the results of each iteration for that fold:

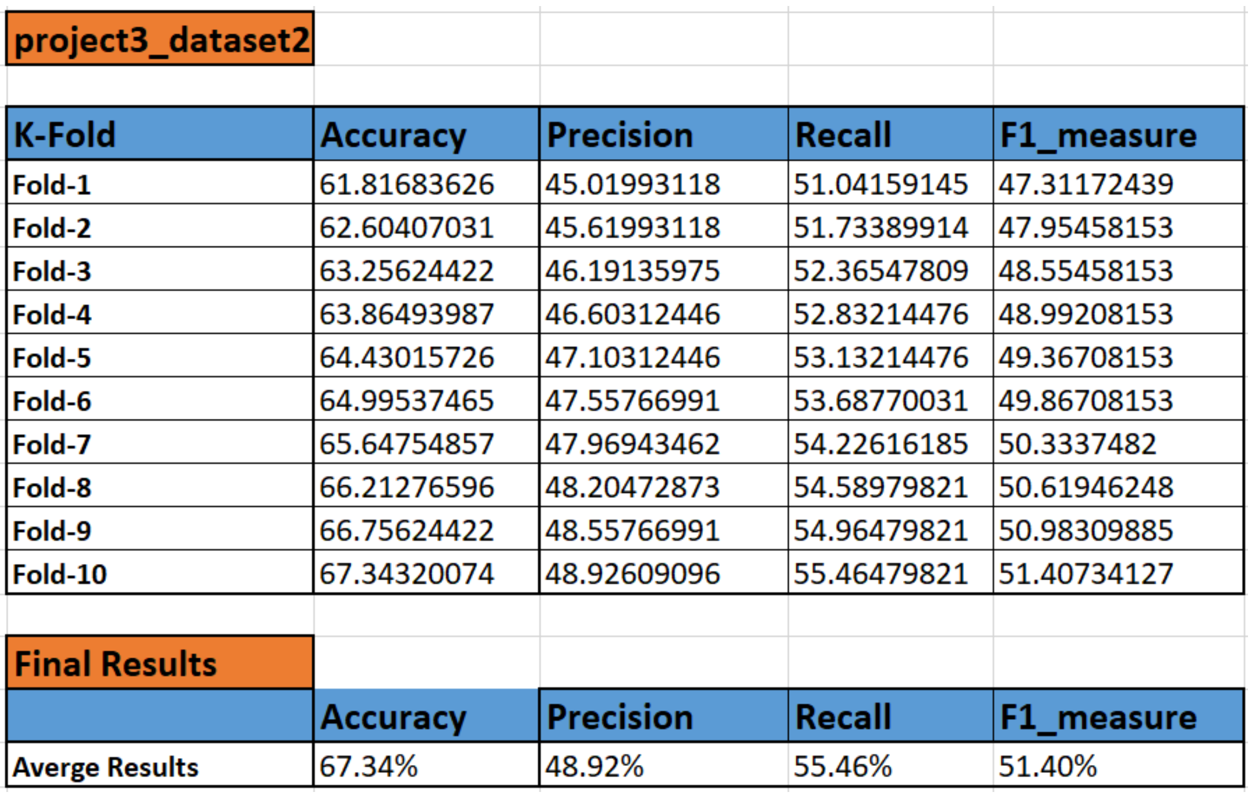


project3_dataset1

K-Fold	Accuracy	Precision	Recall	F1_measure
Fold-1	89.47368421	81.48148148	95.65217391	88
Fold-2	94.73684211	88.88888889	94.11764706	91.42857143
Fold-3	98.24561404	100	92.85714286	96.2962963
Fold-4	96.49122807	95.23809524	95.23809524	95.23809524
Fold-5	85.96491228	83.33333333	75	78.94736842
Fold-6	94.73684211	92.85714286	96.2962963	94.54545455
Fold-7	87.71929825	77.77777778	95.45454545	85.71428571
Fold-8	98.24561404	100	93.33333333	96.55172414
Fold-9	89.47368421	90.32258065	90.32258065	90.32258065
Fold-10	87.5	82.60869565	86.36363636	84.44444444

Final Results

	Accuracy	Precision	Recall	F1_measure
Averge Results	92.26%	89.25%	91.46%	90.15%



project3_dataset2

K-Fold	Accuracy	Precision	Recall	F1_measure
Fold-1	61.81683626	45.01993118	51.04159145	47.31172439
Fold-2	62.60407031	45.61993118	51.73389914	47.95458153
Fold-3	63.25624422	46.19135975	52.36547809	48.55458153
Fold-4	63.86493987	46.60312446	52.83214476	48.99208153
Fold-5	64.43015726	47.10312446	53.13214476	49.36708153
Fold-6	64.99537465	47.55766991	53.68770031	49.86708153
Fold-7	65.64754857	47.96943462	54.22616185	50.3337482
Fold-8	66.21276596	48.20472873	54.58979821	50.61946248
Fold-9	66.75624422	48.55766991	54.96479821	50.98309885
Fold-10	67.34320074	48.92609096	55.46479821	51.40734127

Final Results

	Accuracy	Precision	Recall	F1_measure
Averge Results	67.34%	48.92%	55.46%	51.40%

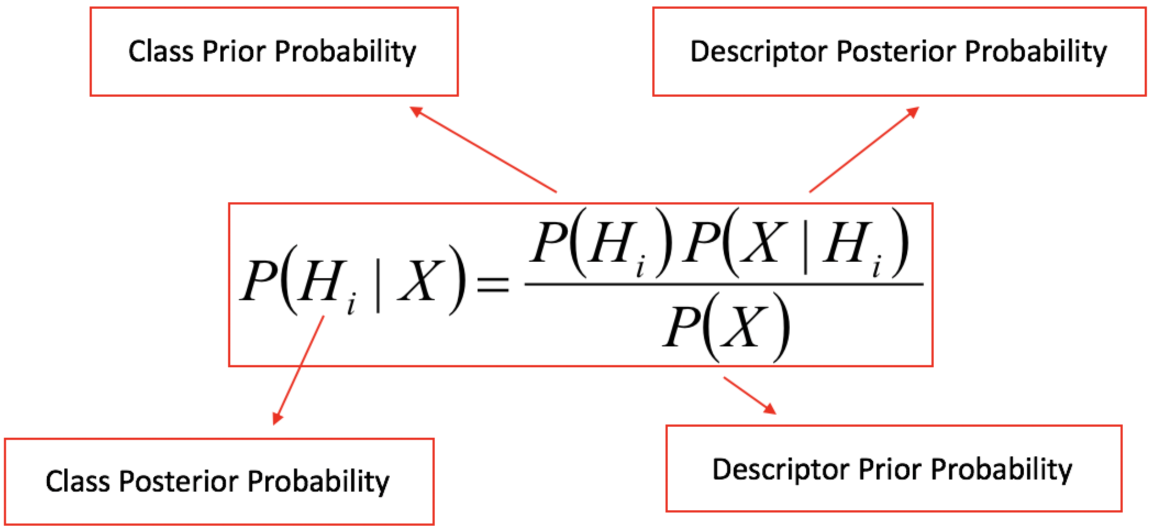
Result analysis:

- From the above results, we can see that the accuracy, precision, recall and F1 measure changes on every iteration of our 10-fold cross validation when we select different segments as test and train sets. Hence, on taking the average, we can determine a more accurate values for above parameters.
- We also observe that the performance of our tree model is good for a large dataset where it gets a higher training percent when compared to that of a smaller dataset. Hence we get the accuracy for dataset1 higher than that of dataset2.
- We can also conclude that KNN has a bit better performance parameters than decision tree.
- The reduced performance of decision tree can be because of overfitting issues which can be handled by pre-processing and post-processing techniques like pruning.
- The overall performance of a decision tree is better for a small dataset. Since our datasets used here are comparatively larger, the accuracy is lower as compared to KNN.

Algorithm 3: Naive Bayes:

Introduction:

Naive Bayes is a classification technique based on Bayes' theorem with an assumption of independence among predictors. It assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. Naive Bayes is a conditional probability model. It predicts the membership probabilities of class. It is represented using the below formula:


Class Prior Probability
Descriptor Posterior Probability
$P(H_i \mid X) = \frac{P(H_i) P(X \mid H_i)}{P(X)}$
Class Posterior Probability
Descriptor Prior Probability

Algorithm:

Bayes' Theorem finds the probability of an event occurring given the probability of another event that has already occurred. Bayes' theorem is stated mathematically as the above equation: where H and X are events and P(X)? 0.

- Basically, we are trying to find probability of event H, given the event X is true. Event X is also termed as **evidence**.
- P(A) is the **priori** of A (the prior probability, i.e. Probability of event before evidence is seen). The evidence is an attribute value of an unknown instance (here, it is event X).
- P(H|X) is a posteriori probability of X, i.e. probability of event after evidence is seen.

Implementation:

1. We are loading the datafile and fetching all the columns of the dataset except for the last one.
2. The dataset provided to us has both continuous and categorical data.
3. To detect the categorical variables in the dataset and give them numerical values. We iterate through all the columns and check if any attribute is string using function 'string_check', if yes then append the column indices to a list 's_indices'. Now we iterate through all such columns in 's_indices' and assign numerical values to the categorical attributes in that column. We are replacing "Absent" and "Present" with 0 and 1 respectively.
4. For dividing the dataset into training data and testing data we adopted 10-fold cross validation. We performed 10-fold cross validation by dividing data into 10 parts and

executing the algorithm 10 times such that each segment is treated as test and other segments as training set in successive iterations.

5. In the above formula, we can neglect the denominator P(X) which is constant for all the classes.
6. In the above formula, the calculation of class Prior Probability is calculated by the ratio of the counts of zeros and ones by the total number of rows in the training data respectively. This is for continuous data.
7. We have calculated separately the probabilities of categorical and continuous data.
8. To calculate P(X|H) that is descriptor Posterior probability in the above formula, we are using the Gaussian probability density function:

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

   where mean and standard deviation are calculated column wise. This is for continuous data.
9. To get the total class-wise probability, for each row of the testing data we have multiplied the above calculated prior probabilities of both categorical and continuous data with the Posterior probability of both categorical and continuous data.
10. We checked which class has higher probability and assigned that row to that class. Test_class is our predicted_class.
11. We now compare our predicted_class with the last column of our test_set i.e. actual_class and form the confusion matrix based on which, we calculate the accuracy, precision, recall and f1_measure which will be an average reading for all the 10 iterations of our 10-fold validation algorithm.

Choice Description:

Categorical Attributes:

For categorical features, we are replacing “Absent” and “Present” with 0 and 1 respectively.We first iterate through all the columns of the first row of our dataset and apply ‘string_check’ function on each value which checks if the value at a given column in row 0 is string or not. If it is string, then we append its index to ‘s_indices’ list. Now for each of the columns whose index are in s_indices, we check the number of unique values in that column and assign them with numerical values, one numerical value for each unique categorical value and we write these original values and replacement values in dictionary. So, if the column has categorical values ‘present’ and ‘absent’, then they will be replaced with ‘0’ and ‘1’ respectively. Then we are dividing the count of occurrence of ‘0’ or ‘1’ for the class divided by the total number of 0’s or 1’s present in the dataset.

Continuous Attributes:

For continuous features, we are calculating mean and standard deviation column wise. We are using this mean and standard deviation for calculating the Gaussian probability distribution on test data.

Zero-Probability:

If an attribute value is missing in the dataset, if we calculate the posterior probability on that row, it will give zero probability. We can use Laplacian correction to rectify it.

Results:

Project3_DataSet1			
Accuracy	**Precision**	**Recall**	**F1_measure**
94.73	91.66	95.65	93.61
92.98	88.23	88.23	88.23
96.49	87.5	100.0	93.33
91.22	90.0	85.714	87.80
89.47	88.88	80.0	84.21
96.49	100.0	92.59	96.15
96.49	95.45	95.45	95.45
96.49	93.33	93.33	93.33
91.228	96.42	87.09	91.52
89.28	86.36	86.36	86.36
Final Results			
Accuracy	**Precision**	**Recall**	**F1_measure**
93.490	91.787	90.444	91.003

Project3_DataSet2			
Accuracy	**Precision**	**Recall**	**F1_measure**
65.95	60.0	71.42	65.21
74.46	52.38	84.61	64.70
78.26	76.47	68.42	72.22
67.39	50.0	60.0	54.54
63.043	60.0	45.0	51.42
63.043	52.63	55.55	54.054
84.78	75.0	69.23	71.99
76.08	50.0	54.54	52.17
60.86	44.44	50.0	47.05
69.56	50.0	57.14	53.33
Final Results			
Accuracy	**Precision**	**Recall**	**F1_measure**
70.347	57.093	61.594	58.674

Result Analysis:

- As Naïve Bayes algorithm assumes all the attributes are unrelated. It performs well on datasets which have unrelated attributes.
- We observe that the performance of our model is good for a large dataset where it gets a higher training percent when compared to that of a smaller dataset. Hence we get the accuracy for dataset1 higher than that of dataset2.

Cross-validation:

We have implemented 10-fold cross validation in our implementation. This way we improve the efficiency of our algorithm as we train our dataset 10 times. We segment our dataset in 10 parts and keeping each segment as test and remaining dataset as train set for each iteration, we calculate the accuracy, precision, recall and f1_measure. We then take average of all the readings and those averaged values are our final accuracy, precision, recall and f1_measure.

Pros and Cons:

Pros:

- It is efficient when dealing with databases which are large.
- It is easy and fast to predict class of test data set. It also performs well in multi class prediction.
- As the labels are independent of each other, it is easy to implement.
- The algorithm is capable of handling both continuous and discrete data.

Cons:

- Naïve Bayes makes assumption of independent predictors. So it does not perform well if data is dependent on each other.
- If categorical variable has a category in test data set, which was not observed in training data set, then model will assign a 0 (zero) probability and will be unable to make a prediction. This is often known as “Zero Frequency”. To solve this, we can use the smoothing technique- Laplacian estimation.

Algorithm 4: Random Forest:

Introduction:

- Random Forest Algorithm is an improvement over top of decision tree algorithm. The main idea behind the random forest algorithm is to take multiple random subsets of data and generate multiple small decision trees, that finally operates as an ensemble. Hence this algorithm is given the name random forest. Each individual tree in random forest gives us a class prediction and the class with maximum votes becomes the model prediction.

- Each of the decision tree that is generated considers only a subset (what we mean by subset is more explained in the Bagging section below) of the data and hence gives a biased classifier. This way each of the decision tree is able to capture a different trend in the data.

- The reason why random forest gives wonderful classification predictions is that the trees protect each other from their individual errors. While some trees may be wrong with a class prediction,others will be right. So, a group of trees are able to move in the correct direction.

Bagging:

To provide the most accurate results we need to ensure that the behaviour of each individual tree is not corelated to behaviour of any other trees in the model. Decision trees are sensitive to the training data and a small change in the training dataset may result in significant difference in the tree structure. Random Forest take advantage of this by allowing each individual tree to randomly sample from the dataset with replacement, thus resulting in different trees. This process is called Bagging. With bagging we are not subsetting the training data into smaller chunks and training each tree on different chunk. But instead of original training data we take a random sample with replacement.

Feature Randomness:

In normal decision tree, we consider every possible feature and pick the one that result in most separation in observations in left node then the one in right node. But in random forest we pick only a random subset of features. This forces even more variation among the trees in the model and ultimately results in lower correlation among the trees.

Algorithm and Implementation:

1. The number of trees is given as input.
2. We are creating a decision tree with random samples which are chosen from the training data with replacement(Bagging). We are using the function np.random.choice to achieve this.
3. For choosing number of features for splitting we have given option to select the number of features. We are applying “random.sample” function on the training data and selecting these features for splitting.
4. We are finding the **best split** i.e. the split node to be treated as our root node from the training set(from this data we are selecting subset of features).

5. Then we are following the steps mentioned in the decision tree.
6. For finding best split we use ‘find_node’ function. Here we iterate through the rows and columns and split the data to ‘left_list’ and ‘right_list’. For every element of data, we find the left and right lists such that all the data less than that element lies in left_list and if greater than that element, then lies in right_list.
7. Then we calculate the gini-index corresponding to that element and its left and right lists. If the gini index for this element is less that the minimum gini index calculated yet, then the element corresponding to the least gini index becomes the best node for split. This is checked for every element in the data and the lowest gini index found will correspond to the best node for split.
8. This best node is considered as the root for the tree which consists of the following attributes- row number of the element, the element itself, column number and corresponding left and right lists.
9. Now we build the decision tree using ‘decision_tree’ function. Here, if left list is not empty, we check if all the elements in left list belong to same class and find that class using ‘find_class’ function where the class is determined by the maximum count that is available for any class. The same way, we check the class of right list, if all the elements belong to the same class.
10. Now if the elements does not belong to the same class, then we do a recursion on tree and pass the root as the best node available in the left and right trees respectively. We repeat the steps 9 and 10 until the stopping condition.
11. We **stop** when all the elements are either in left list or all on right list and then find the corresponding class.
12. In a similar fashion, we have constructed the ‘n’ number of trees.
13. We ran the test data through each of these trees to gather votes from them.
14. We used majority voting to decide which class the data belongs to.
15. We now compare our predicted_class with the last column of our test_set i.e. actual_class and form the confusion matrix based on which, we calculate the accuracy, precision, recall and f1_measure which will be an average reading for all the 10 iterations of our 10-fold validation algorithm.

Choice Description:

Number of Trees:

The more the number of trees, it reduces variability in classification.

Number of Features:

While computing the best split at each node, we randomly choose from these features.

Categorical Features:

To detect the categorical features and process them, we follow step2 of our decision tree implementation. The details are as follows. We first iterate through all the columns of the first row of our dataset and apply ‘string_check’ function on each value which checks if the value at a given column in row 0 is string or not. If it is string, then we append its index to ‘s_indices’ list. Now for each of the columns whose index are in s_indices, we check the number of unique values in that column and assign them with numerical values, one numerical value for each unique categorical value and we write these original values and replacement values in dictionary. So, if the column has categorical values ‘present’ and ‘absent’, then they will be replaced with ‘0’ and ‘1’ respectively. Now when we check for the best split node, we check for these columns separately as since these are not continuous values, we cannot check for less than or greater than values to compose our left and right lists. Hence, if the value exactly matches with the element, then we put them in left list or else in right list. Then we calculate the gini index for each element in the dataset and find the minimum gini which corresponds to the node value for the best split. The same procedure is applied for classification of labels for test data where we calculate the labels for categorical data separately as we cannot use less than and greater than comparison operators here as well.

Continuous Features:

To handle the continuous features, for each value in the dataset, we iterate through all the values of the dataset one by one and if the values are less than that particular value, then we append it to the left list and if greater, then we append it to the right list. Gini index for both the lists are calculated separately and then a combined gini index is calculated corresponding to that node. Likewise, gini index for each data element is calculated and the data for which gini is minimum, is considered as the node for the best split.

Results:

Project3_DataSet1			
Accuracy	**Precision**	**Recall**	**F1_measure**
94.73	88.46	100.0	93.87
92.82	84.21	94.11	88.88
94.73	92.30	85.71	88.88
96.49	95.23	95.23	95.23
91.22	85.71	90.0	87.80
92.98	92.59	92.59	92.59
96.49	95.45	95.45	95.4
94.73	92.85	86.66	89.65
94.73	96.66	93.54	95.08
92.85	87.5	95.45	91.30
Final Results			
Accuracy	**Precision**	**Recall**	**F1_measure**
94.89	93.524	92.879	91.879

Project3_DataSet2			
Accuracy	**Precision**	**Recall**	**F1_measure**
46.88	40.90	42.85	41.86
72.34	50.0	61.53	55.17
76.08	78.57	57.89	66.66
65.21	46.15	40.0	42.85
69.56	71.42	50.0	58.82
60.86	50.0	38.88	43.75
67.39	44.44	61.53	51.61
76.08	50.0	45.45	47.61
56.52	35.71	31.25	33.33
65.21	42.85	42.85	42.85
Final Results			
Accuracy	**Precision**	**Recall**	**F1_measure**
65.611	51.008	47.228	48.455

Project3_dataset1.txt

Number of trees:1
Accuracy: 92.086
Precision: 89.238
Recall: 89.665
F1_measure: 89.276

Number of trees:3
Accuracy: 94.361
Precision: 91.097
Recall: 94.173
F1_measure: 92.053

Number of trees:5
Accuracy: 94.364
Precision: 91.496
Recall: 93.559
F1_measure: 92.436

Project3_dataset2.txt:

Number of trees:1
Accuracy: 63.002
Precision: 47.363

Recall: 49.633
F1_measure: 47.265

Number of trees:3
Accuracy: 61.711
Precision: 44.961
Recall: 43.19
F1_measure: 42.800

Number of trees:10
Accuracy: 68.626
Precision: 57.816
Recall: 41.342
F1_measure: 46.986

Result Analysis:

The reason why random forest gives wonderful classification predictions is that the trees protect each other from their individual errors. While some trees may be wrong with a class prediction, others will be right. So, a group of trees are able to move in the correct direction. This helps reduce the variance of the model without increasing bias.
As random forests are constructed using a random set of features, it corrects the overfitting issues faced by decision trees.

Pros and Cons:

<u>Pros:</u>

- Random Forest is known to be one of the most accurate classification algorithms that produces a highly accurate classifier model.
- As Random forest do not consider all attributes while creating random trees, it can handle large data very well.
- Random forest do not require feature engineering (scaling and normalization).
- Random Forest decorrelates trees. It is important when we are dealing with dataset with multiple features which may be correlated.
- Random Forest handles the overfitting problem that arises in decision tree, since in Random Forest various randomly generated trees are considered.
- Random Forest can train a classification model with relatively small number of samples and get pretty good results.

<u>Cons:</u>

- Overfitting may occur in case data is noisy.
- Unlike decision trees, results are difficult to interpret.
- Since Random Forest takes into account several randomly generated trees to classify the test data, so it is slower as compared to other algorithms.

- Hyperparameters require good tuning for generating accurate results.

Cross-validation:

We have implemented 10-fold cross validation in our implementation. This way we improve the efficiency of our algorithm as we train our dataset 10 times. We segment our dataset in 10 parts and keeping each segment as test and remaining dataset as train set for each iteration, we calculate the accuracy, precision, recall and f1_measure. We then take average of all the readings and those averaged values are our final accuracy, precision, recall and f1_measure.

Kaggle Competition Report

Objective:

We are given with three datafiles on Kaggle. The first datafile (train_features.csv) contains the training data features, the second data file (train_label.csv) contains the training class labels for the records given in the first file and the third file (test_features.csv) contains the test data features. The objective is to design a classification model that use any of the classification algorithms like Decision Tree, Random Forest, KNN, Logistic Regression, SVM, Adaboost, implement some improvement on top of these algorithms to classify the records given in test_features.csv. The results are to be submitted on Kaggle in a Excel Sheet.

Improvements:

1) Parameter Tuning:
   We know that all classification algorithms are driven by parameters. These parameters influence outcome of learning process. The main objective of parameter tuning is to find the optimum value of each parameter to improve the accuracy. We have tuned the parameters for all the classification algorithms and tried to use the tuned algorithms for the classification of the given test dataset. You can see our work for parameter tuning in the parameter tuning section.
2) Ensemble Learning:
   We have used the Ensemble learning method AdaBoost in our code.
3) Preprocessing and cleaning and feature Scaling:
   Before making any actual predictions it is always good to scale the features so that all of them are uniformly evaluated. This helps in better training and testing of the model.
4) Majority Voting for Ensembling:
   We have used individual results from all the stable performing classifier algorithms and done majority voting for the predicted results.

Other Efforts Towards Improvements:

The improvements written in the above points are used for the best result that we got on Kaggle. There were some other improvements that we tried but finally decided not to use it because of less accurate results. These were:

1) Bagging: We tried implementing bagging for all the base classifiers, do soft and hard

majority voting and submit the results on Kaggle. But we found F beta measure to be lower. SO we decided not to go forward with this idea.

2) Feature Extraction: This is based on extracting features from the dataset, that may have a higher ability to explain variance in the training data. This may give improved model accuracy. But we were getting a lower F beta measure on Kaggle, so we dropped this idea.
3) PCA: We also did dimensionality reduction of train and test datasets, then use all selected base algorithms to predict class labels and finally do majority voting. But we got less accurate results than the current method we have used.

Parameter Tuning and Choosing Appropriate Base Algorithms:

In order to get the best performance metrics, we need to first find what are best base algorithms for the given dataset. For this we have followed the steps given below:

- We have individually tuned the parameters of each classification algorithm.
- As we are not provided with any test dataset labels. So, to get an estimate of performance measures for individual algorithms we took the training dataset (train_features.csv and test_features.csv) and divided it into training features, testing features, training labels and testing labels. 70% of the data goes towards training dataset and the rest 30% goes towards test dataset.
- Then we have used the library functions of all the algorithms to train and test the model for various parameters and calculated the performance metrics.
- This has helped us to get the best input parameters for each of the classification algorithms.
- The parameter tuning Analysis and Results for all the algorithms are shown below.

Parameter Tuning, Analysis and Results:

The tables below show the parameter tuning for various algorithms that are under our consideration for choosing the base algorithms. We have chosen the base algorithms based on parameter tuning.

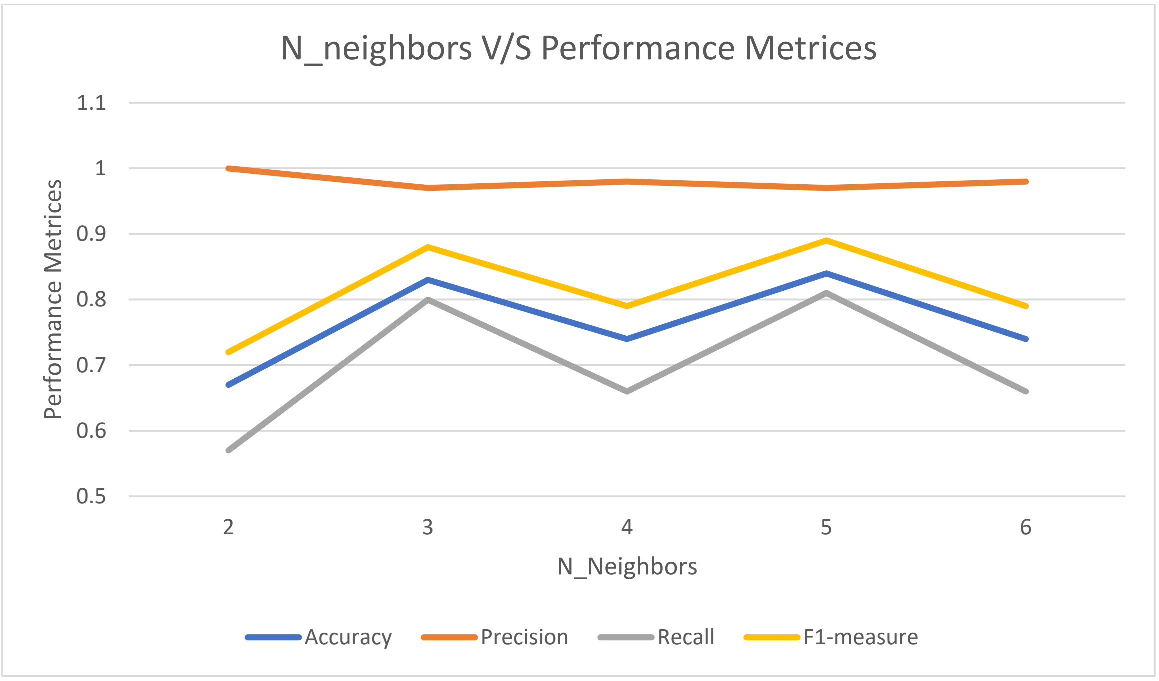
KNN:

<u>Tuning n_neighbors(K):</u>

N_neighbors represent the number of neighbors to use for k-neighbors queries. We can see that we get the best performance for K=5, but when we run k = 6 along with weights = ‘distance’ we get the best performance. This can be seen in the next table.

N_neighbors	Accuracy	Precision	Recall	F1-measure
2	0.67	1	0.57	0.72
3	0.83	0.97	0.80	0.88
4	0.74	0.98	0.66	0.79
5	0.84	0.97	0.81	0.89
6	0.74	0.98	0.66	0.79

The performance graph for the table above is shown below:



Tuning Weights:

- Weights is a parameter used in prediction. We have two available options:
  - ‘uniform’: Uniform weights. All points in each neighborhood are weighted equally.
  - ‘distance’: weight points by inverse of their distance. In this case closer neighbors of a query point will have greater influence then neighbors which are farther away.
- Other parameters set at: n_neighbors=6.

Weights	**Accuracy**	**Precision**	**Recall**	**F1-measure**
distance	0.86	0.99	0.82	0.90
uniform	0.74	0.98	0.66	0.79

Analysis and Final Parameters for KNN:

- Based on the results shown above we have selected n_neighbors = 6 and weights = ‘distance’ as the best parameters for the given dataset.
- Although n_neighbors = 5 give the most accurate and stable performance metrices for KNN. But we choose n_neighbors = 6, because when combined with weights = ‘distance’, it give more stable and accurate results.

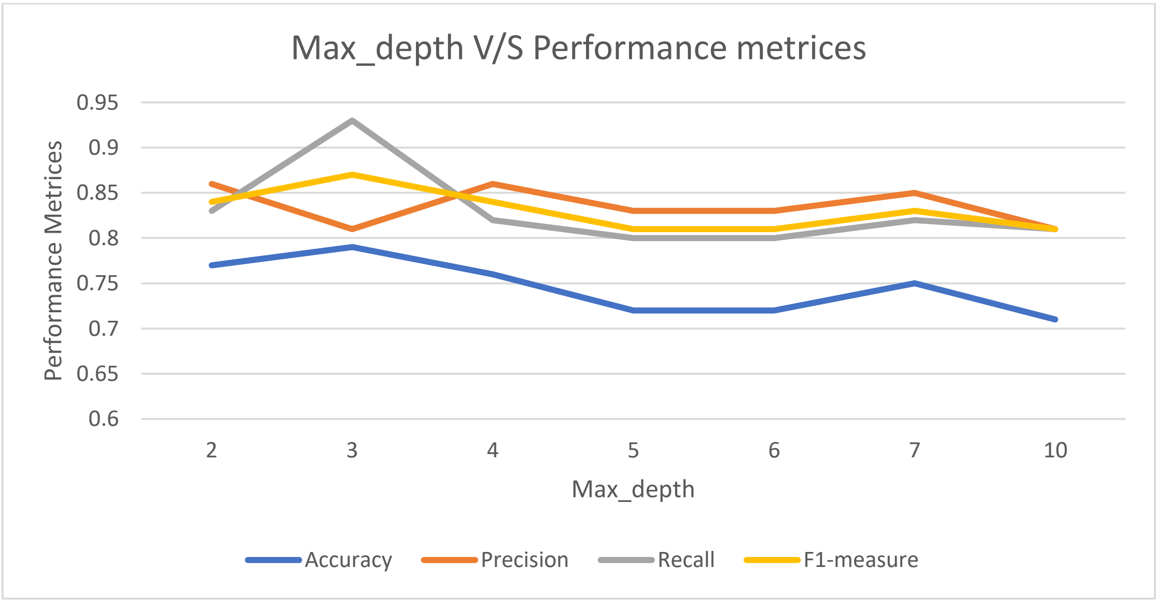
DECISION TREE:

Tuning max_depth:

It is defined as the maximum depth of the decision tree formed.

Max_depth	**Accuracy**	**Precision**	**Recall**	**F1-measure**
2	0.77	0.86	0.83	0.84
3	0.79	0.81	0.93	0.87
4	0.76	0.86	0.82	0.84
5	0.72	0.83	0.80	0.81
6	0.72	0.83	0.80	0.81
7	0.75	0.85	0.82	0.83
10	0.71	0.81	0.81	0.81

The graph plot for the above shown table is:



Tuning criterion:

- This parameter function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain.

- Other parameters set at: max_depth = 3.

criterion	Accuracy	Precision	Recall	F1-measure
gini	0.78	0.81	0.92	0.86
entropy	0.80	0.84	0.91	0.87

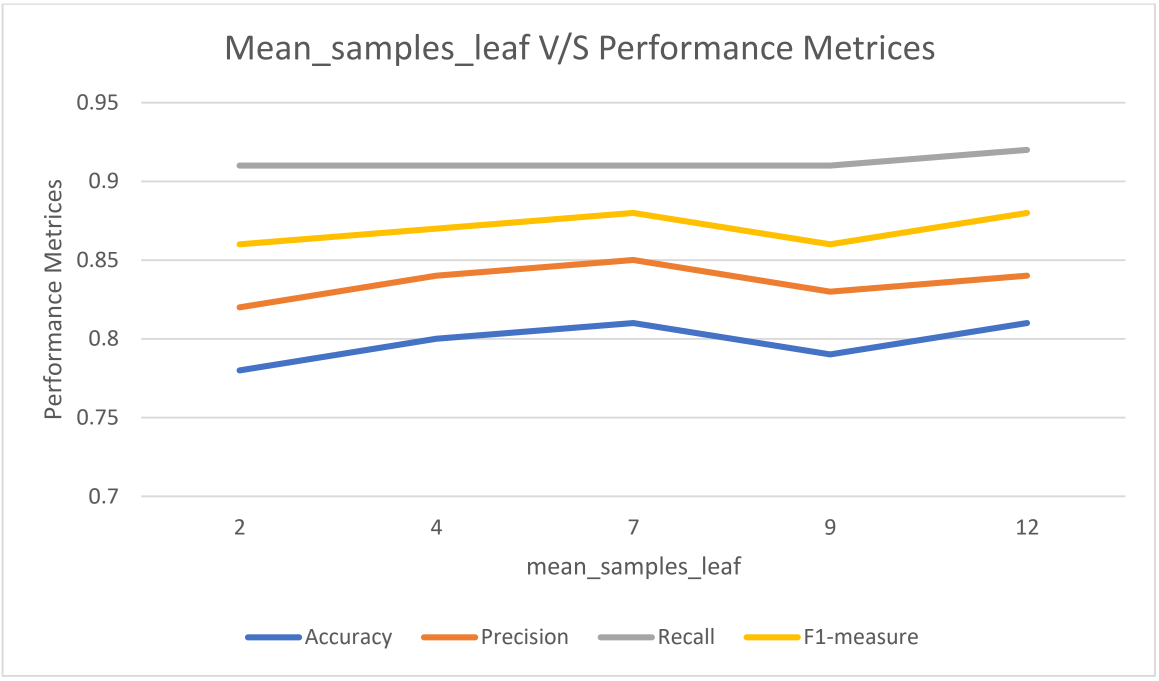
Tuning mean_samples_leaf:

This parameter specifies the minimum number of samples required to be at a leaf node.

Other parameters set at: max_depth = 3, criterion = 'entropy'

mean_samples_leaf	Accuracy	Precision	Recall	F1-measure
2	0.78	0.82	0.91	0.86
4	0.80	0.84	0.91	0.87
7	0.81	0.85	0.91	0.88
9	0.79	0.83	0.91	0.86
12	0.81	0.84	0.92	0.88

The graph plot for the above shown table is:



Final Parameters for Decision Tree:

Based on the tables and graphs shown above, we found that the best parameters for decision tree algorithm for the given dataset are max_depth=3, criterion='entropy', min_samples_leaf=12.

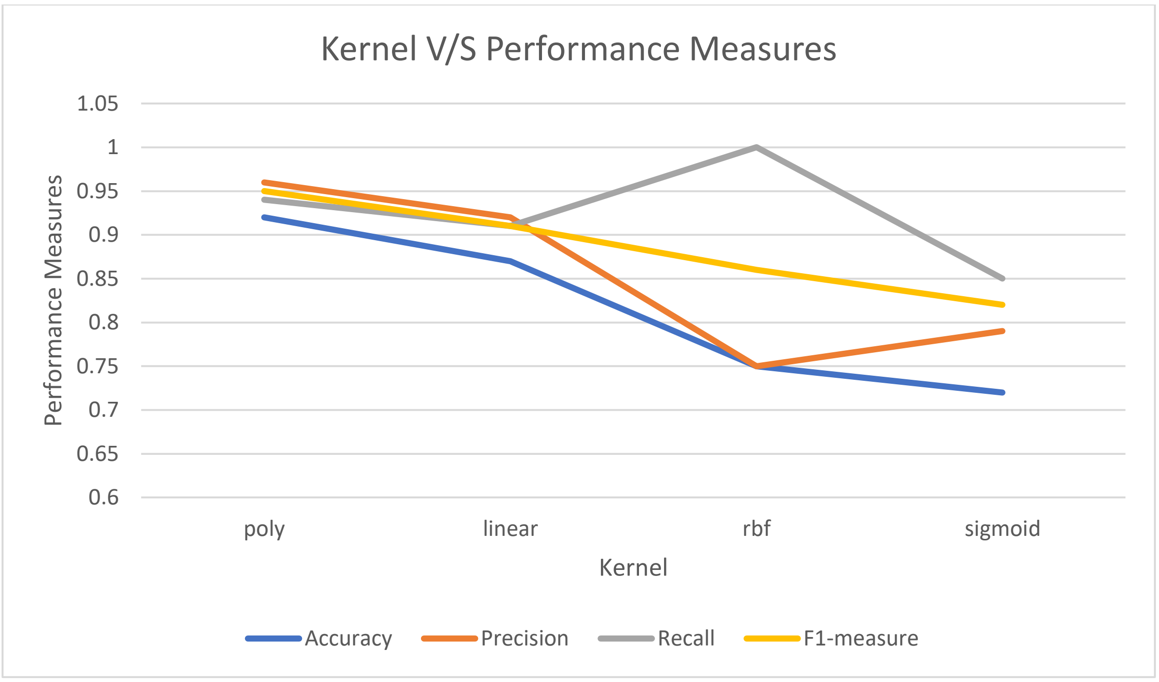
SVM:

Tuning Kernel:

This parameter specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable.

kernel	Accuracy	Precision	Recall	F1-measure
poly	0.92	0.96	0.94	0.95
linear	0.87	0.92	0.91	0.91
rbf	0.75	0.75	1.00	0.86
sigmoid	0.72	0.79	0.85	0.82

The graph plot for the above shown table is:



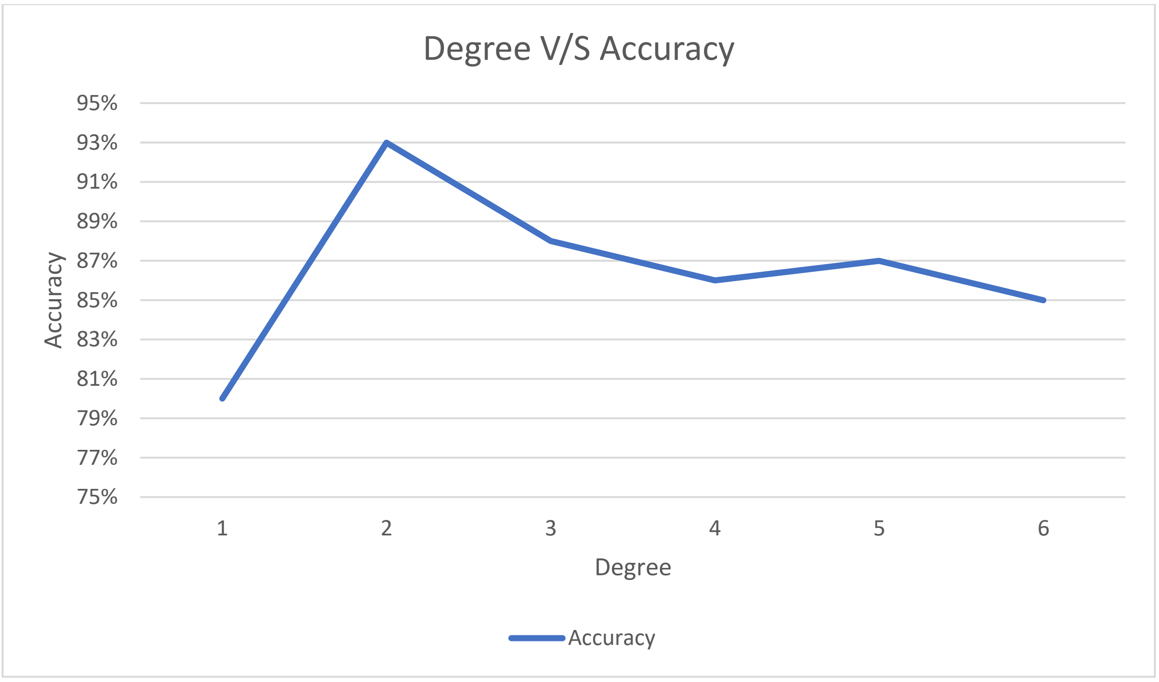
Tuning Degree:

- Degree of the polynomial kernel function ('poly').
- Other parameter set at: kernel = 'poly'

Degree	Accuracy
1	80%
2	93%
3	88%
4	86%
5	87%

6	85%

The graph plot for the above shown table is:



Final Parameters for SVM:

Based on the tables and graphs shown above, we found that the best parameters for SVM algorithm for the given dataset are kernel = ‘poly’, degree = 2.

LOGISTIC REGRESSION:

Tuning penalty:

- Penalty is used to specify the norm used in the penalization. The ‘newton-cg’, ‘sag’ and ‘lbfgs’ solvers support only l2 penalties.
- Other parameters kept at: random_state=10, multi_class='ovr'.

Penalty	Accuracy	Precision	Recall	F1-measure
l1	0.94	0.96	0.97	0.96
l2	0.89	0.97	0.97	0.93

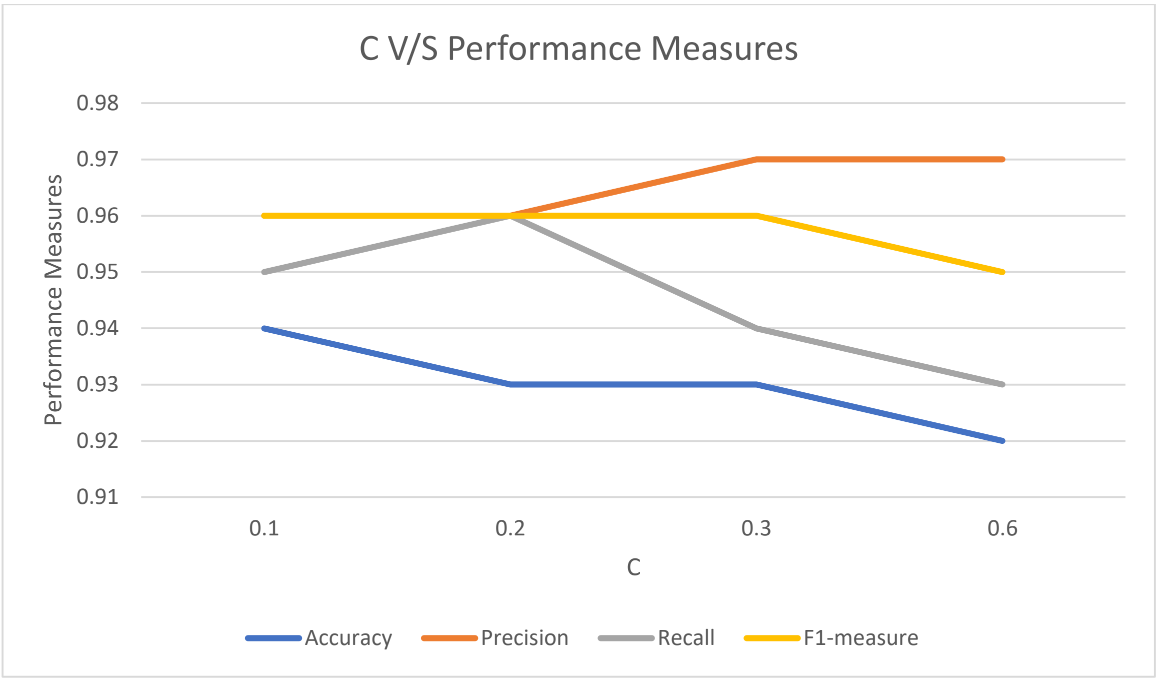
Tuning C:

- C is the Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.
- Other parameters kept at: random_state=10, multi_class='ovr', penalty='l1'.

C	Accuracy	Precision	Recall	F1-measure
0.1	0.94	0.96	0.95	0.96
0.2	0.93	0.96	0.96	0.96
0.3	0.93	0.97	0.94	0.96

0.6	0.92	0.97	0.93	0.95

The graph plot for the above shown table is:



Final Parameters for Logistic Regression:

Based on the tables and graphs shown above, we found that the best parameters for Logistic Regression algorithm for the given dataset are random_state=10, multi_class='ovr', C=0.1, penalty='l1'.

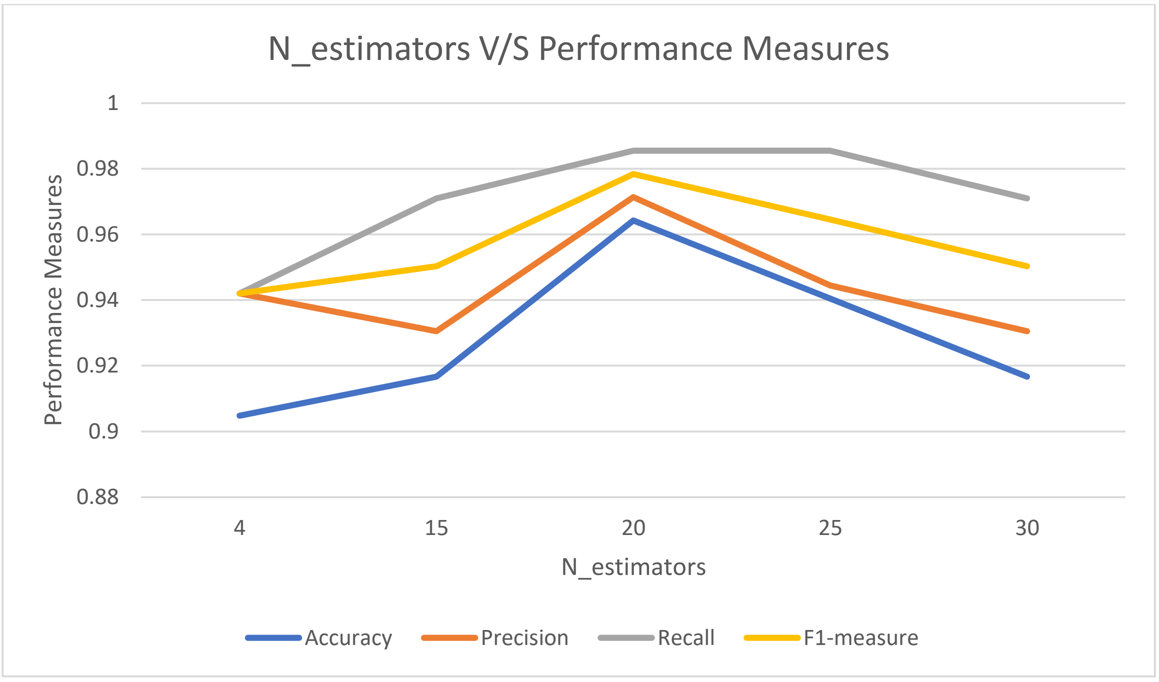
RANDOM FOREST:

Tuning n_estimators:

N_estimators is the number of trees in the forest.

N_estimators	**Accuracy**	**Precision**	**Recall**	**F1-measure**
4	0.9047619047619048	0.9420289855072463	0.9420289855072463	0.9420289855072463
15	0.9166666666666666	0.9305555555555556	0.9710144927536232	0.950354609929078
20	0.9642857142857143	0.9714285714285714	0.9855072463768116	0.9784172661870504
25	0.9404761904761905	0.9444444444444444	0.9855072463768116	0.9645390070921985
30	0.9166666666666666	0.9305555555555556	0.9710144927536232	0.950354609929078

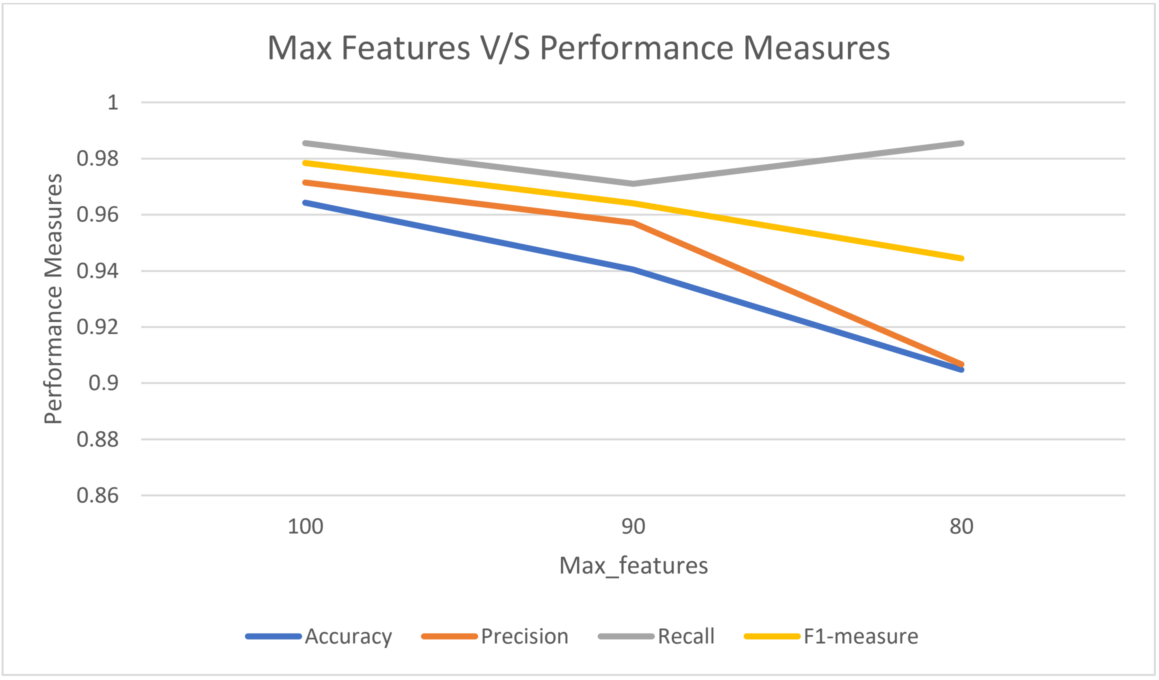
The graph plot for the above shown table is:



Tuning max_features:

- Max features specifies the number of features to consider when looking for the best split.
- Other parameters kept at: n_estimators=20.

Max_features	**Accuracy**	**Precision**	**Recall**	**F1-measure**
100	0.9642857142857143	0.9714285714285714	0.9855072463768116	0.9784172661870504
90	0.9404761904761905	0.9571428571428572	0.9710144927536232	0.9640287769784173
80	0.9047619047619048	0.9066666666666666	0.9855072463768116	0.9444444444444444

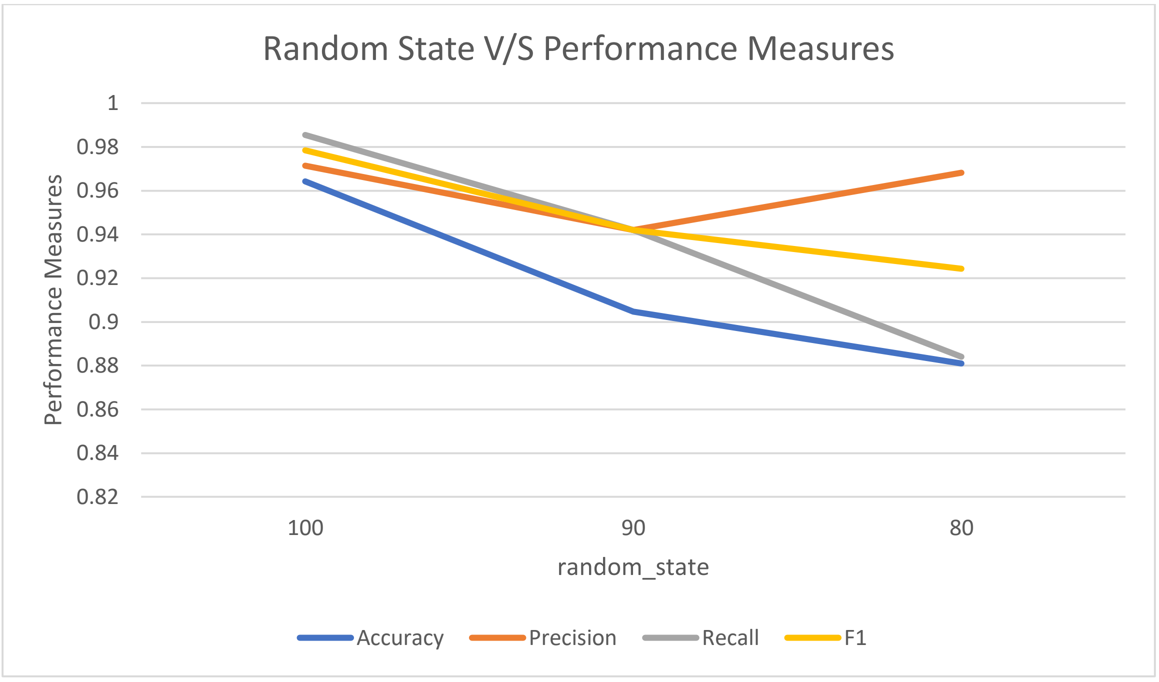


Tuning random state:

- Random_state is the seed used by the random number generator.
- Other parameters kept at: n_estimators=20, max_features=100

Random_state	Accuracy	Precision	Recall	F1
100	0.9642857142857143	0.9714285714285714	0.9855072463768116	0.9784172661870504
90	0.9047619047619048	0.9420289855072463	0.9420289855072463	0.9420289855072463
80	0.8809523809523809	0.9682539682539683	0.8840579710144928	0.9242424242424242

The graph plot for the above shown table is:



Final Parameters for Random Forest:
Based on the tables and graphs shown above, we found that the best parameters for Random Forest algorithm for the given dataset are n_estimators=20, random_state=8, max_features=100.

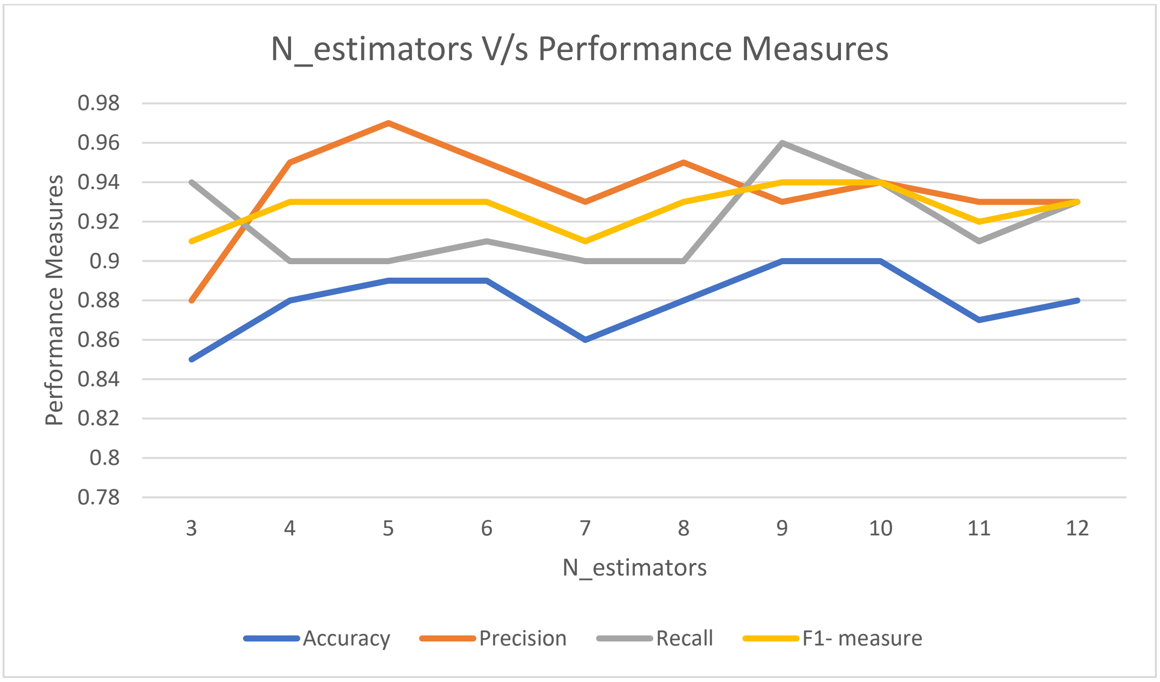
ADABOOST:
Tuning n_estimators:

- N_estimators is the number of weak learners to train iteratively.
- Other parameters kept at: learning_rate = 1.

n_estimators	Accuracy	Precision	Recall	F1- measure
3	0.85	0.88	0.94	0.91
4	0.88	0.95	0.90	0.93
5	0.89	0.97	0.90	0.93
6	0.89	0.95	0.91	0.93

7	0.86	0.93	0.90	0.91
8	0.88	0.95	0.90	0.93
9	0.90	0.93	0.96	0.94
10	0.90	0.94	0.94	0.94
11	0.87	0.93	0.91	0.92
12	0.88	0.93	0.93	0.93

The graph plot for the above shown table is:

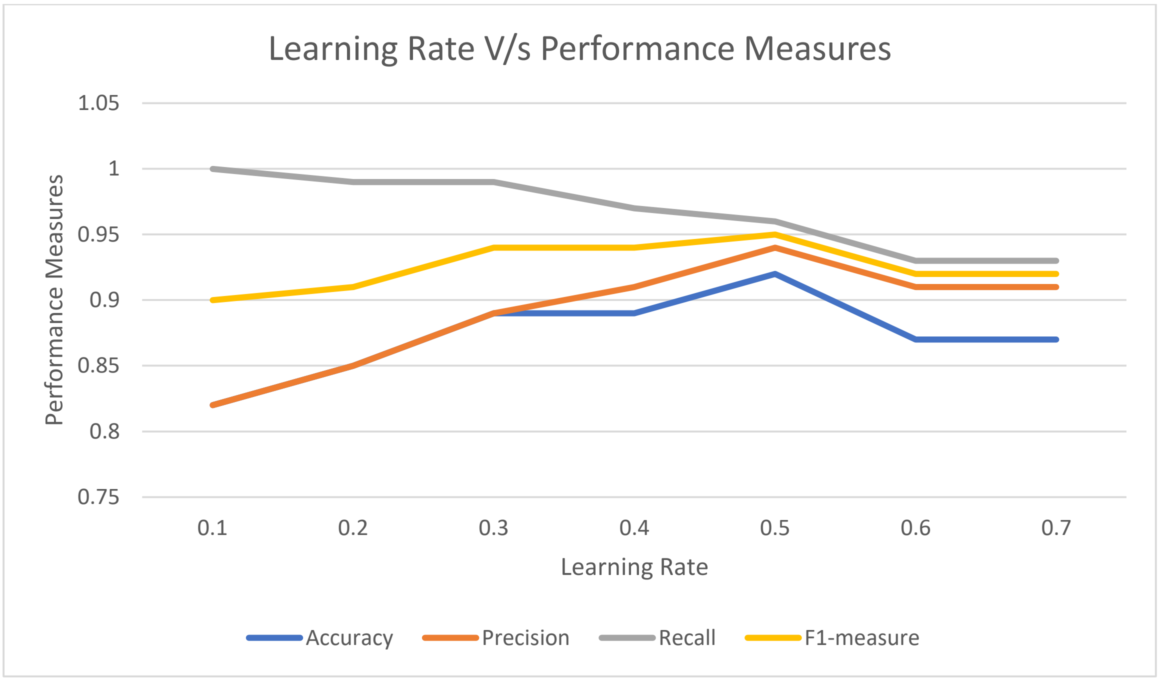


<u>Tuning learning rate:</u>

- This parameter shrinks the contribution of each classifier by learning rate.
- Other parameters kept at: n_estimators = 9 (best value for accuracy).

Learning_rate	**Accuracy**	**Precision**	**Recall**	**F1-measure**
0.1	0.82	0.82	1.00	0.90
0.2	0.85	0.85	0.99	0.91
0.3	0.89	0.89	0.99	0.94
0.4	0.89	0.91	0.97	0.94
0.5	0.92	0.94	0.96	0.95
0.6	0.87	0.91	0.93	0.92
0.7	0.87	0.91	0.93	0.92

The graph plot for the above shown table is:



<u>**Final Parameters for AdaBoost Algorithm:**</u>

Based on the tables and graphs shown above, we found that the best parameters for Adaboost algorithm for the given dataset are n_estimators=9, learning_rate=0.5.

RESULTS:

- We submitted results for each separate tuned classification algorithm on Kaggle to check which one performs better, we found the F-beta measure as shown below:

Algorithm	F-beta measure (Kaggle)
KNN	0.82307
Decision Tree	0.80297
Random Forest	0.85920
Ada Boost	0.82312
SVM	0.81021
Logistic Regression	0.85324

- From the results shown above for different classification algorithms for the best tuned parameters, we see that Decision Tree is the least accurate(as seen in the parameter tuning section), also it is least accurate on Kaggle, so we do not chose it as a base algorithm.
- So we choose SVM, Random Forest, AdaBoost, Logistic Regression and KNN as our base algorithms. We have tried and implemented various improvements on top of these algorithms, as written in the improvement section.

References:

1. https://en.wikipedia.org/wiki/Decision_tree_learning
2. https://www.datacamp.com/community/tutorials/decision-tree-classification-python
3. https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761
4. https://en.wikipedia.org/wiki/Normal_distribution
5. https://becominghuman.ai/naive-bayes-theorem-d8854a41ea08
6. https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/
7. https://dimensionless.in/introduction-to-random-forest/
8. https://towardsdatascience.com/understanding-random-forest-58381e0602d2
9. https://www.geeksforgeeks.org/naive-bayes-classifiers/
10. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
11. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html