Project 3: Classification Algorithms

CSE 601: Data Mining and Bioinformatics

Team members:

Arvind Thirumurugan	athirumu	50289656
Saketh Varma Pericherla	sakethva	50288206
Vijay Jagannathan	vijayjag	50290947

Part 1: Implementing Classification Algorithms:

1.0 Results:

Dataset	**Measures**	**Algorithms**			
		Accuracy	**Precision**	**Recall**	**F1**
1	**K-nn**	0.914285	0.8342827	0.976476	0.893758
	Naïve Bayes	0.935142	0.9241315	0.905343	0.9132090
	Decision Tree	0.917418	0.884163	0.901467	0.889868
	Random Forest	0.954323	0.945627	0.933005	0.937484
2	**K-nn**	0.678260	0.533307	0.493544	0.503983
	Naïve Bayes	0.702173	0.570786	0.617493	0.584200
	Decision Tree	0.629814	0.424779	0.467053	0.441297
	Random Forest	0.664384	0.522194	0.447733	0.450232
3_test	**Decision Tree**	1.0	1.0	1.0	1.0
	Random Forest	1.0	1.0	1.0	1.0

The above table shows the results obtained by our implementation of K-nearest neighbours, Decision Tree, Naïve Bayes and Random forest algorithms for the four datasets provided.

- Datasets 1 and 2 are trained using 10-fold cross validation while dataset 3 is trained with a separate training set and tested on a separate testing set provided.
- The various hyper-parameters chosen for these different algorithms to get these final results and further analysis on the results are discussed in the respective sections below.

1.1 K-Nearest Neighbors:

K-nearest Neighbors is a lazy algorithm that uses the entries in the training data that are closest to classify the data sample under consideration. The distance metric used to find the closest entries is **Euclidean distance.** Each tuple in the training data is point in n-dimensional space, where n is the number of different columns. To classify a new point, the algorithm calculates k closest points and the new point is assigned a class label based on the majority class labels for the k closest points.

1.1.0 Pre-processing:

Since the data needs to be continuous. **All the Categorical data are assigned numerical values** and then **normalized all the converted continuous data** to prevent column values with larger ranges from outweighing those with lower ranges. There are several forms of normalization we chose min max normalization which is given as,

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

1.1.1 Parameter Setting:

The value of k decides the level of granularity for the boundary between predicted classes. A higher value of k makes the algorithm robust against outliers. A lower value of k will give sharper class boundaries but will also suffer from local minima.

1.1.2 Algorithm Implementation:

- Initially we split the data into training and testing data and normalize the training data and use the minimum and maximum column values of training data to normalize the test data.
- Then we calculate the **Euclidean distance** between each test sample and training data and get the k closest training samples to the test data.

$$d(\mathbf{q}, \mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$

- And then find majority class label from the k closest points and assign the class label to the test sample. Repeat this process all the test data

1.1.3 Result Analysis:

We got very good accuracy of 91.4% for dataset 1 because every column contains continuous values and Euclidean distance gave us good results. But for dataset 2 we only got 67.8% accuracy because it contains categorical columns as well, hence the algorithm performed poorly.

1.1.4 Pros:

- It is simple to implement and doesn't train any model
- The distance metric can be changed based on the dataset to get better results

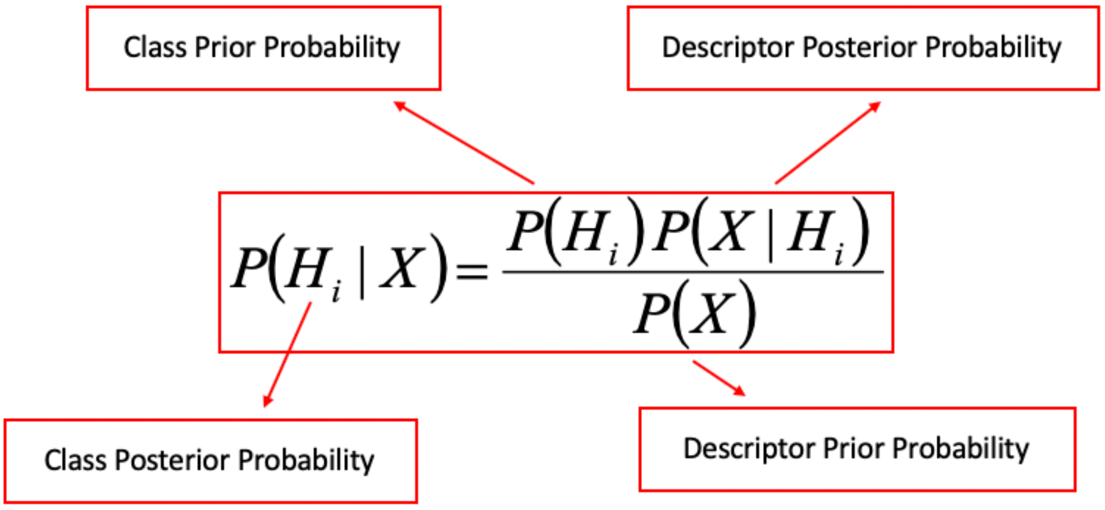
1.1.5 Cons:

- It gives equal weights to all the attributes and hence the accuracy can be significantly lowered due to noisy or irrelevant column values
- It can be computationally expensive for large datasets
- if k is too small it will be sensitive to outliers and if k is too large it may select points from other classes

1.2 Naïve Bayes Classifier:

Naïve Bayesian classifier is a simple classifier which assumes attribute independence. It is efficient when applied to large datasets and it offers comparable performance to decision trees.

1.2.0 Bayes Theorem:


Class Prior Probability
Descriptor Posterior Probability
Class Posterior Probability
Descriptor Prior Probability

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

To classify we need to calculate the highest Class Posterior probability for all classes $C_1, C_2, \ldots C_m$. In order to calculate the Class posterior probabilities Bayes theorem can be utilized.

Class Prior Probability:

$P(H_i)$ is the probability that the given tuple X belongs to a particular class C_i

Descriptor Prior Probability:

P(X) is the probability that we observe the attribute values of X in the dataset

Descriptor Posterior Probability:

$P(X|H_i)$ is the probability that we observe X in class C_i

Class Posterior Probability:

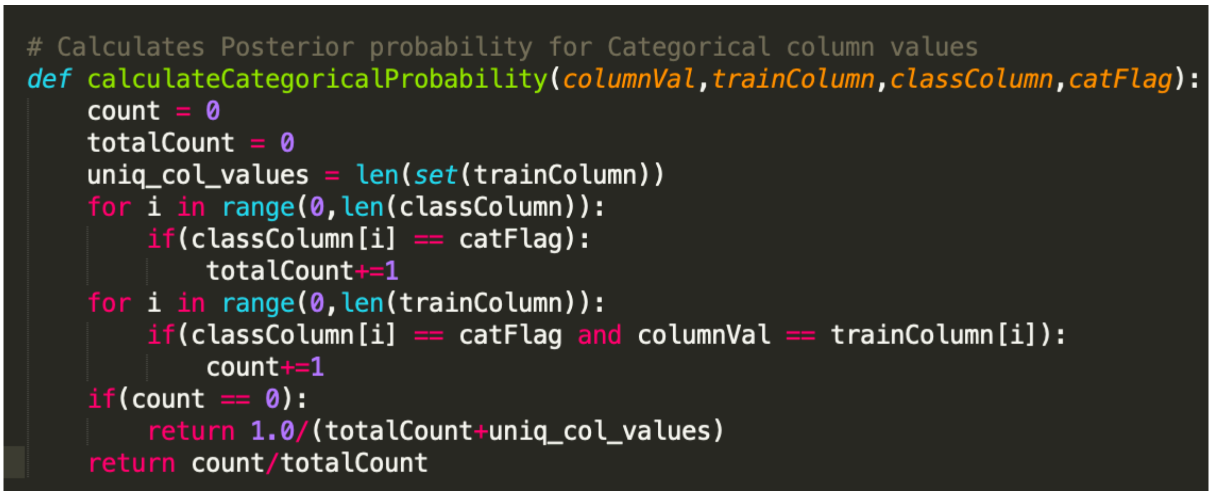
P(H_i) is the probability that a data sample X belongs to class C_i given the attribute values of X

1.2.1 Algorithm implementation:

- In the first step we split the data into training and test data and calculate the Descriptor prior probability
- In the next step we calculate the Posterior mean and standard deviation required for numerical columns in the data
- Then we calculate the class posterior probability for numerical and categorical column values separately and multiply all the values to get the required result.
- For **Continuous values** we assume the values are distributed as a Gaussian distribution and hence in order to calculate the Descriptor posterior probability we use the Gaussian function.

$$P(X_i = x_i | Y = y_j) = \frac{1}{\sqrt{2\pi}\sigma_{ij}} \exp^{-\frac{(x_i - \mu_{ij})^2}{2\sigma_{ij}^2}}$$

- For **Categorical values** we count the number of occurrences of the value in the training data for the given class and divide it by the total number of occurrences of the class in the training data.



```
# Calculates Posterior probability for Categorical column values
def calculateCategoricalProbability(columnVal,trainColumn,classColumn,catFlag):
    count = 0
    totalCount = 0
    uniq_col_values = len(set(trainColumn))
    for i in range(0,len(classColumn)):
        if(classColumn[i] == catFlag):
            totalCount+=1
    for i in range(0,len(trainColumn)):
        if(classColumn[i] == catFlag and columnVal == trainColumn[i]):
            count+=1
    if(count == 0):
        return 1.0/(totalCount+uniq_col_values)
    return count/totalCount
```

- **Categorical values** also face the **zero probability issue** which is handled by the if condition which checks if the count is zero and if it is zero adds the number of unique column values to the total count for the class and returns the ratio of 1/(totalCount+unique_column_values) to handle this scenario

1.2.2 Result Analysis:

We got a high accuracy of 93.5% for dataset 1 and got 70% accuracy for dataset 2. From the results we can infer that for dataset 1 the column values must be independent and for dataset 2 the column values might be dependent and hence the lower accuracy.

Pros:

- It is simple to understand and implement
- Makes probabilistic predictions and is efficient when dealing with large datasets
- If the conditional independence assumption holds it classifier converges faster and requires less data to make accurate predictions
- Can handle numerical and categorical data

Cons:

- If the attributes are correlated the classifier performs poorly.
- Encounters zero probability issue but it can be avoided by using Laplacian correction

1.3 Decision Tree:

A decision tree makes sequential, hierarchical decision about the outcomes variable based on the predictor data. Also known as CART (Classification and Regression Tree), a decision tree model is represented as a binary tree.

1.3.0 Algorithm Description:

Building a tree:

- Before we build a tree we need to know how to create a node and make a binary split.
- There are many possible trees that can be derived for a given dataset. To choose a tree that provides the best split we follow the greedy strategy i.e. Split the records based on an attribute test that optimizes certain criterion.
- To implement the greedy strategy, we will calculate GINI index at each node which indicates how pure the nodes are, where node purity refers to how mixed the training data assigned to each node is. The GINI index is calculated as:

$$GINI(t) = 1 - \sum_j [p(j \mid t)]^2$$

- Further, Information GAIN measures reduction in impurity achieved because of the split. This measure helps us choose the split that achieves most reduction

$$GAIN_{split} = Measure(p) - \left(\sum_{i=1}^{k} \frac{n_i}{n} Measure(i) \right)$$

- If GAIN is zero, make the node a leaf node – this node contains an output variable which is used to make a final prediction.
- If the GAIN is non-zero, then make two branches left and right that form the decision nodes – these nodes can further be split into left and right nodes based on their respective optimal GAIN calculations.

```
def _make_tree(self, subset, level=0):

   if self.max_depth and level == self.max_depth:
       return

   best_gain, best_question, best_gini = self._best_split(subset)

   if best_gain == 0:
       return self._make_leaf_node(best_gini, best_gain, subset)

       true_set, false_set = self._partition(subset, best_question)

       true_branch = self._make_tree(true_set, level+1)

       false_branch = self._make_tree(false_set, level+1)

       return self._make_decision_node(best_gini, best_gain, best_qu
estion, true_branch, false_branch)
```

Making a prediction:

- To make a prediction with the built decision tree we navigate the tree with the given row of data.
- We implement a recursive function, where the same prediction function is called again with left or right child nodes, depending on the condition at which the tree splits the data.
- If the type of node is “decision node” we continue stepping through the tree based on the satisfied condition until we reach a leaf node.
- If the type of node is “leaf node” we return the prediction provided by the leaf node. This is the terminating condition of the prediction.

```
def _classify(self, row, node):
        if not node:
            return np.random.choice(np.unique(self.data[:, -1]))
        elif node["type"] == "leaf":
            return node["prediction"]
        else:
            if node["question"].match(row):
                return self._classify(row, node["true_branch"])
            else:
                return self._classify(row, node["false_branch"])
```

1.3.1 Parameters:

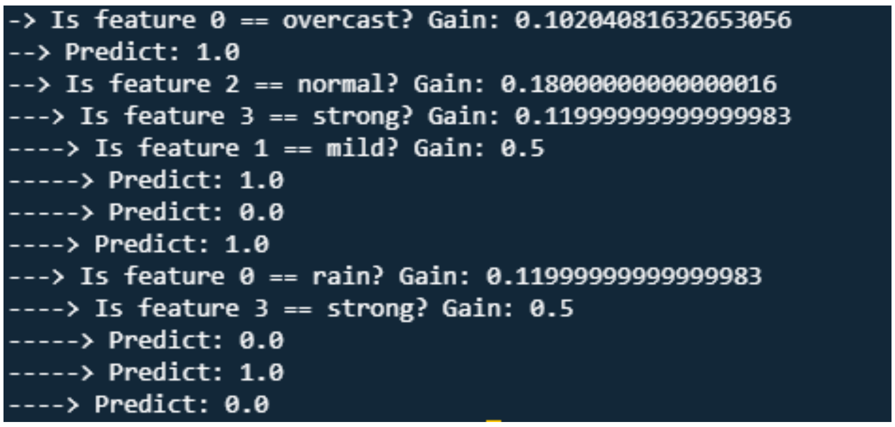
Continuous & Categorical features:

```
def match(self, row):

        val = row[self.column]

        if isinstance(val, float):
            return val >= self.value

        elif isinstance(val, str) or self.column == len(row)-1:
            return val == self.value
```

As shown in the above snippet we detect if the feature is a continuous variable or a categorical variable using python 3’s “**isinstance(value, datatype)”** function. If the column has continuous data, our condition to split the tree is “>=” whereas if the column has categorical data our condition to split the tree is “==”

Best features:

We use GINI and Gain calculations to find out the best split at a given node. The below tree shows the tree that we obtained for dataset 4:



```
-> Is feature 0 == overcast? Gain: 0.10204081632653056
--> Predict: 1.0
--> Is feature 2 == normal? Gain: 0.18000000000000016
---> Is feature 3 == strong? Gain: 0.11999999999999983
----> Is feature 1 == mild? Gain: 0.5
-----> Predict: 1.0
-----> Predict: 0.0
----> Predict: 1.0
---> Is feature 0 == rain? Gain: 0.11999999999999983
----> Is feature 3 == strong? Gain: 0.5
-----> Predict: 0.0
-----> Predict: 1.0
----> Predict: 0.0
```

Stop criteria:

There are two scenarios in which the algorithm stops building the tree. They are:

- If the information gain at a node is found to be zero, we cannot split it further, so we make it a leaf node and once there are no decision nodes to be filled the tree is said to be completely built.

- We provide maximum depth as an optional parameter. If specified, once the tree reaches the maximum depth specified we terminate the process of building the tree. We also provide an optional parameter called "n_features" which restricts the tree to utilize only "n" number of features specified in order to build the tree. These parameters are used judiciously to avoid overfitting the data.

1.3.2 Analysis:

- Decision trees can provide great accuracy even on sufficiently complicated datasets which is evident with the results of dataset 1.

- But from the dataset 2 it is evident that decision tree can be prone to overfitting small datasets as it tries to memorize and perfectly fit all the data points using rectangular decision boundaries.

Pros:

- Decision tree takes a non-parametric approach to do classification i.e. it doesn't need to know the probability distributions of various classes and other properties of the dataset.
- They are extremely fast at doing predictions at O(logn) time complexity where n is the maximum depth of the tree.
- Even though the idea of decision trees is trivial, they provide great accuracy that can be comparable with any modern algorithm.
- Decision trees are intuitive and easy to interpret and which makes it relatively easier to analyse and optimize for the given dataset.

Cons:

- Overfitting is a huge concern for decision trees since a decision tree is a rule based classifier. If parameters like maximum depth, number of features etc. are not provided, the tree tries to fit in all the data perfectly and ends up performing poorly on test dataset.
- Decision trees draw rectangular decision boundaries which is not optimal for datasets having more complex decision boundaries.
- Without pruning, decision tree can generate leaf nodes that do not have sufficient number of records to make the right prediction with confidence.
- Decision trees can produce trees that can be difficult to interpret unless we provide parameters like maximum depth, reduced number of features, minimum count at leaf node etc.

1.4 Random Forest:

Random forest is an ensemble learning algorithm that constructs a multitude of decision trees at training time and outputs the mode of the classes predicted by these trees for classification.

1.4.0 Algorithm Description:

Training:

- We choose "T" number of trees to build.
- For each tree "t" in the forest:
    - We generate "b" number of bags of data by choosing "s" number of data points from the training set with replacement.
    - We choose "m" number of features to select from randomly. Generally, $m < M$ (where M is total number of features) to avoid complexity and overfitting the model.

As shown in the below snippet we configure the parameters like number of trees, number of bags, number of features etc. and we reuse the decision tree implementation to build multiple trees.

```
def fit(self, X, y):

     for _ in range(self.n_estimators):

         data = np.concatenate((X, y), axis=1)

         if self.n_bootstrap:
             size = self.n_bootstrap
         else:
             size = data.shape[0]

         data_bootstrapped = self._get_bootstrap_set(data, size)

         X_bootstrapped, y_bootstrapped = data_bootstrapped[:,:,\
-1], data_bootstrapped[:, -1].reshape((-1, 1))

         tree = DecisionTreeClassifier(
             n_features=self.n_features, max_depth=self.max_depth)

         tree.fit(X_bootstrapped, y_bootstrapped)

         self.trees.append(tree)
```

Prediction:

We run predictions on all the above trees and the class which has more votes across the trees is made the final prediction. We essentially calculate the mode of all the classes predicted across the trees for each data point to make the final prediction.

```
def predict(self, X):
    predictions = []

    for tree in self.trees:
        pred = tree.predict(X).reshape((-1, 1))

        predictions.append(pred)

    return np.array(mode(np.array(predictions), axis=0)).flatten()
```

1.4.1 Parameters:

Number of trees:

- As we increased the number of trees in the forest, 10-fold accuracy increased and algorithm runtime has also been increased but the algorithm starts to overfit if number of trees has been increased to more than 10 for dataset 1.

Number of features:

- According to the results we obtained, if the dataset doesn't have a lot of features a good setting for number of features would be n or n/2 where n is total number of features in the dataset.

- If the dataset has a large number of features for example dataset 1, a good setting for number of features is √n

Size of bootstrap:

- If we have a large dataset, optionally, we can set the size of the bootstrap set to be less than the dataset size to help the algorithm run faster without affecting the accuracy.
- For dataset 1, a bootstrap size of 250 gave comparable accuracy to running the same algorithm with a bootstrap size of m where m is the number of rows in the dataset.
- Moreover, the algorithm ran around 2 times faster than running on m rows.

1.4.2 Analysis:

- Random forest gave better results for 10-fold cross validation than decision tree because of the multiple trees and random features accounting to more complex model distribution.

- It is not necessarily slower than decision tree as parameters like max_depth, n_features, n_bootstrap reduce the computational complexity of the individual trees and size of the training data.

Pros:

- Random forest is one of the most accurate algorithms that can provide great accuracy by identifying complex non-linearities in the dataset by leveraging multiple decision trees.
- It provides great results even on huge datasets since
- Even with datasets containing unbalanced labels, random forest can provide accurate prediction due to the diverse set of trees available to model different variances in data and due to the availability of majority voting to build more confidence on the prediction.

Cons:

- They are still prone to overfitting similar to decision trees with the introduction of more hyper-parameters on top of decision trees.
- They can be significantly slower due to the requirement of bagging the data, training on multiple trees and prediction using majority voting.
- Difficult to interpret due to the presence of multiple trees that are randomly initialized into complex forests.

1.5 Cross Validation:

- We used 10-fold cross validation for datasets 1 and 2 for all the above algorithms. Instead of relying solely on accuracy which can be skewed based on the imbalance of labels, we also use precision, recall and F1 score to better uncover the true accuracy of the model w.r.t various labels.

- We basically split the dataset into 10 folds where 9 folds would be combined into a training set and 1 fold is made a test set. We repeat the same process 10 times to get 10 sets of data to be trained and tested on 10 different times and we show the average of the metrics obtained from the 10 datasets.

- We do this process to reduce the chance of overfitting the training dataset and help the model generalize to real examples in the test set.

Part 2: Competition

Before we choose a machine learning algorithm we need to better understand the dataset provided, the data distribution etc. Data pre-processing is a crucial step that is often overlooked but it can provide much better results. We have seen at least a 10% percent increase in accuracy due to data pre-processing for the given dataset.

2.1 Data pre-processing:

2.1.1 Standardization:

- As a first step, to make sure that all the columns in the data are consistent and have smaller values to avoid disparities in scale between them and to avoid overflows and underflows, we perform standardization.
- Standardization rescales the columns of data to have a mean of 0 and a standard deviation of 1.
- We use sklearn's StandardScaler() to standardize the training and test features as shown below:

```
scaler = preprocessing.StandardScaler().fit(X_train)
```

```
    X_train_standardized = scaler.transform(X_train)
    X_test_standardized = scaler.transform(X_test)
```

- For algorithms like K-nn, SVM and logistic regression we have seen around 5-9% increase in accuracy just by doing standardization!
- But for algorithms like decision tree and random forest standardization doesn't make much difference as these are rule based classifiers and are independent of scale of the feature.

2.1.2 Oversampling:

- We used the follow line to determine the number of examples per label in the given training set:

```
print(sorted(Counter(list(y_train.flatten())).items()))
# [(0, 107), (1, 311)]
```

- The above line of output means that there is roughly a 3:1 ratio between positive and negative examples which means that the given dataset is rather skewed towards more positive examples.
- **Random oversampling:** To reduce the disparity between number of positive classes and negative classes, we randomly sample from the negative examples until we have 311 negative examples in our set. The following code uses "imblearn" package to do random oversampling as follows:

```
from imblearn.over_sampling import RandomOverSampler
X_resampled, y_resampled =
RandomOverSampler().fit_resample(X_train, y_train.ravel())
print(sorted(Counter(list(y_resampled.flatten())).items()))
# [(0, 311), (1, 311)]
```

- **SMOTE - Synthetic Minority Oversampling Technique**: While the RandomOverSampler is over-sampling by duplicating some of the original samples of the minority class, SMOTE generates new samples in by interpolation. Our final submission uses SMOTE to oversample the data and has provided the best accuracy of around 95% consistently across 5 folds.

2.2 Models:

- We started off training with multiple models including K-nn, Naïve Bayes, logistic regression, decision tree and random forest.

- 10-fold cross validation might lead to smaller testing data, so we used 5-fold cross validation i.e. 80% training data and 20% testing data.

2.2.1 Classic classification algorithms:

- Naïve Bayes gave the least accurate predictions of them all with training accuracy hovering around 70%. Bernoulli Naïve Bayes has provided much better training accuracy around 80% but not quite accurate when compared to other classification algorithms.

- Algorithms like K-nn, logistic regression and SVM have seen 5-9% increase in training accuracy due to standardization and they have given impressive training accuracies around 83%.

- Of all of the non-ensemble algorithms decision tree has always provided consistently better results of around 85% with the following parameters:

max_depth=5, max_features=90, min_samples_split=60, min_samples_leaf=30

where min_samples_split is minimum number of samples required at decision nodes to split and min_samples_leaf is minimum samples required at leaf node.

- These additional parameters for decision nodes and leaf nodes has provided considerable increase in accuracy for decision tree and made it the clear winner among classic classifications algorithms.

2.2.2 Ensemble algorithms:

- We implemented ensemble algorithms like random forest classifier, bagging classifier, AdaBoost classifier and Voting classifier.

- Straightaway, random forest has provided greater training accuracy than all of the classic classification algorithms discussed before and has provided the best submission on the leader board among all of them. But we have noticed that it tends to overfit the data and any further improvements in training accuracy didn’t correlate with improvement in leader board scores. We chose around 100 estimators with a maximum depth of 5, min_samples_split of 60 and min_samples_leaf of 30 to achieve training accuracies close to 88%

- Since decision tree has been the better classifier among the classic algorithms we have chosen it as a base classifier for bagging and AdaBoost. Both bagging and Adaboost with SMOTE have given nearly 90% training accuracies but couldn’t provide any better results on the leader board.

- Finally we have used Voting classifier and chose the best performing algorithms i.e. logistic regression, Bernoulli Naïve Bayes, SVM and Random forest as the different estimators. We have noticed that **soft voting has** consistently out-performed **hard voting** as soft voting predicts the class label based on the "argmax" of the sums of the predicted probabilities instead of just doing majority voting.

- Since random forest has been the best estimator we have given it more weight than other estimators to do majority voting and surely that has shown a considerable amount of improvement in accuracy.

- Ultimately, using techniques like standardization, SMOTE and Voting Classifier has given the best accuracy among all the different strategies.

- Below table summarizes the best results obtained for each algorithm:

Algorithm	**Accuracy**	**Precision**	**Recall**	**F1**
K-nn	0.83980561	0.94558594	0.8328213	0.88539684
Naïve Bayes	0.81809153	0.85329473	0.91315924	0.8821036
SVM	0.8278428	0.81892379	0.98725038	0.89513694
Logistic	0.83033782	0.88372326	0.89078341	0.88652149
Decision Tree	0.85241692	0.87829104	0.86930271	0.88678493
Random Forest	0.87284381	0.89857237	0.84419023	0.87928491
Bagging	0.8827701	0.87514141	0.89395801	0.884238
AdaBoost	0.93407578	0.95666748	0.90993344	0.93219244
Voting	0.94891449	0.93917103	0.93896569	0.93879852