In [1]:
```
import numpy as np
from numpy import genfromtxt
import scipy.io
from scipy import stats
from sklearn.tree import DecisionTreeClassifier
from sklearn.base import BaseEstimator, ClassifierMixin
import pandas as pd
from random import randint

eps = 1e-5  # a small number


class DecisionTree:

    def __init__(self, max_depth=3, feature_labels=None):
        self.max_depth = max_depth
        self.features = feature_labels
        self.left, self.right = None, None  # for non-leaf nodes
        self.split_idx, self.thresh = None, None  # for non-leaf nodes
        self.data, self.pred = None, None  # for leaf nodes

    @staticmethod
    def entropy(y):
        # TODO implement entropy function
        n = y.size
        hash_map = {}
        for i in y:
            hash_map[i] = 1 if i not in hash_map else hash_map[i] + 1
            # if i not in hash_map:
            #     hash_map[i] = 1
            # else:
            #     hash_map[i] += 1
        ret = 0
        for i in hash_map:
            ret -= hash_map[i] / n * np.log(hash_map[i] / n)
        return ret


    @staticmethod
    def information_gain(X, y, thresh):
        # TODO implement information gain function
        # print(X)
        index_0 = np.where(X < thresh)[0]
        index_1 = np.where(X >= thresh)[0]
        y0, y1 = y[index_0], y[index_1]
        n0, n1 = y0.size, y1.size
        return DecisionTree.entropy(y) - ( n0 / (n0 + n1) * DecisionTree.ent
ropy(y0) + n1 / (n0 + n1) * DecisionTree.entropy(y1) )

    def split(self, X, y, idx, thresh):
        X0, idx0, X1, idx1 = self.split_test(X, idx=idx, thresh=thresh)
        y0, y1 = y[idx0], y[idx1]
        return X0, y0, X1, y1

    def split_test(self, X, idx, thresh):
```

```
        idx0 = np.where(X[:,idx] < thresh)[0]
        idx1 = np.where(X[:,idx] >= thresh)[0]
        X0, X1 = X[idx0, :], X[idx1, :]
        return X0, idx0, X1, idx1

    def fit(self, X, y):
        if self.max_depth > 0:
            # compute entropy gain for all single-dimension splits,
            # thresholding with a linear interpolation of 10 values
            gains = []
            thresh = np.array([np.linspace(np.min(X[:, i]) + eps,
                                           np.max(X[:, i]) - eps, num=10) fo
r i
                               in range(X.shape[1])])
            for i in range(X.shape[1]):
                gains.append([self.information_gain(X[:, i], y, t) for t in
                              thresh[i, :]])

            gains = np.nan_to_num(np.array(gains))
            self.split_idx, thresh_idx = np.unravel_index(np.argmax(gains),
                                                          gains.shape)
            self.thresh = thresh[self.split_idx, thresh_idx]
            X0, y0, X1, y1 = self.split(X, y, idx=self.split_idx,
                                        thresh=self.thresh)
            if X0.size > 0 and X1.size > 0:
                self.left = DecisionTree(max_depth=self.max_depth-1,
                                         feature_labels=self.features)
                self.left.fit(X0, y0)
                self.right = DecisionTree(max_depth=self.max_depth-1,
                                          feature_labels=self.features)
                self.right.fit(X1, y1)
            else:
                self.max_depth = 0
                self.data, self.labels = X, y
                self.pred = stats.mode(y).mode[0]
        else:
            self.data, self.labels = X, y
            self.pred = stats.mode(y).mode[0]
        return self

    def predict(self, X):
        if self.max_depth == 0:
            return self.pred * np.ones(X.shape[0])
        else:
            X0, idx0, X1, idx1 = self.split_test(X, idx=self.split_idx,
                                                 thresh=self.thresh)
            yhat = np.zeros(X.shape[0], dtype="int")
            yhat[idx0] = self.left.predict(X0)
            yhat[idx1] = self.right.predict(X1)
            return yhat


class BaggedTrees(BaseEstimator, ClassifierMixin):

    def __init__(self, params=None, n=200):
        if params is None:
            params = {}
```

```
        self.params = params
        self.n = n
        self.decision_trees = [
            DecisionTreeClassifier(random_state=i, **self.params) for i in
            range(self.n)]

    def fit(self, X, y):
        row_num, _ = X.shape
        for i in range(self.n):
            X_sampling = []
            y_sampling = []
            for _ in range(row_num):
                index = randint(0, row_num - 1)
                X_sampling.append(X[index])
                y_sampling.append(y[index])
            X_sampling = np.array(X_sampling)
            y_sampling = np.array(y_sampling)
            self.decision_trees[i].fit(X_sampling, y_sampling)


    def predict(self, X):
        ret = []
        for i in range(self.n):
            ret.append(self.decision_trees[i].predict(X))
        ret = np.array(ret)
        ret = np.mean(ret, axis=0)
        array_round = np.vectorize(lambda x: int(round(x)))
        return array_round(ret)


class RandomForest(BaggedTrees):

    def __init__(self, params=None, n=200, m=1):
        if params is None:
            params = {}
        # TODO implement function
        self.params = params
        self.n = n
        self.m = m
        self.decision_trees = [
            DecisionTreeClassifier(random_state=i, **self.params) for i in
            range(self.n)]
        self.features_list = []


    def fit(self, X, y):
        row_num, feature_num = X.shape
        for i in range(self.n):
            self.features_list.append([randint(0, feature_num - 1) for _ in
range(self.m)])
            X_sampling = []
            y_sampling = []
            for _ in range(row_num):
                index = randint(0, row_num - 1)
                X_sampling.append(X[index, self.features_list[i]])
                y_sampling.append(y[index])
            X_sampling = np.array(X_sampling)
```

```
            y_sampling = np.array(y_sampling)
            self.decision_trees[i].fit(X_sampling, y_sampling)


    def predict(self, X):
        ret = []
        for i in range(self.n):
            ret.append(self.decision_trees[i].predict(X[:, self.features_lis
t[i]]))
        ret = np.array(ret)
        ret = np.mean(ret, axis=0)
        array_round = np.vectorize(lambda x: int(round(x)))
        return array_round(ret)


class BoostedRandomForest(RandomForest):

    def fit(self, X, y):
        self.w = np.ones(X.shape[0]) / X.shape[0]  # Weights on data
        self.a = np.zeros(self.n)  # Weights on decision trees
        # TODO implement function
        return self

    def predict(self, X):
        # TODO implement function
        pass


def preprocess_titanic_data(data_path):
    df = pd.read_csv(data_path)
    df.drop(["ticket", "cabin"], axis=1, inplace=True)

    row_indices = []
    for i, row in df.iterrows():
        if pd.isnull(row).all():
            row_indices.append(i)

    df.drop(df.index[row_indices], inplace=True)

    for i, row in df.iterrows():
        df.at[i, "sex"] = 0 if row["sex"] == "female" else 1

    df.at[df.age.isnull(), "age"] = df["age"].mean()
    df.at[df.fare.isnull(), "fare"] = df[df.pclass == 1]["fare"].mean()

    df["e1"], df["e2"], df["e3"] = [0, 0, 0]
    for i, row in df.iterrows():
        if row["embarked"] == 'C':
            df.at[i, "e1"] = 1
        elif row["embarked"] == 'Q':
            df.at[i, "e2"] = 1
        else:
            df.at[i, "e3"] = 1

    df.drop("embarked", axis=1, inplace=True)

    data_path_list = data_path.split('/')
```

```
    data_path_list[-1] = "preprocessed_" + data_path_list[-1]
    preprocessed_data_path = '/'.join(data_path_list)
    df.to_csv(preprocessed_data_path, index=False)
    return preprocessed_data_path


if __name__ == "__main__":
    dataset = "spam"
    params = {
        "max_depth": 20,
        # "random_state": 6,
        "min_samples_leaf": 10,
    }

    if dataset == "titanic":
        # Load titanic data
        path_train = 'datasets/titanic/titanic_training.csv'
        data = genfromtxt(path_train, delimiter=',', dtype=None)
        features = data[0, 1:]  # features = all columns except survived
        y = data[1:, 0]  # label = survived
        class_names = ["Died", "Survived"]

        # TODO implement preprocessing of Titanic dataset
        preprocessed_training_path = preprocess_titanic_data(path_train)
        path_test = 'datasets/titanic/titanic_testing_data.csv'
        preprocessed_test_path = preprocess_titanic_data(path_test)

        training_data = genfromtxt(preprocessed_training_path, delimiter=','
)
        test_data = genfromtxt(preprocessed_test_path, delimiter=',')
        X, Z = training_data[1:, 1:], test_data[1:, :]
        y = training_data[1:, 0]
        y.astype(int)
    elif dataset == "spam":
        features = ["pain", "private", "bank", "money", "drug", "spam",
                    "prescription", "creative", "height", "featured", "diffe
r",
                    "width", "other", "energy", "business", "message",
                    "volumes", "revision", "path", "meter", "memo", "plannin
g",
                    "pleased", "record", "out", "semicolon", "dollar", "shar
p",
                    "exclamation", "parenthesis", "square_bracket", "ampersa
nd"]
        assert len(features) == 32

        # Load spam data
        path_train = 'datasets/spam_data/spam_data.mat'
        data = scipy.io.loadmat(path_train)
        X = data['training_data']
        y = np.squeeze(data['training_labels'])
        Z = data['test_data']
        class_names = ["Ham", "Spam"]
    else:
        raise NotImplementedError("Dataset %s not handled" % dataset)

    print("Features", features)
```

```
print("Train/test size", X.shape, Z.shape)

print("\nPart 0: constant classifier")
print("Accuracy", 1 - np.sum(y) / y.size)

# Basic decision tree
print("\nPart (a-b): simplified decision tree")
dt = DecisionTree(max_depth=3, feature_labels=features)
dt.fit(X, y)
print("Predictions", dt.predict(Z)[:100])
print()

# TODO implement and evaluate parts c-h

print("========== Question 4.c ==========")

print()
print("Titanic ====>")
dataset == "titanic"

path_train = 'datasets/titanic/titanic_training.csv'
data = genfromtxt(path_train, delimiter=',', dtype=None)
features = data[0, 1:]  # features = all columns except survived
y = data[1:, 0]  # label = survived
class_names = ["Died", "Survived"]

preprocessed_training_path = preprocess_titanic_data(path_train)
path_test = 'datasets/titanic/titanic_testing_data.csv'
preprocessed_test_path = preprocess_titanic_data(path_test)

training_data = genfromtxt(preprocessed_training_path, delimiter=',')
test_data = genfromtxt(preprocessed_test_path, delimiter=',')
X, Z = training_data[1:, 1:], test_data[1:, :]
y = training_data[1:, 0]
y.astype(int)

print("Features", features)
print("Train/test size", X.shape, Z.shape)
print()
print("Part (c): simplified decision tree - titanic")
dt = DecisionTree(max_depth=10, feature_labels=features)
dt.fit(X, y)
y_predicted = dt.predict(X)
count = 0
for i, e in enumerate(y):
    count += 1 if abs(y[i] - y_predicted[i]) < 0.01 else 0
print("Accuracy", count/y.size)
y_predicted = dt.predict(Z)
with open("submission_titanic_simpified.txt", "w") as f:
    for i in y_predicted:
        f.write(str(i) + "\n")

print()
print("Part (e): bagged - titanic")
dt = BaggedTrees(params)
dt.fit(X, y)
y_predicted = dt.predict(X)
```

```
    count = 0
    for i, e in enumerate(y):
        count += 1 if abs(y[i] - y_predicted[i]) < 0.01 else 0
    print("Accuracy", count/y.size)
    y_predicted = dt.predict(Z)
    with open("submission_titanic_bagged.txt", "w") as f:
        for i in y_predicted:
            f.write(str(i) + "\n")

    print()
    print("Part (g): random forest - titanic")
    dt = RandomForest(params)
    dt.fit(X, y)
    y_predicted = dt.predict(X)
    count = 0
    for i, e in enumerate(y):
        count += 1 if abs(y[i] - y_predicted[i]) < 0.01 else 0
    print("Accuracy", count/y.size)
    y_predicted = dt.predict(Z)
    with open("submission_titanic_randomforest.txt", "w") as f:
        for i in y_predicted:
            f.write(str(i) + "\n")

    print()
    print("Spam ====>")
    dataset == "spam"

    features = ["pain", "private", "bank", "money", "drug", "spam",
                    "prescription", "creative", "height", "featured", "diffe
r",
                    "width", "other", "energy", "business", "message",
                    "volumes", "revision", "path", "meter", "memo", "plannin
g",
                    "pleased", "record", "out", "semicolon", "dollar", "shar
p",
                    "exclamation", "parenthesis", "square_bracket", "ampersa
nd"]
    assert len(features) == 32

    # Load spam data
    path_train = 'datasets/spam_data/spam_data.mat'
    data = scipy.io.loadmat(path_train)
    X = data['training_data']
    y = np.squeeze(data['training_labels'])
    y.astype(int)
    Z = data['test_data']
    class_names = ["Ham", "Spam"]

    print("Features", features)
    print("Train/test size", X.shape, Z.shape)
    print()
    print("Part (c): simplified decision tree - spam")
    dt = DecisionTree(max_depth=5, feature_labels=features)
    dt.fit(X, y)
    y_predicted = dt.predict(X)
    count = 0
    for i, e in enumerate(y):
```

```
    count += 1 if abs(y[i] - y_predicted[i]) < 0.01 else 0
print("Accuracy", count/y.size)
y_predicted = dt.predict(Z)
with open("submission_spam_simpified.txt", "w") as f:
    for i in y_predicted:
        f.write(str(i) + "\n")

print()
print("Part (e): bagged - spam")
dt = BaggedTrees(params)
dt.fit(X, y)
y_predicted = dt.predict(X)
count = 0
for i, e in enumerate(y):
    count += 1 if abs(y[i] - y_predicted[i]) < 0.01 else 0
print("Accuracy", count/y.size)
y_predicted = dt.predict(Z)
with open("submission_spam_bagged.txt", "w") as f:
    for i in y_predicted:
        f.write(str(i) + "\n")

print()
print("Part (g): random forest - spam")
dt = RandomForest(params)
dt.fit(X, y)
y_predicted = dt.predict(X)
count = 0
for i, e in enumerate(y):
    count += 1 if abs(y[i] - y_predicted[i]) < 0.01 else 0
print("Accuracy", count/y.size)
y_predicted = dt.predict(Z)
with open("submission_spam_randomforest.txt", "w") as f:
    for i in y_predicted:
        f.write(str(i) + "\n")
```

```
Features ['pain', 'private', 'bank', 'money', 'drug', 'spam', 'prescription',
'creative', 'height', 'featured', 'differ', 'width', 'other', 'energy', 'busi
ness', 'message', 'volumes', 'revision', 'path', 'meter', 'memo', 'planning',
'pleased', 'record', 'out', 'semicolon', 'dollar', 'sharp', 'exclamation', 'p
arenthesis', 'square_bracket', 'ampersand']
Train/test size (5172, 32) (5857, 32)

Part 0: constant classifier
Accuracy 0.709976798144

Part (a-b): simplified decision tree
Predictions [0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 1
1 0 0 1 0
 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 1 0 1 0 0 1 0 0 1 0 0
 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 1 1 0 1]

========== Question 4.c ==========

Titanic ====>
Features [b'pclass' b'sex' b'age' b'sibsp' b'parch' b'ticket' b'fare' b'cabi
n'
 b'embarked']
Train/test size (999, 9) (310, 9)

Part (c): simplified decision tree - titanic
Accuracy 0.8698698698698699

Part (e): bagged - titanic
Accuracy 0.8638638638638638

Part (g): random forest - titanic
Accuracy 0.6456456456456456

Spam ====>
Features ['pain', 'private', 'bank', 'money', 'drug', 'spam', 'prescription',
'creative', 'height', 'featured', 'differ', 'width', 'other', 'energy', 'busi
ness', 'message', 'volumes', 'revision', 'path', 'meter', 'memo', 'planning',
'pleased', 'record', 'out', 'semicolon', 'dollar', 'sharp', 'exclamation', 'p
arenthesis', 'square_bracket', 'ampersand']
Train/test size (5172, 32) (5857, 32)

Part (c): simplified decision tree - spam
Accuracy 0.8002706883217324

Part (e): bagged - spam
Accuracy 0.8451276102088167

Part (g): random forest - spam
Accuracy 0.7099767981438515
```