CS564 Foundations of Machine Learning
Assignment: 5

Mukuntha N S (1601CS27)
R J Srivatsa (1601CS35)
Shikhar Jaiswal (1601CS44)

November 19, 2019

1 Assignment Description

Designing and implementing an ensemble classification model using stacking and boosting techniques and applying different available classifiers for the **Ensemble Dataset**.

2 Procedure

Installation

Install the following dependencies either using pip or through conda in a Python 3.5+ environment:
- jupyter
- numpy
- pandas
- sklearn

```
python3 -m pip install jupyter numpy pandas sklearn
```

or alternatively,

```
conda install -c anaconda jupyter numpy pandas sklearn
```

Running The Notebook

To run the program, head to the directory $Assignment5/Q1$ and $Assignment5/Q2$. Please note that the **dataset should be present in the assignment folder**. Use the following command to run the notebook:

```
jupyter notebook Q1.ipynb
jupyter notebook Q2.ipynb
```

3 Discussion

The primary objective of the assignment is to implement the stacking and boosting ensemble techniques on different classifiers. The following sub-sections contain the explanation for individual code snippets. For a detailed look at the output, please refer the notebook and the individual files provided.

Notebook Code - Stacking

Pre-processing and Visualization

Import the Python dependencies, and check the data samples and their values and pre-process the corpus.

```
# Import the required libraries.
import copy
import random
import numpy as np
import pandas as pd
from string import ascii_lowercase
from itertools import combinations
from sklearn.model_selection import train_test_split, KFold
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import precision_recall_fscore_support

# Set the random seed.
random.seed(15)
np.random.seed(15)

# Read the dataset.
data = pd.read_csv('../ensemble_data.csv')
data = data.apply(lambda x: x.astype(str).str.lower())

# Generate the map for categorical data.
letter_map = {letter: int(index) for index, letter in enumerate(ascii_lowercase, start = 1)}
letter_map['?'] = 0

# Make the categorical variables as numeric.
for col in data.columns:
    data[col] = data[col].map(letter_map)

# Generate the attribute and class datasets.
X = data.loc[:, data.columns != 'type']
Y = data.loc[:, data.columns == 'type']
```

```
X_Train, X_Test, Y_Train, Y_Test = train_test_split(X.values, Y.values, test_size = 0.40,
                                                    random_state = 42)
kf = KFold(n_splits = 5, shuffle = True)
```

Stacking Model Code

We split the overall dataset into training and testing dataset in a 60-40 ratio. We initialize the classifiers and use them to generate the training dataset for the meta-classifier. This is done by making use of predictions for 5-fold cross validation and using the outputs as a training dataset. Finally, we obtain the results on the base as well as meta classifiers.

```
names = ["K-Nearest Neighbors", "Linear SVM", "Decision Tree", "Random Forest",
         "Neural Net", "Naive Bayes", "Logistic Regression"]
classifiers = [KNeighborsClassifier(2),
               SVC(kernel = "linear", C = 0.025),
               DecisionTreeClassifier(max_depth = 4),
               RandomForestClassifier(max_depth = 4, n_estimators = 5, max_features = 1),
               MLPClassifier(alpha = 1, max_iter = 100),
               GaussianNB(),
               LogisticRegression(solver = 'liblinear', max_iter = 300)]
```

```
X1_Train = np.zeros((X_Train.shape[0], len(names)))
Y1_Train = np.zeros((Y_Train.shape[0], 1))

# Generate the training dataset for the Level-1 classifier.
j = 0
for train_index, val_index in kf.split(X_Train):
    X_train, X_val = X_Train[train_index], X_Train[val_index]
    Y_train, Y_val = Y_Train[train_index], Y_Train[val_index]
    preds = np.zeros((X_val.shape[0], len(names)))
    i = 0

    for name, classifier in zip(names, classifiers):
        classifier.fit(X_train, Y_train.ravel())
        pred = classifier.predict(X_val)
        preds[:, i] = pred
        i += 1

    X1_Train[j: j + X_val.shape[0], :] = preds
    Y1_Train[j: j + Y_val.shape[0],] = Y_val
    j += X_val.shape[0]
```

```
# Train the individual base models.
trained_base_classifiers = copy.deepcopy(classifiers)
for name, classifier in zip(names, trained_base_classifiers):
    classifier.fit(X_Train, Y_Train.ravel())
```

```
X1_Test = np.zeros((X_Test.shape[0], len(names)))
Y1_Test = Y_Test

# Generate the test dataset for the Level-1 classifier.
i = 0
for name, classifier in zip(names, trained_base_classifiers):
    pred = classifier.predict(X_Test)
    X1_Test[:, i] = pred
    i += 1
```

```
# Train the individual meta models.
trained_meta_classifiers = copy.deepcopy(classifiers)
for name, classifier in zip(names, trained_meta_classifiers):
    classifier.fit(X1_Train, Y1_Train.ravel())
```

```
# Output the scores of the trained base classifiers.
print("Base Learners:")
for name, classifier in zip(names, trained_base_classifiers):
    print(name + ":")
    Y_Pred = classifier.predict(X_Test)
    prf = precision_recall_fscore_support(Y_Test, Y_Pred, average = 'macro')
    score = classifier.score(X_Test, Y_Test.ravel())
    print("Accuracy: ", round(score, 4), " Precision: ", round(prf[0], 4), " Recall: ",
          round(prf[1], 4), " F-Score: ", round(prf[2], 4))
```

```
Output:
Base Learners:
K-Nearest Neighbors:
Accuracy:  0.9997  Precision:  0.9997  Recall:  0.9997  F-Score:  0.9997
Linear SVM:
Accuracy:  0.9665  Precision:  0.9663  Recall:  0.9668  F-Score:  0.9664
Decision Tree:
Accuracy:  0.9791  Precision:  0.979  Recall:  0.9794  F-Score:  0.9791
Random Forest:
Accuracy:  0.9268  Precision:  0.9288  Recall:  0.9257  F-Score:  0.9265
Neural Net:
Accuracy:  1.0  Precision:  1.0  Recall:  1.0  F-Score:  1.0
Naive Bayes:
Accuracy:  0.8643  Precision:  0.8743  Recall:  0.867  F-Score:  0.8639
Logistic Regression:
Accuracy:  0.9582  Precision:  0.958  Recall:  0.9583  F-Score:  0.9581
```

```
# Output the scores of the trained meta classifiers.
print("Meta-Learners:")
for name, classifier in zip(names, trained_meta_classifiers):
    print(name + ":")
```

```
    Y1_Pred = classifier.predict(X1_Test)
    prf = precision_recall_fscore_support(Y1_Test, Y1_Pred, average = 'macro')
    score = classifier.score(X1_Test, Y1_Test.ravel())
    print("Accuracy: ", round(score, 4), " Precision: ", round(prf[0], 4), " Recall: ",
          round(prf[1], 4), " F-Score: ", round(prf[2], 4))
```

```
Output:
Meta-Learners:
K-Nearest Neighbors:
Accuracy:  0.9997  Precision:  0.9997  Recall:  0.9997  F-Score:  0.9997
Linear SVM:
Accuracy:  1.0  Precision:  1.0  Recall:  1.0  F-Score:  1.0
Decision Tree:
Accuracy:  1.0  Precision:  1.0  Recall:  1.0  F-Score:  1.0
Random Forest:
Accuracy:  0.9895  Precision:  0.9894  Recall:  0.9899  F-Score:  0.9895
Neural Net:
Accuracy:  1.0  Precision:  1.0  Recall:  1.0  F-Score:  1.0
Naive Bayes:
Accuracy:  1.0  Precision:  1.0  Recall:  1.0  F-Score:  1.0
Logistic Regression:
Accuracy:  1.0  Precision:  1.0  Recall:  1.0  F-Score:  1.0
```

Notebook Code - Adaboost

Pre-processing and Visualization

Import the Python dependencies, and check the data samples and their values and pre-process the corpus.

```
# Import the required libraries.
import copy
import math
import queue
import random
import numpy as np
import pandas as pd
from scipy import stats
import matplotlib.pyplot as plt
from collections import Counter
from string import ascii_lowercase
from itertools import combinations
from sklearn.model_selection import train_test_split, KFold
from sklearn.metrics import precision_recall_fscore_support

# Set the random seed.
random.seed(14)
np.random.seed(14)
```

```
# Read the dataset.
data = pd.read_csv('../ensemble_data.csv')
data = data.apply(lambda x: x.astype(str).str.lower())

# Generate the attribute and class datasets.
X = data.loc[:, data.columns != 'type']
Y = data.loc[:, data.columns == 'type']
X.insert(loc = 0, column = 'dummy', value = 0)
Y = (Y != 'e').astype(np.int32)
columns = X.columns[1:]

for col in columns:
    X = pd.concat([X, pd.get_dummies(X[col], prefix = col)], axis = 1)
    X.drop([col], axis = 1, inplace = True)

X = X.values
Y = Y.values

kf = KFold(n_splits = 5, shuffle = True)
```

Decision Tree Model Code

We split the overall dataset into training and validation dataset in a 5-fold cross validation ratio. We build the decision tree from the training dataset and finally, we obtain the results on the validation dataset for the decision tree.

```
class Node():
    cnt = 0

    def __init__(self, ):
        self.leaf = False
        self.majority_class = None
        self.attribute_index = None
        # Key: attribute_value, Value: child_node
        self.children = dict()
        # Used at inference time, if attribute_index is 0
        self.sent_len_split_val = None
        self.id = Node.cnt
        Node.cnt += 1

    def __str__(self,):
        return 'ID: {}  isLeaf: {} majority: {}\
                split_idx: {} split_val = {}'.format(self.id, self.leaf,
                self.majority_class, self.attribute_index, list(self.children.keys()))

    def __repr__(self):
        return str(self)
```

```
    def traverse_print(self,):
        print(self)
        for _, child in self.children:
              child.traverse_print()

    @classmethod
    def reset_cnt(cls,):
        cls.cnt = 0

class DecisionTree():

    # Score has to be from 'entropy', 'gini', 'misclassification'.
    def __init__(self, score = 'entropy'):
        score_functions = {'entropy': (DecisionTree.compute_entropy,
                                       DecisionTree.get_gain_entropy),
                           'gini': (DecisionTree.compute_gini, DecisionTree.get_gain_gini),
                           'misclassification': (DecisionTree.compute_misclassification,
                                                 DecisionTree.get_gain_misclassification)}
        self.root = None
        assert score in score_functions.keys()
        self.score = score
        self.compute_score = score_functions[score][0]
        self.get_gain = score_functions[score][1]

    @staticmethod
    def compute_entropy(labels):
        entropy = 0.0
        totSamples = len(labels)
        labelSet = set(labels.reshape(-1))
        for label in labelSet:
            prob = np.sum(labels == label) / totSamples
            if prob > 1e-12:
                entropy -= np.log(prob) * prob

        return entropy

    @staticmethod
    def get_gain_entropy(parent_info, data_i, labels):
        attr_split_info = 0
        attr_count = dict()
        for attr_val in set(data_i.reshape(-1)):
            ids = np.where(data_i == attr_val)[0]
            attr_count[attr_val] = len(ids)
            attr_split_info += attr_count[attr_val] *
                               DecisionTree.compute_entropy(labels[ids])
        attr_gain = parent_info - attr_split_info
        attr_gain_ratio = DecisionTree.compute_dict_entropy(attr_count) * attr_gain
        return attr_gain, attr_gain_ratio, attr_count.keys()
```

```
    @staticmethod
    def compute_dict_entropy(attr_count):
        entropy = 0
        totSamples = sum(attr_count.values())

        labelSet = attr_count.keys()
        for label in labelSet:
            prob = attr_count[label] / totSamples
            if prob > 1e-12:
                entropy -= np.log(prob) * prob
        return entropy

    @staticmethod
    def compute_gini(labels):
        prob_sq = 0.0
        totSamples = len(labels)
        labelSet = set(labels.reshape(-1))
        for label in labelSet:
            prob = np.sum(labels == label) / totSamples
            if prob > 1e-12:
                prob_sq += prob*prob
        return 1-prob_sq


    @staticmethod
    def get_gain_gini(parent_info, data_i, labels):
        attr_split_info = 0
        attr_count = dict()

        for attr_val in set(data_i.reshape(-1)):
            ids = np.where(data_i == attr_val)[0]
            attr_count[attr_val] = len(ids)
            attr_split_info += attr_count[attr_val] * DecisionTree.compute_gini(labels[ids])

        attr_split_info /= data_i.shape[0]
        attr_gain = parent_info - attr_split_info
        return attr_gain, attr_gain, attr_count.keys()

    @staticmethod
    def compute_misclassification(labels):
        max_prob = -1
        totSamples = len(labels)
        labelSet = set(labels.reshape(-1))
        for label in labelSet:
            prob = np.sum(labels == label) / totSamples
            if prob > max_prob:
                max_prob = prob
```

```
        return 1-max_prob

    @staticmethod
    def get_gain_misclassification(parent_info, data_i, labels):
        attr_split_info = -1
        attr_count = dict()

        for attr_val in set(data_i.reshape(-1)):
            ids = np.where(data_i == attr_val)[0]
            attr_count[attr_val] = len(ids)
            attr_split_info = attr_count[attr_val] *
                              DecisionTree.compute_misclassification(labels[ids])

        attr_split_info /= data_i.shape[0]
        attr_gain = parent_info - attr_split_info
        return attr_gain, attr_gain, attr_count.keys()

    def split_node(self, parent, data, labels, used_attr_index):
        num_instances = data.shape[0]
        parent_info = self.compute_score(labels) * num_instances
        parent.majority_class = Counter(labels.reshape(-1)).most_common(1)[0][0]

        if parent_info == 0 :
            parent.leaf = True

        best_attr_index = None
        best_info_gain = -float('inf')
        best_gain_ratio = -float('inf')
        best_attr_keys = None
        sent_len_split_val = stats.mode(data[:, 0])[0][0]
        le_ids = np.where(data[:, 0] <= sent_len_split_val)[0]
        gt_ids = np.where(data[:, 0] > sent_len_split_val)[0]
        data_0 = np.zeros(data.shape[0], dtype=np.int32)
        data_0[gt_ids] = 1
        attr_gain, attr_gain_ratio, attr_count_keys = self.get_gain(parent_info, data_0,
                                                                    labels)

        if best_gain_ratio < attr_gain_ratio and  attr_gain_ratio > 0 :
                best_attr_index = 0
                best_info_gain = attr_gain
                best_gain_ratio = attr_gain_ratio
                best_attr_keys = attr_count_keys

        for i in range(1, data.shape[1]):
            if i in used_attr_index:
                continue
```

```
            attr_gain, attr_gain_ratio, attr_count_keys = self.get_gain(parent_info,
                                                                        data[:, i], labels)
            if best_gain_ratio < attr_gain_ratio:
                best_attr_index = i
                best_info_gain = attr_gain
                best_gain_ratio = attr_gain_ratio
                best_attr_keys = attr_count_keys

        if best_gain_ratio <= 0 or len(best_attr_keys) == 1 :
            parent.leaf = True
            return []
        else:
            parent.attribute_index =  best_attr_index
            parent.children = { i: Node() for i in best_attr_keys}
            to_return = []

            if best_attr_index != 0:
                used_attr_index.append(best_attr_index)

                for i in best_attr_keys:
                    inds = np.where(data[:, best_attr_index] == i)[0]
                    to_return.append((parent.children[i], data[inds], labels[inds],
                                      used_attr_index))
            else:
                parent.sent_len_split_val = sent_len_split_val

                for i in best_attr_keys:
                    inds = np.where(data_0 == i)[0]
                    to_return.append((parent.children[i], data[inds], labels[inds],
                                      used_attr_index))

            return to_return

    def build_tree(self, data, labels):
        traversal_q = queue.Queue()
        root = Node()
        self.root = root
        traversal_q.put_nowait( (root, data, labels, [] ))

        while not traversal_q.empty():
            node_to_split = traversal_q.get_nowait()
            child_nodes = self.split_node(*node_to_split)
            for child in child_nodes:
                traversal_q.put_nowait(child)

        return root

    def split_infer(self, node, data, data_indices):
```

```
        if node.leaf:
            return (True, data_indices, np.zeros((data.shape[0]), dtype = np.int32) +
                    node.majority_class)
        else:
            to_queue = []
            if (node.attribute_index == 0):
                left_idx = np.where(data[:,0] <= node.sent_len_split_val)[0]
                right_idx = np.where(data[:,0] > node.sent_len_split_val)[0]
                to_queue.append((node.children[0], data[left_idx], data_indices[left_idx]))
                to_queue.append((node.children[1], data[right_idx], data_indices[right_idx]))
                return (False, to_queue)
            else:
                for i in node.children.keys():
                    split_inds = np.where( data[:, node.attribute_index]  == i)[0]
                    if len(split_inds) > 0:
                        to_queue.append((node.children[i], data[split_inds],
                                         data_indices[split_inds]))
                return (False, to_queue)

    def split_infer_depth(self, node, data, data_indices, depth):
        if node.leaf or depth >= 3:
            return (True, data_indices, np.zeros( (data.shape[0]), dtype = np.int32) +
                    node.majority_class)
        else:
            to_queue = []
            if (node.attribute_index == 0):
                left_idx = np.where(data[:, 0] <= node.sent_len_split_val)[0]
                right_idx = np.where(data[:, 0] > node.sent_len_split_val)[0]
                to_queue.append((node.children[0], data[left_idx], data_indices[left_idx],
                                 depth + 1))
                to_queue.append((node.children[1], data[right_idx], data_indices[right_idx],
                                 depth + 1))
                return (False, to_queue)
            else:
                for i in node.children.keys():
                    split_inds = np.where( data[:, node.attribute_index]  == i)[0]
                    if len(split_inds) > 0:
                        to_queue.append((node.children[i], data[split_inds],
                                         data_indices[split_inds], depth + 1))
                return (False, to_queue)

    def get_labels(self, data):
        root = self.root
        data_idx = np.arange(data.shape[0], dtype = np.int32)
        labels = np.zeros( (data.shape[0]), dtype = np.int32) + -1
        traversal_q = queue.Queue()
        traversal_q.put_nowait( (root, data, data_idx ))
        while not traversal_q.empty():
```

```
            node_to_split = traversal_q.get_nowait()
            split_return = self.split_infer(*node_to_split)
            if split_return[0]:
                labels[split_return[1]] = split_return[2]
            else:
                for child in split_return[1]:
                    traversal_q.put_nowait(child)
        return labels

    def get_labels_depth(self, data):
        root = self.root
        data_idx = np.arange(data.shape[0], dtype = np.int32)
        labels = np.zeros( (data.shape[0]), dtype = np.int32) + -1
        traversal_q = queue.Queue()
        traversal_q.put_nowait( (root, data, data_idx ,0))
        while not traversal_q.empty():
            node_to_split = traversal_q.get_nowait()
            split_return = self.split_infer_depth(*node_to_split)
            if split_return[0]:
                labels[split_return[1]] = split_return[2]
            else:
                for child in split_return[1]:
                    traversal_q.put_nowait(child)
        return labels
```

```
def get_scores(Y_test_idx, y_pred_test, average = 'macro'):
    acc = (y_pred_test == Y_test_idx).mean()
    prec, rec, fscore, _ = precision_recall_fscore_support(Y_test_idx, y_pred_test,
                                                           average = average)
    return round(acc, 5), round(prec, 5), round(rec, 5), round(fscore, 5)

dtree = DecisionTree('misclassification')
decision_tree_scores = []

print('For 5-Fold Cross Validation')
for train_index, val_index in kf.split(X):
    X_train, X_val = X[train_index], X[val_index]
    Y_train, Y_val = Y[train_index], Y[val_index]

    root = dtree.build_tree(data = X_train, labels = Y_train)
    print(dtree.root)

    y_pred_test = dtree.get_labels(data = X_val)
    decision_tree_scores.append(get_scores(Y_val.ravel(), y_pred_test))
    print('Acc: {}, Prec: {}, Rec: {}, Fscore: {}\n'.format(*get_scores(Y_val.ravel(),
                                                                        y_pred_test)))
```

```
Output:
```

```
For 5-Fold Cross Validation
ID: 0  isLeaf: False majority: 0 split_idx: 2 split_val = [0, 1]
Acc: 0.99938, Prec: 0.99943, Rec: 0.99934, Fscore: 0.99938

ID: 87  isLeaf: False majority: 0 split_idx: 2 split_val = [0, 1]
Acc: 1.0, Prec: 1.0, Rec: 1.0, Fscore: 1.0

ID: 174  isLeaf: False majority: 0 split_idx: 2 split_val = [0, 1]
Acc: 1.0, Prec: 1.0, Rec: 1.0, Fscore: 1.0

ID: 261  isLeaf: False majority: 0 split_idx: 2 split_val = [0, 1]
Acc: 1.0, Prec: 1.0, Rec: 1.0, Fscore: 1.0

ID: 348  isLeaf: False majority: 0 split_idx: 2 split_val = [0, 1]
Acc: 1.0, Prec: 1.0, Rec: 1.0, Fscore: 1.0
```

Adaboost Model Code

We split the overall dataset into training and validation dataset in a 5-fold cross validation ratio. We build the adaboost model using 5 decision tree classifiers, and then we obtain the results on the validation dataset.

```
# Decision Stump is used as classifier in this implementation of Adaboost.
class DecisionStump(DecisionTree):

    def __init__(self):
        self.polarity = 1
        self.feature_index = None
        self.threshold = None
        self.alpha = None

class Adaboost():

    def __init__(self, n_clf = 5):
        self.n_clf = n_clf

    def fit(self, X, Y):
        n_samples, n_features = np.shape(X)
        w = np.full(n_samples, (1 / n_samples))

        self.clfs = []
        for _ in range(self.n_clf):
            clf = DecisionStump()
            min_error = float('inf')

            for feature_i in range(n_features):
                feature_values = np.expand_dims(X[:, feature_i], axis = 1)
                unique_values = np.unique(feature_values)
```

```
                for threshold in unique_values:
                    p = 1
                    prediction = np.ones(np.shape(Y))
                    prediction[X[:, feature_i] <= threshold] = 0
                    error = sum(w[Y != prediction])

                    if error > 0.5:
                        error = 1 - error
                        p = -1

                    if error < min_error:
                        clf.polarity = p
                        clf.threshold = threshold
                        clf.feature_index = feature_i
                        min_error = error

            clf.alpha = 0.5 * math.log((1.0 - min_error) / (min_error + 1e-10))
            predictions = np.ones(np.shape(Y))
            negative_idx = (clf.polarity * X[:, clf.feature_index] <=
                            clf.polarity * clf.threshold)
            predictions[negative_idx] = 0
            w *= np.exp(-clf.alpha * Y * predictions.ravel())
            w /= np.sum(w)
            self.clfs.append(clf)

    def predict(self, X):
        n_samples = np.shape(X)[0]
        y_pred = np.zeros((n_samples, 1))

        for clf in self.clfs:
            predictions = np.ones(np.shape(y_pred))
            negative_idx = (clf.polarity * X[:, clf.feature_index] <=
                            clf.polarity * clf.threshold)
            predictions[negative_idx] = 0
            y_pred += clf.alpha * predictions

        y_pred = np.sign(y_pred).flatten()
        return y_pred
```

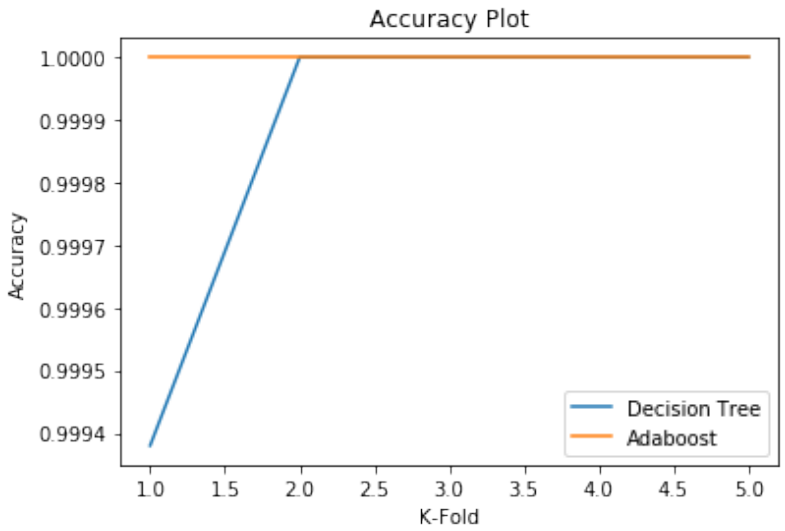
```
# Adaboost classification with 5 weak classifiers.
adaboost_scores = []

print('For 5-Fold Cross Validation')
for train_index, val_index in kf.split(X):
    X_train, X_val = X[train_index], X[val_index]
    Y_train, Y_val = Y[train_index], Y[val_index]

    ada = Adaboost(n_clf = 5)
```

```
    ada.fit(X_train, Y_train.ravel())
    y_pred_test = dtree.get_labels(data = X_val)

    adaboost_scores.append(get_scores(Y_val.ravel(), y_pred_test))
    print('Acc: {}, Prec: {}, Rec: {}, Fscore: {}\n'.format(*get_scores(Y_val.ravel(),
                                                                        y_pred_test)))
```

```
Output:
For 5-Fold Cross Validation
Acc: 1.0, Prec: 1.0, Rec: 1.0, Fscore: 1.0

Acc: 1.0, Prec: 1.0, Rec: 1.0, Fscore: 1.0

Acc: 1.0, Prec: 1.0, Rec: 1.0, Fscore: 1.0

Acc: 1.0, Prec: 1.0, Rec: 1.0, Fscore: 1.0

Acc: 1.0, Prec: 1.0, Rec: 1.0, Fscore: 1.0
```

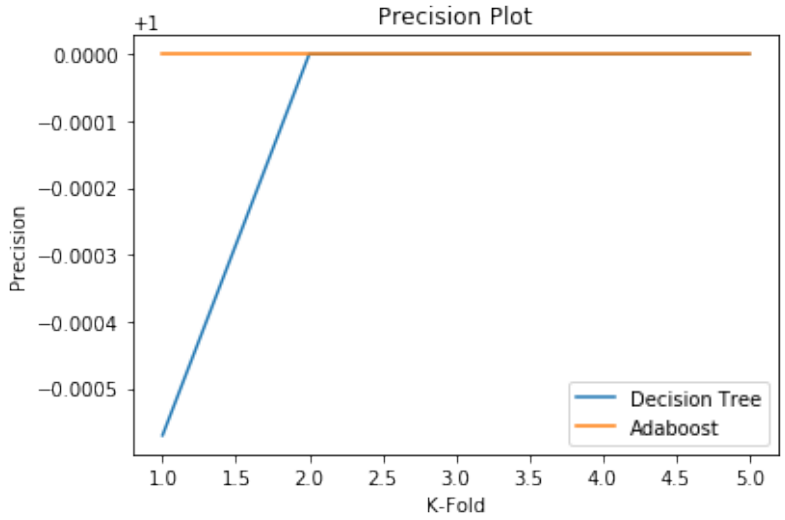
```
x = [1, 2, 3, 4, 5]
y1 = [elem[0] for elem in decision_tree_scores]
plt.plot(x, y1, label = "Decision Tree")
y2 = [elem[0] for elem in adaboost_scores]
plt.plot(x, y2, label = "Adaboost")
plt.xlabel('K-Fold')
plt.ylabel('Accuracy')
plt.title('Accuracy Plot')
plt.legend()
plt.show()
```

Plot



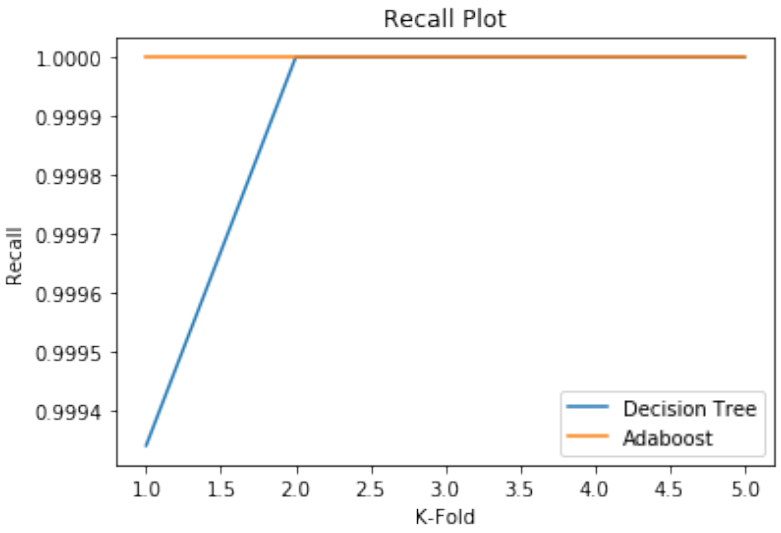

```
y1 = [elem[1] for elem in decision_tree_scores]
plt.plot(x, y1, label = "Decision Tree")
y2 = [elem[1] for elem in adaboost_scores]
plt.plot(x, y2, label = "Adaboost")
plt.xlabel('K-Fold')
plt.ylabel('Precision')
plt.title('Precision Plot')
plt.legend()
plt.show()
```

Plot



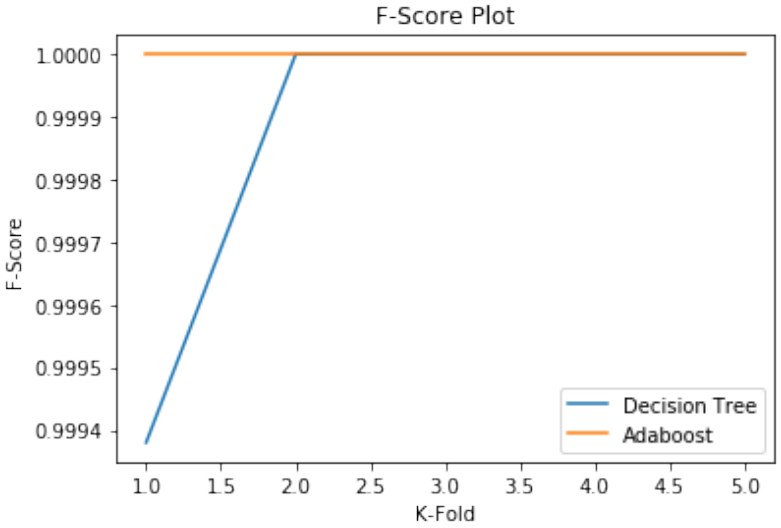
```
y1 = [elem[2] for elem in decision_tree_scores]
plt.plot(x, y1, label = "Decision Tree")
y2 = [elem[2] for elem in adaboost_scores]
plt.plot(x, y2, label = "Adaboost")
plt.xlabel('K-Fold')
plt.ylabel('Recall')
plt.title('Recall Plot')
plt.legend()
plt.show()
```

Plot




```
y1 = [elem[3] for elem in decision_tree_scores]
plt.plot(x, y1, label = "Decision Tree")
y2 = [elem[3] for elem in adaboost_scores]
plt.plot(x, y2, label = "Adaboost")
plt.xlabel('K-Fold')
plt.ylabel('F-Score')
plt.title('F-Score Plot')
plt.legend()
plt.show()
```

Plot



Additionally please refer $Q1/Q1.ipynb$ and $Q2/Q2.ipynb$ for the outputs.