Hiva Mohammadzadeh

3036919598

Question 1: Honor Code

> *"I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted."*

Question 4: Decision Trees for Classification

Set the random seed to 150 for titanic and 200 for spam

Part 1: Implement Decision Trees:

Code in appendix.

Part 2: Implement a Random Forest:

Code in appendix.

Part 3: Describe Implementation Details:

1. Categorial features and missing values: I didn't change the preprocessing skeleton code so much. So, I just one-hot some defined set of categorical features and did some imputation of missing values. I just replaced the missing data values with the mode value as it made more sense for the categorial features.
2. Stopping criterion: I just go until the maximum depth value.
3. Random Forest: I just used a list to hold all the decision trees. I also used m for random forest. Didn't add any additional functionality. My Decision Tree and Bagged Decision Tree implementation had everything needed for it.
4. Any Speedup: Not really. My implementation met the requirements.
5. Anything else: I just hyperparameter tuned everything with cross validation.

Part 4: Performance Evaluation:

- **Titanic:**

Given code output:

```
TITANIC DATASET

Part (b): preprocessing the titanic dataset
Titanic Features: ['pclass', 'sex', 'age', 'sibsp', 'parch', 'ticket', 'fare', 'cabin', 'embarked', 'male', 'female', 'S', 'C'
, 'Q']
Train/test size: (999, 14) (310, 14)


Part 0: constant classifier
Accuracy 0.6166166166166166
```

  - Decision Tree

```
Training Base Decision Tree with tuned depth
Base Decision Tree Training Accuracy: 0.7985714285714286, Validation Accuracy: 0.782608695652174

Training Bagged Decision Tree
Bagged Decision Tree Training Accuracy: 0.9471428571428572, Validation Accuracy: 0.7792642140468228
```

  - Random Forest

```
Training Random Forest
Predictions are calculated and are saved to the csv file.

Random Forest Training Accuracy: 0.8628571428571429, Validation Accuracy: 0.8160535117056856
```

- **Spam:**

Given code output:

```
SPAM DATASET
```

  - Decision Tree

```
Training Base Decision Tree
SPAM Base Decision Tree Training Accuracy: 0.8984375, Validation Accuracy: 0.8314393939393939
```

```
Training Bagged Decision Tree
SPAM Bagged Decision Tree Training Accuracy: 0.8700284090909091, Validation Accuracy: 0.8333333333333334
```

  - Random Forest

```
Training Random Forest
Predictions are calculated and are saved to the csv file.

SPAM Random Forest Training Accuracy: 0.8399621212121212, Validation Accuracy: 0.8428030303030303
```

- **Kaggle submission:**

  Kaggle username: **Hiva Mohammadzadeh**

  Kaggle Scores:

  a) Spam: Score: 0.84918 = 84.92%

  b) Titanic: Score: 0.78064 = 78.06%

Part 5: Writeup Requirements for the Spam Dataset:

1. Optional.
2. The splits:

```
(" exclamation ") >= 1e-05
(" money ") < 1e-05
(" parenthesis ") < 1e-05
(" prescription ") >= 1e-05
(" pain ") < 1e-05
Therefore this email was spam.
```

```
(" exclamation ") < 1e-05
(" parenthesis ") < 1e-05
(" creative ") < 1e-05
(" meter ") < 1e-05
(" money ") < 1e-05
(" prescription ") < 1e-05
(" volumes ") < 1e-05
(" dollar ") < 1e-05
(" pain ") < 1e-05
(" ampersand ") >= 1e-05
(" height ") < 1e-05
(" business ") < 1e-05
(" message ") < 1e-05
(" ampersand ") < 1.00001
(" revision ") < 1e-05
(" energy ") < 1e-05
(" out ") >= 1e-05
Therefore this email was ham.
```

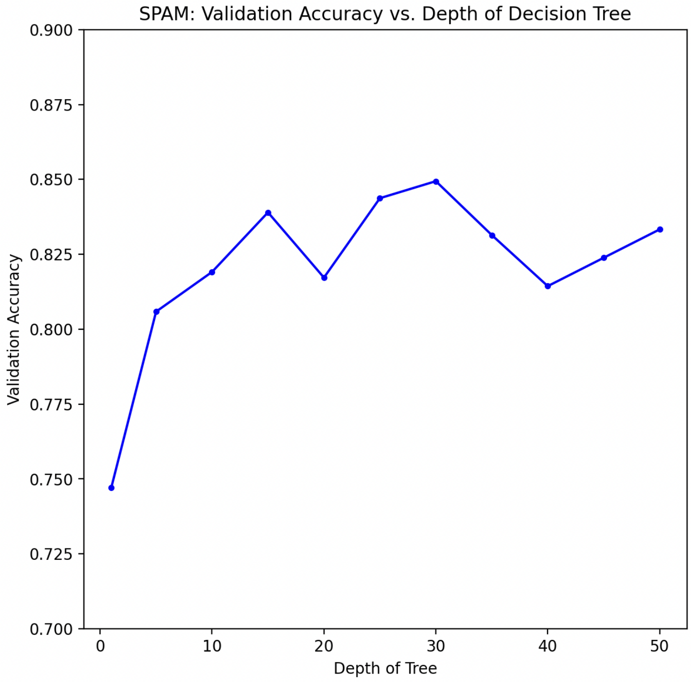
3. Varying maximum depths:

I tired depths from depth = 1 to depth = 50.

```
SPAM DATASET

Depth 1 Validation Accuracy: 0.7471590909090909
Depth 5 Validation Accuracy: 0.8058712121212122
Depth 10 Validation Accuracy: 0.8191287878787878
Depth 15 Validation Accuracy: 0.8390151515151515
Depth 20 Validation Accuracy: 0.8172348484848485
Depth 25 Validation Accuracy: 0.84375
Depth 30 Validation Accuracy: 0.8494318181818182
Depth 35 Validation Accuracy: 0.8314393939393939
Depth 40 Validation Accuracy: 0.8143939393939394
Depth 45 Validation Accuracy: 0.8238636363636364
Depth 50 Validation Accuracy: 0.8333333333333334

Training Decision Tree
SPAM Base Decision Tree Validation Accuracy: 0.8285214348206474
```



From the plot, the validation accuracy grows very fast during depths 1 through around 5 and starts growing slower after that until around 15. Then it drops at 20 then starts growing again to its highest value at depth 30. Then it repeats going down and up again but lower than 0.849 which was at depth 30. Therefore, Depth of 30 had the highest validation accuracy.

Part 6: Writeup Requirements for the Titanic Tree:

Train and visualize a Shallow Decision tree: I trained a depth 4 tree. I just printed the tree since I was not able to get any of the external libraries to do well.

```
Part 6: Visualize a shallow Decision Tree
Tree:
[exclamation < 1e-05: [money < 1e-05: [parenthesis < 1e-05: [prescription < 1e-05: 0.0 (253) | 1.0 (4)] | [energy < 1e-05: 0.0
 (187) | 0.0 (42)]] | [business < 1e-05: [out < 1e-05: 1.0 (4) | 0.0 (5)] | 1.0 (10)]] | [parenthesis < 1e-05: [exclamation <
5.666674444444444: [sharp < 1.5555611111111114: 1.0 (82) | 0.0 (4)] | 1.0 (11)] | [featured < 1e-05: [dollar < 1e-05: 0.0 (65)
 | 1.0 (22)] | 1.0 (11)]]]
```

Tree:

```
[exclamation < 1e-05:
        [money < 1e-05:
                [parenthesis < 1e-05:
                        [prescription < 1e-05: 0.0 (253) | 1.0 (4)]
                        [energy < 1e-05: 0.0 (187) | 0.0 (42)]
                ]
                [business < 1e-05: [out < 1e-05: 1.0 (4) | 0.0 (5)]
                1.0 (10)
                ]
        ]
        [parenthesis < 1e-05:
                [exclamation < 5.666674444444444:
                        [sharp < 1.5555611111111114: 1.0 (82) | 0.0 (4)]
                        1.0 (11)
                        ]
                        [featured < 1e-05:
                                [dollar < 1e-05: 0.0 (65) | 1.0 (22)]
                                1.0 (11)
                        ]
                ]
        ]
]
```

Hyperparameter Tuning

Titanic

Hyperparameter tuning for the Base Decision Tree for the Titanic dataset:

```
Training simplified Decision Tree with tuned depth
Depth: 4
Average Validation Accuracy:  0.7917889447236182
Depth: 5
Average Validation Accuracy:  0.7777738693467338
Depth: 6
Average Validation Accuracy:  0.7748090452261307
Depth: 7
Average Validation Accuracy:  0.7867839195979899
Depth: 8
Average Validation Accuracy:  0.7857336683417085
Depth: 9
Average Validation Accuracy:  0.7907989949748744
Depth: 10
Average Validation Accuracy:  0.7837939698492462
Depth: 11
Average Validation Accuracy:  0.7717738693467336
Depth: 12
Average Validation Accuracy:  0.7687587939698493

Base DT Validation Accuracy: 0.8160535117056856
```

Depth of 4 had the highest average validation accuracy for the base decision trees.

Hyperparameter tuning for the Bagged Decision Tree for the Titanic dataset:

```
Training Bagged Decision Tree
Depth: 9 and Number of Trees: 80
Average Validation Accuracy:  0.7897688442211055
Depth: 10 and Number of Trees: 80
Average Validation Accuracy:  0.8008040201005026
Depth: 11 and Number of Trees: 80
Average Validation Accuracy:  0.7918190954773869
Depth: 12 and Number of Trees: 80
Average Validation Accuracy:  0.7897989949748745
Depth: 13 and Number of Trees: 80
Average Validation Accuracy:  0.7937939698492462
Depth: 14 and Number of Trees: 80
Average Validation Accuracy:  0.7968090452261306
Depth: 15 and Number of Trees: 80
Average Validation Accuracy:  0.7887638190954774
Depth: 9 and Number of Trees: 90
Average Validation Accuracy:  0.7977839195979899
Depth: 10 and Number of Trees: 90
Average Validation Accuracy:  0.7978291457286433
Depth: 11 and Number of Trees: 90
Average Validation Accuracy:  0.7957788944723618
Depth: 12 and Number of Trees: 90
Average Validation Accuracy:  0.8098391959798995
Depth: 13 and Number of Trees: 90
Average Validation Accuracy:  0.7848090452261307
Depth: 14 and Number of Trees: 90
Average Validation Accuracy:  0.7767688442211056
Depth: 15 and Number of Trees: 90
Average Validation Accuracy:  0.7837839195979899
Depth: 9 and Number of Trees: 100
Average Validation Accuracy:  0.7948341708542713
Depth: 10 and Number of Trees: 100
Average Validation Accuracy:  0.7947989949748744
Depth: 11 and Number of Trees: 100
Average Validation Accuracy:  0.8017889447236181
```

```
Depth: 12 and Number of Trees: 100
Average Validation Accuracy:  0.7877788944723618
Depth: 13 and Number of Trees: 100
Average Validation Accuracy:  0.7928090452261307
Depth: 14 and Number of Trees: 100
Average Validation Accuracy:  0.80078391959799
Depth: 15 and Number of Trees: 100
Average Validation Accuracy:  0.7968090452261306
Depth: 9 and Number of Trees: 110
Average Validation Accuracy:  0.7938090452261306
Depth: 10 and Number of Trees: 110
Average Validation Accuracy:  0.787788944723618
Depth: 11 and Number of Trees: 110
Average Validation Accuracy:  0.78678391959799
Depth: 12 and Number of Trees: 110
Average Validation Accuracy:  0.7967688442211055
Depth: 13 and Number of Trees: 110
Average Validation Accuracy:  0.7967738693467338
Depth: 14 and Number of Trees: 110
Average Validation Accuracy:  0.7827487437185929
Depth: 15 and Number of Trees: 110
Average Validation Accuracy:  0.7827889447236182
Depth: 9 and Number of Trees: 120
Average Validation Accuracy:  0.7917537688442211
Depth: 10 and Number of Trees: 120
Average Validation Accuracy:  0.7987889447236182
Depth: 11 and Number of Trees: 120
Average Validation Accuracy:  0.7887939698492462
Depth: 12 and Number of Trees: 120
Average Validation Accuracy:  0.7918040201005025
Depth: 13 and Number of Trees: 120
Average Validation Accuracy:  0.8088442211055277
Depth: 14 and Number of Trees: 120
Average Validation Accuracy:  0.7958040201005024
Depth: 15 and Number of Trees: 120
Average Validation Accuracy:  0.7818140703517586
```

Depth of 12 and 100 trees had the highest average validation accuracy

Hyperparameter tuning for the Random Forest for the Titanic dataset:

```
Training Random Forest
Depth: 6 and Number of Trees: 70
Average validation accuracy:  0.7928341708542714
Depth: 7 and Number of Trees: 70
Average validation accuracy:  0.7957688442211055
Depth: 8 and Number of Trees: 70
Average validation accuracy:  0.7917839195979899
Depth: 9 and Number of Trees: 70
Average validation accuracy:  0.7958140703517588
Depth: 10 and Number of Trees: 70
Average validation accuracy:  0.7937839195979899
Depth: 11 and Number of Trees: 70
Average validation accuracy:  0.7948341708542714
Depth: 6 and Number of Trees: 80
Average validation accuracy:  0.7977989949748744
Depth: 7 and Number of Trees: 80
Average validation accuracy:  0.7957939698492462
Depth: 8 and Number of Trees: 80
Average validation accuracy:  0.8028090452261306
Depth: 9 and Number of Trees: 80
Average validation accuracy:  0.7978140703517587
Depth: 10 and Number of Trees: 80
Average validation accuracy:  0.8027638190954773
Depth: 11 and Number of Trees: 80
Average validation accuracy:  0.7947437185929649
Depth: 6 and Number of Trees: 90
Average validation accuracy:  0.7917738693467338
Depth: 7 and Number of Trees: 90
Average validation accuracy:  0.7927587939698493
Depth: 8 and Number of Trees: 90
Average validation accuracy:  0.8017738693467337
Depth: 9 and Number of Trees: 90
Average validation accuracy:  0.79978391959799
Depth: 10 and Number of Trees: 90
Average validation accuracy:  0.7877638190954773
Depth: 11 and Number of Trees: 90
Average validation accuracy:  0.7987738693467337
Depth: 6 and Number of Trees: 100
Average validation accuracy:  0.8018291457286433
Depth: 7 and Number of Trees: 100
Average validation accuracy:  0.7988391959798996
Depth: 8 and Number of Trees: 100
Average validation accuracy:  0.7998542713567839
Depth: 9 and Number of Trees: 100
Average validation accuracy:  0.7947638190954773
```

```
Depth: 9 and Number of Trees: 100
Average validation accuracy:  0.7947638190954773
Depth: 10 and Number of Trees: 100
Average validation accuracy:  0.7947688442211056
Depth: 11 and Number of Trees: 100
Average validation accuracy:  0.7977738693467337
Depth: 6 and Number of Trees: 110
Average validation accuracy:  0.7948140703517588
Depth: 7 and Number of Trees: 110
Average validation accuracy:  0.7988140703517589
Depth: 8 and Number of Trees: 110
Average validation accuracy:  0.7978140703517589
Depth: 9 and Number of Trees: 110
Average validation accuracy:  0.7898040201005025
Depth: 10 and Number of Trees: 110
Average validation accuracy:  0.7937587939698492
Depth: 11 and Number of Trees: 110
Average validation accuracy:  0.7898140703517587
Depth: 6 and Number of Trees: 120
Average validation accuracy:  0.7918341708542713
Depth: 7 and Number of Trees: 120
Average validation accuracy:  0.7927738693467337
Depth: 8 and Number of Trees: 120
Average validation accuracy:  0.7967939698492462
Depth: 9 and Number of Trees: 120
Average validation accuracy:  0.7957939698492462
Depth: 10 and Number of Trees: 120
Average validation accuracy:  0.795788944723618
Depth: 11 and Number of Trees: 120
Average validation accuracy:  0.793788944723618
Depth: 6 and Number of Trees: 130
Average validation accuracy:  0.7908140703517589
Depth: 7 and Number of Trees: 130
Average validation accuracy:  0.7977386934673367
Depth: 8 and Number of Trees: 130
Average validation accuracy:  0.7918140703517589
Depth: 9 and Number of Trees: 130
Average validation accuracy:  0.8028291457286432
Depth: 10 and Number of Trees: 130
Average validation accuracy:  0.8028140703517588
Depth: 11 and Number of Trees: 130
Average validation accuracy:  0.7907889447236182

Predictions are calculated and are saved to the csv file.

Random Forest Validation Accuracy: 0.7993311036789298
```

Depth of 9 and 130 trees had the highest average validation accuracy.

Spam

Hyperparameter tuning for the Bagged Decision Tree for the Spam dataset:

```
Training Bagged Decision Tree
Depth: 9 and Number of Trees: 80
Average Validation Accuracy:  0.8323863636363636
Depth: 10 and Number of Trees: 80
Average Validation Accuracy:  0.8320075757575758
Depth: 11 and Number of Trees: 80
Average Validation Accuracy:  0.8321969696969695
Depth: 12 and Number of Trees: 80
Average Validation Accuracy:  0.8333333333333334
Depth: 13 and Number of Trees: 80
Average Validation Accuracy:  0.8378787878787879
Depth: 14 and Number of Trees: 80
Average Validation Accuracy:  0.8382575757575758
Depth: 15 and Number of Trees: 80
Average Validation Accuracy:  0.8340909090909092
Depth: 9 and Number of Trees: 90
Average Validation Accuracy:  0.8339015151515152
Depth: 10 and Number of Trees: 90
Average Validation Accuracy:  0.8320075757575758
Depth: 11 and Number of Trees: 90
Average Validation Accuracy:  0.8369318181818182
Depth: 12 and Number of Trees: 90
Average Validation Accuracy:  0.8363636363636363
Depth: 13 and Number of Trees: 90
Average Validation Accuracy:  0.8333333333333334
```

```
Depth: 10 and Number of Trees: 120
Average Validation Accuracy:  0.8327651515151514
Depth: 11 and Number of Trees: 120
Average Validation Accuracy:  0.8344696969696971
Depth: 12 and Number of Trees: 120
Average Validation Accuracy:  0.83125
Depth: 13 and Number of Trees: 120
Average Validation Accuracy:  0.8357954545454545
Depth: 14 and Number of Trees: 120
Average Validation Accuracy:  0.834280303030303
Depth: 15 and Number of Trees: 120
Average Validation Accuracy:  0.8363636363636363

SPAM Bagged Decision Tree Validation Accuracy: 0.8320209973753281
```

Depth of 9 and 130 trees had the highest average validation accuracy.

Hyperparameter tuning for the base Random Forest for the Spam dataset:

```
Training Random Forest
Depth: 6 and Number of Trees: 70
Average validation accuracy:  0.7975378787878789
Depth: 7 and Number of Trees: 70
Average validation accuracy:  0.7918560606060605
Depth: 8 and Number of Trees: 70
Average validation accuracy:  0.7884469696969697
Depth: 9 and Number of Trees: 70
Average validation accuracy:  0.7982954545454545
Depth: 10 and Number of Trees: 70
Average validation accuracy:  0.790151515151515
Depth: 11 and Number of Trees: 70
Average validation accuracy:  0.7935606060606061
Depth: 6 and Number of Trees: 80
Average validation accuracy:  0.7960227272727272
Depth: 7 and Number of Trees: 80
Average validation accuracy:  0.7969696969696969
Depth: 8 and Number of Trees: 80
Average validation accuracy:  0.7869318181818181
Depth: 9 and Number of Trees: 80
Average validation accuracy:  0.7965909090909091
Depth: 10 and Number of Trees: 80
Average validation accuracy:  0.7960227272727273
Depth: 11 and Number of Trees: 80
Average validation accuracy:  0.793939393939394
Depth: 6 and Number of Trees: 90
Average validation accuracy:  0.7892045454545455
Depth: 7 and Number of Trees: 90
Average validation accuracy:  0.7912878787878788
Depth: 8 and Number of Trees: 90
Average validation accuracy:  0.790719696969697
Depth: 9 and Number of Trees: 90
Average validation accuracy:  0.7931818181818182
Depth: 10 and Number of Trees: 90
Average validation accuracy:  0.7977272727272728
Depth: 11 and Number of Trees: 90
Average validation accuracy:  0.7956439393939394
Depth: 6 and Number of Trees: 100
Average validation accuracy:  0.7912878787878788
Depth: 7 and Number of Trees: 100
Average validation accuracy:  0.7956439393939394
Depth: 8 and Number of Trees: 100
Average validation accuracy:  0.7926136363636364
Depth: 9 and Number of Trees: 100
Average validation accuracy:  0.7994318181818182
```

```
Depth: 10 and Number of Trees: 100
Average validation accuracy:  0.7941287878787879
Depth: 11 and Number of Trees: 100
Average validation accuracy:  0.7952651515151515
Depth: 6 and Number of Trees: 110
Average validation accuracy:  0.7895833333333333
Depth: 7 and Number of Trees: 110
Average validation accuracy:  0.7920454545454545
Depth: 8 and Number of Trees: 110
Average validation accuracy:  0.7965909090909091
Depth: 9 and Number of Trees: 110
Average validation accuracy:  0.7964015151515151
Depth: 10 and Number of Trees: 110
Average validation accuracy:  0.7965909090909091
Depth: 11 and Number of Trees: 110
Average validation accuracy:  0.7979166666666667
Depth: 6 and Number of Trees: 120
Average validation accuracy:  0.790151515151515
Depth: 7 and Number of Trees: 120
Average validation accuracy:  0.7984848484848485
Depth: 8 and Number of Trees: 120
Average validation accuracy:  0.7946969696969697
Depth: 9 and Number of Trees: 120
Average validation accuracy:  0.7960227272727273
Depth: 10 and Number of Trees: 120
Average validation accuracy:  0.7952651515151514
Depth: 11 and Number of Trees: 120
Average validation accuracy:  0.7929924242424243
Depth: 6 and Number of Trees: 130
Average validation accuracy:  0.7960227272727273
Depth: 7 and Number of Trees: 130
Average validation accuracy:  0.7928030303030303
Depth: 8 and Number of Trees: 130
Average validation accuracy:  0.8001893939393939
Depth: 9 and Number of Trees: 130
Average validation accuracy:  0.7939393939393938
Depth: 10 and Number of Trees: 130
Average validation accuracy:  0.7962121212121211
Depth: 11 and Number of Trees: 130
Average validation accuracy:  0.7950757575757577
```

Depth of 9 and 130 trees had the highest average validation accuracy.

CODE APPENDIX:

Question 4: Decision Trees for Classification

```
# You may want to install "gprof2dot"
import io
from collections import Counter
import numpy as np
import scipy.io
import sklearn.model_selection
import sklearn.tree
from numpy import genfromtxt
from scipy import stats
from sklearn.base import BaseEstimator, ClassifierMixin
import pydot
eps = 1e-5  # a small number
import math
from save_csv import results_to_csv
import matplotlib.pyplot as plt
```

Part 1: Implement Decision Trees:

```
#### QUESTION 4.1 Implement Decision Trees:
class DecisionTree:
   def __init__(self, max_depth=3, feature_labels=None, m=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
       # variable m to know the number of samples in a subset
       self.m = m
   # Helper function for the information gain function to calculate the entropy of y,
   # left and right in order to better calculate the gain from new entropy and previous
entropy.
   # entropy(labels):A method that takes in the labels of data stored at a node and compute
the entropy for the distribution of the labels.
   @staticmethod
   def entropy(y):
       probs = []
       for label in np.unique(y):
           count = len(y[np.where(y==label)])
           probs.append(float(count / len(y)))
       entropy = -1 * sum([prob * np.log2(prob) for prob in probs])
       return entropy

   @staticmethod
   def information_gain(X, y, thresh):
       # TODO: implement information gain function
       # return np.random.rand()
       entropy = DecisionTree.entropy(y)
       left = DecisionTree.entropy(y[np.where(X < thresh)])
       right = DecisionTree.entropy(y[np.where(X >= thresh)])
```

```
        new_entropy = (len(y[np.where(X < thresh)]) * left + len(y[np.where(X >= thresh)]) *
right) / len(y)
        return entropy - new_entropy

    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 = []
            # The following logic prevents thresholding on exactly the minimum
            # or maximum values, which may not lead to any meaningful node
            # splits.
            # added the functionality of including m in the fit's implementation of the
decision tree since I added it in init.
            data = X
            if self.m:
                attribute_bag = np.random.choice(list(range(len(self.features))), size=self.m,
replace=False)
                X = data[:, attribute_bag]
            else:
                attribute_bag = None
                X = data
            thresh = np.array([
                np.linspace(np.min(X[:, i]) + eps, np.max(X[:, i]) - eps, num=10)
                for 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]
            # added the functionality of including m in the fit's implementation of the
decision tree since I added it in init.
            if self.m:
                self.split_idx = attribute_bag[self.split_idx]
            X0, y0, X1, y1 = self.split(data, 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, m=self.m)
            self.left.fit(X0, y0)
            self.right = DecisionTree(
                max_depth=self.max_depth - 1, feature_labels=self.features, m=self.m)
            self.right.fit(X1, y1)
        else:
            self.max_depth = 0
            self.data, self.labels = data, y
            self.pred = stats.mode(y, keepdims= True).mode[0]
    else:
        self.data, self.labels = X, y
        self.pred = stats.mode(y, keepdims= True).mode[0]
    return self

def predict(self, X, verbose=False):
    if self.max_depth == 0:
        return self.pred * np.ones(X.shape[0])
    else:
        ##Question 4.5 part 2: The Splits
        if (verbose and X.shape[0] != 0):
            if X[0, self.split_idx] < self.thresh:
                print('("', self.features[self.split_idx], '")', "<", self.thresh)
            else:
                print('("', self.features[self.split_idx], '")',">=", self.thresh)
        X0, idx0, X1, idx1 = self.split_test(X, idx=self.split_idx, thresh=self.thresh)
        yhat = np.zeros(X.shape[0])
        yhat[idx0] = self.left.predict(X0, verbose=verbose)
        yhat[idx1] = self.right.predict(X1, verbose=verbose)
        return yhat

def __repr__(self):
    if self.max_depth == 0:
        return "%s (%s)" % (self.pred, self.labels.size)
    else:
        return "[%s < %s: %s | %s]" % (self.features[self.split_idx],
                                       self.thresh, self.left.__repr__(),
                                       self.right.__repr__())

#### Bagged Decision Trees
class BaggedTrees:
  def __init__(self, maxdepth=3, n=25, features=None, sample_size=None):
      ### Params was confusing me. So I added parameters myself.
      # if params is None:
      #     params = {}
      # self.params = params
      self.n = n
      self.sample_size = sample_size
      self.decision_trees = [
```

```
        ### Params was confusing me. So I added parameters myself. I also used my own Decision
tree and not the sklearn's one
            # sklearn.tree.DecisionTreeClassifier(random_state=i, **self.params)
            DecisionTree(max_depth=maxdepth, feature_labels=features)
            for i in range(self.n)
        ]

    def fit(self, X, y):
        # TODO: implement function
        # pass
        all_data = np.concatenate((X,y.reshape(-1,1)), axis=1)
        for dt in self.decision_trees:
            samples = np.random.choice(list(range(len(all_data))), size=self.sample_size,
replace=True)
            train = all_data[samples, :]
            train_data = train[:, :-1]
            train_label = train[:, -1:]
            dt.fit(train_data, train_label)

    def predict(self, X):
        # TODO: implement function
        # pass
        predictions = []
        for dt in self.decision_trees:
            predictions.append(dt.predict(X))
        all_predictions = np.vstack(predictions)
        mode_predictions = stats.mode(all_predictions, keepdims = True).mode[0]
        return mode_predictions
```

Part 2: Implement a Random Forest:

```
#### QUESTION 4.2 Implement a Random Forest
class RandomForest(BaggedTrees):
    def __init__(self, maxdepth=7, n=25, features=None, sample_size=None, m=1):
        # TODO: implement function
        # pass
        ### Params was confusing me. So I added parameters myself.
        # if params is None:
        #     params = {}
        self.n = n
        self.sample_size = sample_size
        self.decision_trees = [
            DecisionTree(max_depth=maxdepth, feature_labels=features, m=m)
            for i in range(self.n)
        ]
```

Some extra functions to preprocess and do cross validation of hyperparameters and evaluate the models:

```
### Given code but didn't use
# 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

# Given Function for preprocessing. Didn't change anything.
def preprocess(data, fill_mode=True, min_freq=10, onehot_cols=[]):
   # fill_mode = False
   # Temporarily assign -1 to missing data
   data[data == ''] = '-1'
   # Hash the columns (used for handling strings)
   onehot_encoding = []
   onehot_features = []
   for col in onehot_cols:
       counter = Counter(data[:, col])
       for term in counter.most_common():
           if term[0] == '-1':
               continue
           if term[-1] <= min_freq:
               break
           onehot_features.append(term[0])
           onehot_encoding.append((data[:, col] == term[0]).astype(float))
       data[:, col] = '0'
   onehot_encoding = np.array(onehot_encoding).T
   data = np.hstack([np.array(data, dtype=float), np.array(onehot_encoding)])
   # Replace missing data with the mode value. We use the mode instead of
   # the mean or median because this makes more sense for categorical
   # features such as gender or cabin type, which are not ordered.
   if fill_mode:
       for i in range(data.shape[-1]):
           mode = stats.mode(data[((data[:, i] < -1 - eps) +
                                   (data[:, i] > -1 + eps))][:, i], keepdims = True).mode[0]
           data[(data[:, i] > -1 - eps) * (data[:, i] < -1 + eps)][:, i] = mode
   return data, onehot_features

# Given Function for evaluating Clf. Didn't change anything.
def evaluate(clf):
   print("Cross validation", sklearn.model_selection.cross_val_score(clf, X, y))
   if hasattr(clf, "decision_trees"):
       counter = Counter([t.tree_.feature[0] for t in clf.decision_trees])
       first_splits = [(features[term[0]], term[1]) for term in counter.most_common()]
```

```
        print("First splits", first_splits)

## Defined helper function to find the best depth for the base decision tree that gives the
highest average validation accuracy.
def validate_base_decision_tree(X, y, features):
    # Tried tuning depth by using depths of 4 through 12.
    for depth in [4,5,6,7,8,9,10,11,12]:
        all_data = np.concatenate((X,y.reshape(-1, 1)), axis=1)
        np.random.shuffle(all_data)
        kfold = np.array_split(all_data, 5, axis=0)
        print("Depth: {}".format(depth))
        accuracies = []
        for i in range(len(kfold)):
            validation = kfold[i]
            train = np.concatenate(kfold[:i] + kfold[i+1:], axis=0)
            train_data, train_label = train[:, :-1], train[:, -1:]
            validation_data, validation_label = validation[:, :-1], validation[:, -1:]
            decision_tree = DecisionTree(max_depth=depth, feature_labels=features)
            decision_tree.fit(train_data, train_label)
            accuracies.append(np.sum(decision_tree.predict(validation_data) ==
validation_label.reshape(-1,)) / len(validation_label.reshape(-1,)))
        accuracies = np.array(accuracies)
        print("Average Validation Accuracy: ", np.mean(accuracies))
    print()

### Defined helper function to find the best depth and the number of trees for the bagged
decision tree
# that gives the highest average validation accuracy.
def validate_bagged_decision_tree(X, y, features, sample_size=500):
    # Tried tuning number of trees by using num_trees of 80 through 120.
    for num_trees in [80, 90, 100, 110, 120]:
        # Tried tuning depth by using depths of 4 through 12.
        for depth in [9,10,11,12,13,14,15]:
            all_data = np.concatenate((X,y.reshape(-1, 1)), axis=1)
            np.random.shuffle(all_data)
            kfold = np.array_split(all_data, 5, axis=0)
            print("Depth: {} and Number of Trees: {}".format(depth,num_trees))
            accuracies = []
            for i in range(len(kfold)):
                validation = kfold[i]
                train = np.concatenate(kfold[:i] + kfold[i+1:], axis=0)
                train_data, train_label = train[:, :-1], train[:, -1:]
                validation_data, validation_label = validation[:, :-1], validation[:, -1:]
                decision_tree = BaggedTrees(maxdepth=depth, n=25, features=features,
sample_size=sample_size)
                decision_tree.fit(train_data, train_label)
                accuracies.append(np.sum(decision_tree.predict(validation_data) ==
validation_label.reshape(-1,)) / len(validation_label.reshape(-1,)))
            accuracies = np.array(accuracies)
```

```
            print("Average Validation Accuracy: ", np.mean(accuracies))
    print()

## Defined helper function to find the best depth and the number of trees for random forest
# that gives the highest average validation accuracy.
def validate_random_forest(X, y, features, m, sample_size=500):
    # Tried tuning number of trees by using num_trees of 80 through 120.
    for num_trees in [70, 80, 90, 100, 110, 120, 130]:
        # Tried tuning depth by using depths of 6 through 11.
        for depth in [6,7,8,9,10,11]:
            all_data = np.concatenate((X,y.reshape(-1, 1)), axis=1)
            np.random.shuffle(all_data)
            kfold = np.array_split(all_data, 5, axis=0)
            print("Depth: {} and Number of Trees: {}".format(depth,num_trees))
            accuracies = []
            for i in range(len(kfold)):
                validation = kfold[i]
                train = np.concatenate(kfold[:i] + kfold[i+1:], axis=0)
                train_data, train_label = train[:, :-1], train[:, -1:]
                validation_data, validation_label = validation[:, :-1], validation[:, -1:]
                random_forest = RandomForest(maxdepth=5, n=num_trees, features=features, m=m,
sample_size=sample_size)
                random_forest.fit(train_data, train_label)
                accuracies.append(np.sum(random_forest.predict(validation_data) ==
validation_label.reshape(-1,)) / len(validation_label.reshape(-1,)))
            accuracies = np.array(accuracies)
            print("Average validation accuracy: ", np.mean(accuracies))
    print()
```

Part 4: Performance Evaluation:

```
#### Defined helper function for performance Evaluation calculation of the model and
# also save the predictions to a file.
def eval(X, y, split, model, filename=None, Z=None):
    #shuffle the data
    all_data = np.concatenate((X,y.reshape(-1,1)), axis=1)
    np.random.shuffle(all_data)
    train = all_data[:split, :]
    validation = all_data[split:, :]
    train_data, train_label = train[:, :-1], train[:, -1:].reshape(-1,)
    validation_data, validation_label = validation[:, :-1], validation[:, -1:].reshape(-1)
    model.fit(train_data, train_label)
    if filename:
        results_to_csv(model.predict(Z), filename)
        print("Predictions are calculated and are saved to the csv file. \n")
    return (np.sum(model.predict(train_data) == train_label) / len(train_label)),
(np.sum(model.predict(validation_data) == validation_label) / len(validation_label))
```

- **Spam:**

```
##SPAM DATASET
print("\nSPAM DATASET")
np.random.seed(200)
dataset = "spam"
if dataset == "spam":
    features = [
        "pain", "private", "bank", "money", "drug", "spam", "prescription", "creative",
        "height", "featured", "differ", "width", "other", "energy", "business", "message",
        "volumes", "revision", "path", "meter", "memo", "planning", "pleased", "record",
"out",
        "semicolon", "dollar", "sharp", "exclamation", "parenthesis", "square_bracket",
        "ampersand"
    ]
    assert len(features) == 32
    # Load spam data
    path_train = 'dataset/spam/spam_data.mat'
    data = scipy.io.loadmat(path_train)
    X = data['training_data']
    # print(X.shape)
    y = np.squeeze(data['training_labels'])
    Z = data['test_data']
    class_names = ["Ham", "Spam"]


#### QUESTION 4.4: Performance Evaluation:
### Training Base Decision Tree
print("\nTraining Base Decision Tree")
# Depth of 30 had the highest validation accuracy
base_decision_tree = DecisionTree(max_depth=30, feature_labels=features)
# 80/20 split -> 5280(0.20) = 1056 , 5280(0.80) = 4224
training_accuracy, validation_accuracy = eval(X, y, 4224, base_decision_tree)
print("SPAM Base Decision Tree Training Accuracy: {}, Validation Accuracy:
{}".format(training_accuracy, validation_accuracy))

##### QUESTION 4.4: Performance Evaluation:
### Training Bagged Decision Tree
print("\nTraining Bagged Decision Tree")
# To use validation to find the best depth and number of trees, uncomment and run the
following line of code:
# validate_bagged_decision_tree(X, y, features, sample_size=3000)
# Depth 14 and N=80 trees works best
bagged_decision_tree = BaggedTrees(maxdepth=14, n=80, features=features, sample_size=3000)
# 80/20 split -> 5280(0.20) = 1056 , 5280(0.80) = 4224
training_accuracy, validation_accuracy = eval(X, y, 4224, bagged_decision_tree)
print("SPAM Bagged Decision Tree Training Accuracy: {}, Validation Accuracy:
{}".format(training_accuracy, validation_accuracy))
```

- Decision Tree

```
#### QUESTION 4.4: Performance Evaluation:
### Training Base Decision Tree
print("\nTraining Base Decision Tree")
# Depth of 30 had the highest validation accuracy
base_decision_tree = DecisionTree(max_depth=30, feature_labels=features)
# 80/20 split -> 5280(0.20) = 1056 , 5280(0.80) = 4224
training_accuracy, validation_accuracy = eval(X, y, 4224, base_decision_tree)
print("SPAM Base Decision Tree Training Accuracy: {}, Validation Accuracy:
{}".format(training_accuracy, validation_accuracy))


##### QUESTION 4.4: Performance Evaluation:
### Training Bagged Decision Tree
print("\nTraining Bagged Decision Tree")
```

```
    # To use validation to find the best depth and number of trees, uncomment and run the
following line of code:
    # validate_bagged_decision_tree(X, y, features, sample_size=3000)
    # Depth 14 and N=80 trees works best
    bagged_decision_tree = BaggedTrees(maxdepth=14, n=80, features=features, sample_size=3000)
    # 80/20 split -> 5280(0.20) = 1056 , 5280(0.80) = 4224
    training_accuracy, validation_accuracy = eval(X, y, 4224, bagged_decision_tree)
    print("SPAM Bagged Decision Tree Training Accuracy: {}, Validation Accuracy:
{}".format(training_accuracy, validation_accuracy))
```

- Random Forest

```
#### QUESTION 4.4: Performance Evaluation:
    ### Training Random Forest
    print("\nTraining Random Forest")
    # To use validation to find the best depth and number of trees, uncomment and run the
following line of code:
    # validate_random_forest(X, y, features, math.ceil(math.sqrt(len(features))))
    # Depth 9 and N=100 trees works best
    random_forest = RandomForest(maxdepth=9, n=100, features=features, sample_size=3000,
m=math.ceil(math.sqrt(len(features))))
    # 80/20 split -> 5280(0.20) = 1056 , 5280(0.80) = 4224
    training_accuracy, validation_accuracy = eval(X, y, 4224, random_forest,
filename="spam.csv", Z=Z)
    print("SPAM Random Forest Training Accuracy: {}, Validation Accuracy:
{}".format(training_accuracy, validation_accuracy))
```

- **Titanic**

```
#### QUESTION 4.4 Performance Evaluation:
# For each of the 2 datasets, train both a decision tree and random forest and report your
training and validation
# accuracies. You should be reporting 8 numbers (2 datasets × 2 classifiers ×
training/validation).
if __name__ == "__main__":
      ##TITANIC DATASET
    dataset = "titanic"
    ### Params was confusing me. So I added parameters myself.
    # params = {
    #     "max_depth": 5,
    #     # "random_state": 6,
    #     "min_samples_leaf": 10,
    # }
    # N = 100
    print("\nTITANIC DATASET")
    ## Set the random seed to 150
    np.random.seed(150)

    if dataset == "titanic":
        # Load titanic data
        path_train = './dataset/titanic/titanic_training.csv'
        data = genfromtxt(path_train, delimiter=',', dtype=None, encoding=None)
        path_test = './dataset/titanic/titanic_test_data.csv'
        test_data = genfromtxt(path_test, delimiter=',', dtype=None, encoding=None)
        y = data[1:, -1]  # label = survived
        class_names = ["Died", "Survived"]
        labeled_idx = np.where(y != '')[0]
        y = np.array(y[labeled_idx])
        y = y.astype(float).astype(int)
```

```
        print("\nPart (b): preprocessing the titanic dataset")
        X, onehot_features = preprocess(data[1:, :-1], onehot_cols=[1, 5, 7, 8])
        X = X[labeled_idx, :]
        Z, _ = preprocess(test_data[1:, :], onehot_cols=[1, 5, 7, 8])
        assert X.shape[1] == Z.shape[1]
        features = list(data[0, :-1]) + onehot_features
      # print(math.ceil(math.sqrt(len(features))))
   print("Titanic Features:", features)
   print("Train/test size:", X.shape, Z.shape)
    print("\n\nPart 0: constant classifier")
   print("Accuracy", 1 - np.sum(y) / y.size)
```

- Decision Tree

```
#### QUESTION 4.4: Performance Evaluation:
   ### Training simplified Decision Tree with tuned depth
   print("\nTraining Base Decision Tree with tuned depth ")
   # To use validation to find the best depth, uncomment and run the following line of code:
   # validate_base_decision_tree(X, y, features)
   # Depth 4 had the highest average validation accuracy
   base_decision_tree  = DecisionTree(max_depth=4, feature_labels=features)
   training_accuracy, validation_accuracy = eval(X, y, 700, base_decision_tree)
   print("Base Decision Tree Training Accuracy: {}, Validation Accuracy: {}".format(
training_accuracy, validation_accuracy))

   #### QUESTION 4.4: Performance Evaluation:
   ### Training Bagged Decision Tree
   print("\nTraining Bagged Decision Tree")
   # To use validation to find the best depth and number of trees, uncomment and run the
following line of code:
   # validate_bagged_decision_tree(X, y, features)
   # Depth 12 and 100 trees had the highest average validation accuracy
   bagged_decision_tree = BaggedTrees(maxdepth=12, n=100, features=features, sample_size=700)
   training_accuracy, validation_accuracy = eval(X, y, 700, bagged_decision_tree)
   print("Bagged Decision Tree Training Accuracy: {}, Validation Accuracy: {}".format(
training_accuracy, validation_accuracy))
```

- Random Forest

```
   #### QUESTION 4.4: Performance Evaluation:
   ### Training Random Forest
   print("\nTraining Random Forest")
   # To use validation to find the best depth and number of trees, uncomment and run the
following line of code:
   # validate_random_forest(X, y, features, math.ceil(math.sqrt(len(features))))
   # Depth 9 and N=130 trees works best
   random_forest = RandomForest(maxdepth=9, n=130, features=features, sample_size=700,
m=math.ceil(math.sqrt(len(features))))
   training_accuracy, validation_accuracy = eval(X, y, 700, random_forest,
filename="titanic.csv", Z=Z)
   print("Random Forest Training Accuracy: {}, Validation Accuracy: {}".format(
training_accuracy, validation_accuracy))

   ### Given code but didn't use
   # # Basic decision tree
   # print("\n\nPart (a-b): simplified decision tree")
   # dt = DecisionTree(max_depth=3, feature_labels=features)
   # basic_val_acc = eval(X, y, 700, dt)
   # print(dt)
   # print("Predictions", dt.predict(Z)[:100])

   # print("\n\nPart (c): sklearn's decision tree")
   # clf = sklearn.tree.DecisionTreeClassifier(random_state=0, max_depth=3)
```

```
    # clf.fit(X, y)
    # evaluate(clf)
    # out = io.StringIO()

    # # You may want to install "gprof2dot"
    # sklearn.tree.export_graphviz(
    #     clf, out_file=out, feature_names=features, class_names=class_names)
    # graph = pydot.graph_from_dot_data(out.getvalue())
    # pydot.graph_from_dot_data(out.getvalue())[0].write_pdf("%s-tree.pdf" % dataset)
```

Part 5: Writeup Requirements for the Spam Dataset:

1. Optional
2. The splits:

```
### QUESTION 4.5 part 2: The Splits: For your decision tree, and for a data point of your
choosing from each class (spam and ham),
    # state the splits (i.e., which feature and which value of that feature to split on) your
decision tree made to classify it
    spam_sample = X[y==1, :][0,:].reshape(1, 32)
    ham_sample = X[y==0, :][4, :].reshape(1, 32)
    #### Predictions for spam sample
    print("\n")
    base_decision_tree.predict(spam_sample, verbose=True)
    print("Therefore this email was spam.\n")
    #### Predictions for ham sample
    base_decision_tree.predict(ham_sample, verbose=True)
    print("Therefore this email was ham.\n")
```

3. Varying maximum depths:

```
### QUESTION 4.5 question 3: Varying maximum depths:
    # I tired depths from depth = 1 to depth = 50.
    #### Visualizing accuracies vs. depth.
    accuracies = []
    depths = [1,5,10,15,20,25,30,35,40,45,50]
    for depth in depths:
        base_dt = DecisionTree(max_depth=depth, feature_labels=features)
        #80/20 split -> 5280(0.20) = 1056 , 5280(0.80) = 4224
        training_accuracy, validation_accuracy = eval(X, y, 4224, base_dt)
        accuracies.append(validation_accuracy)
        print("Depth {} Validation Accuracy: {}".format(depth, validation_accuracy))
    fig, axes = plt.subplots(1, 1, figsize=(7, 7))
    axes.plot(depths, accuracies)
    axes.set_title("SPAM: Validation Accuracy vs. Depth of Decision Tree")
    axes.set_xlabel("Depth of Tree")
    axes.set_ylabel("Validation Accuracy")
    plt.show()
```

Part 6: Writeup Requirements for the Titanic Dataset:

```
#### QUESTION 4.6 Writeup Requirements for the Titanic Dataset:
# Train and visualize a Shallow Decision tree: I trained a depth 4 tree.
```

```
# I just printed the tree since I was not able to get any of the external libraries to do
well.
# Basic decision tree
   print("\n\nPart (a-b): simplified decision tree")
   dt = DecisionTree(max_depth=4, feature_labels=features)
   basic_val_acc = eval(X, y, 700, dt)
   print("Tree:")
   print(dt)
```

All of code appendix: In case I missed something above

```
# You may want to install "gprof2dot"
import io
from collections import Counter
import numpy as np
import scipy.io
import sklearn.model_selection
import sklearn.tree
from numpy import genfromtxt
from scipy import stats
from sklearn.base import BaseEstimator, ClassifierMixin
import pydot
eps = 1e-5  # a small number
import math
from save_csv import results_to_csv
import matplotlib.pyplot as plt

#### QUESTION 4.1 Implement Decision Trees:
class DecisionTree:
   def __init__(self, max_depth=3, feature_labels=None, m=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
       # variable m to know the number of samples in a subset
       self.m = m

   # Helper function for the information gain function to calculate the entropy of y,
   # left and right in order to better calculate the gain from new entropy and previous entropy.
   # entropy(labels):A method that takes in the labels of data stored at a node and compute the entropy
for the distribution of the labels.
   @staticmethod
   def entropy(y):
       probs = []
       for label in np.unique(y):
           count = len(y[np.where(y==label)])
           probs.append(float(count / len(y)))
       entropy = -1 * sum([prob * np.log2(prob) for prob in probs])
       return entropy

   @staticmethod
   def information_gain(X, y, thresh):
       # TODO: implement information gain function
       # return np.random.rand()
       entropy = DecisionTree.entropy(y)
       left = DecisionTree.entropy(y[np.where(X < thresh)])
       right = DecisionTree.entropy(y[np.where(X >= thresh)])
       new_entropy = (len(y[np.where(X < thresh)]) * left + len(y[np.where(X >= thresh)]) * right) /
len(y)
       return entropy - new_entropy


   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 = []
            # The following logic prevents thresholding on exactly the minimum
            # or maximum values, which may not lead to any meaningful node
            # splits.

            # added the functionality of including m in the fit's implementation of the decision tree
since I added it in init.
            data = X
            if self.m:
                attribute_bag = np.random.choice(list(range(len(self.features))), size=self.m,
replace=False)
                X = data[:, attribute_bag]
            else:
                attribute_bag = None
                X = data
            thresh = np.array([
                np.linspace(np.min(X[:, i]) + eps, np.max(X[:, i]) - eps, num=10)
                for 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]
            # added the functionality of including m in the fit's implementation of the decision tree
since I added it in init.
            if self.m:
                self.split_idx = attribute_bag[self.split_idx]
            X0, y0, X1, y1 = self.split(data, 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, m=self.m)
                self.left.fit(X0, y0)
                self.right = DecisionTree(
                    max_depth=self.max_depth - 1, feature_labels=self.features, m=self.m)
                self.right.fit(X1, y1)
            else:
                self.max_depth = 0
                self.data, self.labels = data, y
                self.pred = stats.mode(y, keepdims= True).mode[0]
        else:
            self.data, self.labels = X, y
            self.pred = stats.mode(y, keepdims= True).mode[0]
        return self

    def predict(self, X, verbose=False):
        if self.max_depth == 0:
            return self.pred * np.ones(X.shape[0])
        else:
            ##Question 4.5 part 2: The Splits
            if (verbose and X.shape[0] != 0):
                if X[0, self.split_idx] < self.thresh:
                    print('("', self.features[self.split_idx], '")', "<", self.thresh)
                else:
                    print('("', self.features[self.split_idx], '")',">=", self.thresh)

            X0, idx0, X1, idx1 = self.split_test(X, idx=self.split_idx, thresh=self.thresh)
            yhat = np.zeros(X.shape[0])
            yhat[idx0] = self.left.predict(X0, verbose=verbose)
            yhat[idx1] = self.right.predict(X1, verbose=verbose)
            return yhat

    def __repr__(self):
        if self.max_depth == 0:
            return "%s (%s)" % (self.pred, self.labels.size)
        else:
            return "[%s < %s: %s | %s]" % (self.features[self.split_idx],
                                           self.thresh, self.left.__repr__(),
                                           self.right.__repr__())

#### Bagged Decision Trees
class BaggedTrees:
    def __init__(self, maxdepth=3, n=25, features=None, sample_size=None):
        ### Params was confusing me. So I added parameters myself.
        # if params is None:
        #     params = {}
```

```
        # self.params = params
        self.n = n
        self.sample_size = sample_size
        self.decision_trees = [
        ### Params was confusing me. So I added parameters myself. I also used my own Decision tree and
not the sklearn's one
            # sklearn.tree.DecisionTreeClassifier(random_state=i, **self.params)
            DecisionTree(max_depth=maxdepth, feature_labels=features)
            for i in range(self.n)
        ]

    def fit(self, X, y):
        # TODO: implement function
        # pass
        all_data = np.concatenate((X,y.reshape(-1,1)), axis=1)
        for dt in self.decision_trees:
            samples = np.random.choice(list(range(len(all_data))), size=self.sample_size, replace=True)
            train = all_data[samples, :]
            train_data = train[:, :-1]
            train_label = train[:, -1:]
            dt.fit(train_data, train_label)


    def predict(self, X):
        # TODO: implement function
        # pass
        predictions = []
        for dt in self.decision_trees:
            predictions.append(dt.predict(X))
        all_predictions = np.vstack(predictions)
        mode_predictions = stats.mode(all_predictions, keepdims = True).mode[0]
        return mode_predictions

#### QUESTION 4.2 Implement a Random Forest
class RandomForest(BaggedTrees):
    def __init__(self, maxdepth=7, n=25, features=None, sample_size=None, m=1):
        # TODO: implement function
        # pass
        ### Params was confusing me. So I added parameters myself.
        # if params is None:
        #     params = {}
        self.n = n
        self.sample_size = sample_size
        self.decision_trees = [
            DecisionTree(max_depth=maxdepth, feature_labels=features, m=m)
            for i in range(self.n)
        ]

### Given code but didn't use
# 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


# Given Function for preprocessing. Didn't change anything.
def preprocess(data, fill_mode=True, min_freq=10, onehot_cols=[]):
    # fill_mode = False
    # Temporarily assign -1 to missing data
    data[data == ''] = '-1'
    # Hash the columns (used for handling strings)
    onehot_encoding = []
    onehot_features = []
    for col in onehot_cols:
        counter = Counter(data[:, col])
        for term in counter.most_common():
            if term[0] == '-1':
                continue
            if term[-1] <= min_freq:
                break
            onehot_features.append(term[0])
            onehot_encoding.append((data[:, col] == term[0]).astype(float))
        data[:, col] = '0'
    onehot_encoding = np.array(onehot_encoding).T
    data = np.hstack([np.array(data, dtype=float), np.array(onehot_encoding)])
    # Replace missing data with the mode value. We use the mode instead of
    # the mean or median because this makes more sense for categorical
```

```python
    # features such as gender or cabin type, which are not ordered.
    if fill_mode:
        for i in range(data.shape[-1]):
            mode = stats.mode(data[((data[:, i] < -1 - eps) +
                                    (data[:, i] > -1 + eps))][:, i], keepdims = True).mode[0]
            data[(data[:, i] > -1 - eps) * (data[:, i] < -1 + eps)][:, i] = mode
    return data, onehot_features

# Given Function for evaluating Clf. Didn't change anything.
def evaluate(clf):
    print("Cross validation", sklearn.model_selection.cross_val_score(clf, X, y))
    if hasattr(clf, "decision_trees"):
        counter = Counter([t.tree_.feature[0] for t in clf.decision_trees])
        first_splits = [(features[term[0]], term[1]) for term in counter.most_common()]
        print("First splits", first_splits)

## Defined helper function to find the best depth for the base decision tree that gives the highest
average validation accuracy.
def validate_base_decision_tree(X, y, features):
    # Tried tuning depth by using depths of 4 through 12.
    for depth in [4,5,6,7,8,9,10,11,12]:
        all_data = np.concatenate((X,y.reshape(-1, 1)), axis=1)
        np.random.shuffle(all_data)
        kfold = np.array_split(all_data, 5, axis=0)
        print("Depth: {}".format(depth))
        accuracies = []
        for i in range(len(kfold)):
            validation = kfold[i]
            train = np.concatenate(kfold[:i] + kfold[i+1:], axis=0)
            train_data, train_label = train[:, :-1], train[:, -1:]
            validation_data, validation_label = validation[:, :-1], validation[:, -1:]
            decision_tree = DecisionTree(max_depth=depth, feature_labels=features)
            decision_tree.fit(train_data, train_label)
            accuracies.append(np.sum(decision_tree.predict(validation_data) ==
validation_label.reshape(-1,)) / len(validation_label.reshape(-1,)))
        accuracies = np.array(accuracies)
        print("Average Validation Accuracy: ", np.mean(accuracies))
    print()

### Defined helper function to find the best depth and the number of trees for the bagged decision tree
# that gives the highest average validation accuracy.
def validate_bagged_decision_tree(X, y, features, sample_size=500):
    # Tried tuning number of trees by using num_trees of 80 through 120.
    for num_trees in [80, 90, 100, 110, 120]:
        # Tried tuning depth by using depths of 4 through 12.
        for depth in [9,10,11,12,13,14,15]:
            all_data = np.concatenate((X,y.reshape(-1, 1)), axis=1)
            np.random.shuffle(all_data)
            kfold = np.array_split(all_data, 5, axis=0)
            print("Depth: {} and Number of Trees: {}".format(depth,num_trees))
            accuracies = []
            for i in range(len(kfold)):
                validation = kfold[i]
                train = np.concatenate(kfold[:i] + kfold[i+1:], axis=0)
                train_data, train_label = train[:, :-1], train[:, -1:]
                validation_data, validation_label = validation[:, :-1], validation[:, -1:]
                decision_tree = BaggedTrees(maxdepth=depth, n=25, features=features,
sample_size=sample_size)
                decision_tree.fit(train_data, train_label)
                accuracies.append(np.sum(decision_tree.predict(validation_data) ==
validation_label.reshape(-1,)) / len(validation_label.reshape(-1,)))

            accuracies = np.array(accuracies)
            print("Average Validation Accuracy: ", np.mean(accuracies))
    print()

## Defined helper function to find the best depth and the number of trees for random forest
# that gives the highest average validation accuracy.
def validate_random_forest(X, y, features, m, sample_size=500):
    # Tried tuning number of trees by using num_trees of 80 through 120.
    for num_trees in [70, 80, 90, 100, 110, 120, 130]:
        # Tried tuning depth by using depths of 6 through 11.
        for depth in [6,7,8,9,10,11]:
            all_data = np.concatenate((X,y.reshape(-1, 1)), axis=1)
            np.random.shuffle(all_data)
            kfold = np.array_split(all_data, 5, axis=0)
            print("Depth: {} and Number of Trees: {}".format(depth,num_trees))
            accuracies = []
            for i in range(len(kfold)):
                validation = kfold[i]
                train = np.concatenate(kfold[:i] + kfold[i+1:], axis=0)
                train_data, train_label = train[:, :-1], train[:, -1:]
                validation_data, validation_label = validation[:, :-1], validation[:, -1:]
```

```
                random_forest = RandomForest(maxdepth=5, n=num_trees, features=features, m=m,
sample_size=sample_size)
                random_forest.fit(train_data, train_label)
                accuracies.append(np.sum(random_forest.predict(validation_data) ==
validation_label.reshape(-1,)) / len(validation_label.reshape(-1,)))

            accuracies = np.array(accuracies)
            print("Average validation accuracy: ", np.mean(accuracies))
   print()

#### Defined helper function for performance Evaluation calculation of the model and
# also save the predictions to a file.
def eval(X, y, split, model, filename=None, Z=None):
   #shuffle the data
   all_data = np.concatenate((X,y.reshape(-1,1)), axis=1)
   np.random.shuffle(all_data)
   train = all_data[:split, :]
   validation = all_data[split:, :]
   train_data, train_label = train[:, :-1], train[:, -1:].reshape(-1,)
   validation_data, validation_label = validation[:, :-1], validation[:, -1:].reshape(-1)
   model.fit(train_data, train_label)
   if filename:
       results_to_csv(model.predict(Z), filename)
       print("Predictions are calculated and are saved to the csv file. \n")

   return (np.sum(model.predict(train_data) == train_label) / len(train_label)),
(np.sum(model.predict(validation_data) == validation_label) / len(validation_label))

#### QUESTION 4.4 Performance Evaluation:
# For each of the 2 datasets, train both a decision tree and random forest and report your training and
validation
# accuracies. You should be reporting 8 numbers (2 datasets × 2 classifiers × training/validation).
if __name__ == "__main__":
   ##TITANIC DATASET
   dataset = "titanic"
   ### Params was confusing me. So I added parameters myself.
   # params = {
   #     "max_depth": 5,
   #     # "random_state": 6,
   #     "min_samples_leaf": 10,
   # }
   # N = 100
   print("\nTITANIC DATASET")
   ## Set the random seed to 150
   np.random.seed(150)
   if dataset == "titanic":
       # Load titanic data
       path_train = './dataset/titanic/titanic_training.csv'
       data = genfromtxt(path_train, delimiter=',', dtype=None, encoding=None)
       path_test = './dataset/titanic/titanic_test_data.csv'
       test_data = genfromtxt(path_test, delimiter=',', dtype=None, encoding=None)
       y = data[1:, -1]  # label = survived
       class_names = ["Died", "Survived"]
       labeled_idx = np.where(y != '')[0]
       y = np.array(y[labeled_idx])
       y = y.astype(float).astype(int)
       print("\nPart (b): preprocessing the titanic dataset")
       X, onehot_features = preprocess(data[1:, :-1], onehot_cols=[1, 5, 7, 8])
       X = X[labeled_idx, :]
       Z, _ = preprocess(test_data[1:, :], onehot_cols=[1, 5, 7, 8])
       assert X.shape[1] == Z.shape[1]
       features = list(data[0, :-1]) + onehot_features
   # print(math.ceil(math.sqrt(len(features))))
   print("Titanic Features:", features)
   print("Train/test size:", X.shape, Z.shape)
    print("\n\nPart 0: constant classifier")
   print("Accuracy", 1 - np.sum(y) / y.size)

   #### QUESTION 4.4: Performance Evaluation:
   ### Training simplified Decision Tree with tuned depth
   print("\nTraining Base Decision Tree with tuned depth ")
   # To use validation to find the best depth, uncomment and run the following line of code:
   # validate_base_decision_tree(X, y, features)
   # Depth 4 had the highest average validation accuracy
   base_decision_tree  = DecisionTree(max_depth=4, feature_labels=features)
   training_accuracy, validation_accuracy = eval(X, y, 700, base_decision_tree)
   print("Base Decision Tree Training Accuracy: {}, Validation Accuracy: {}".format( training_accuracy,
validation_accuracy))

   #### QUESTION 4.4: Performance Evaluation:
   ### Training Bagged Decision Tree
   print("\nTraining Bagged Decision Tree")
```

```
    # To use validation to find the best depth and number of trees, uncomment and run the following line
of code:
    # validate_bagged_decision_tree(X, y, features)
    # Depth 12 and 100 trees had the highest average validation accuracy
    bagged_decision_tree = BaggedTrees(maxdepth=12, n=100, features=features, sample_size=700)
    training_accuracy, validation_accuracy = eval(X, y, 700, bagged_decision_tree)
    print("Bagged Decision Tree Training Accuracy: {}, Validation Accuracy: {}".format(
training_accuracy, validation_accuracy))

    #### QUESTION 4.4: Performance Evaluation:
    ### Training Random Forest
    print("\nTraining Random Forest")
    # To use validation to find the best depth and number of trees, uncomment and run the following line
of code:
    # validate_random_forest(X, y, features, math.ceil(math.sqrt(len(features))))
    # Depth 9 and N=130 trees works best
    random_forest = RandomForest(maxdepth=9, n=130, features=features, sample_size=700,
m=math.ceil(math.sqrt(len(features))))
    training_accuracy, validation_accuracy = eval(X, y, 700, random_forest, filename="titanic.csv", Z=Z)
    print("Random Forest Training Accuracy: {}, Validation Accuracy: {}".format( training_accuracy,
validation_accuracy))

    ### Given code but didn't use
    # # Basic decision tree
    # print("\n\nPart (a-b): simplified decision tree")
    # dt = DecisionTree(max_depth=3, feature_labels=features)
    # basic_val_acc = eval(X, y, 700, dt)
    # print(dt)
    # print("Predictions", dt.predict(Z)[:100])

    # print("\n\nPart (c): sklearn's decision tree")
    # clf = sklearn.tree.DecisionTreeClassifier(random_state=0, max_depth=3)
    # clf.fit(X, y)
    # evaluate(clf)
    # out = io.StringIO()

    # # You may want to install "gprof2dot"
    # sklearn.tree.export_graphviz(
    #     clf, out_file=out, feature_names=features, class_names=class_names)
    # graph = pydot.graph_from_dot_data(out.getvalue())
    # pydot.graph_from_dot_data(out.getvalue())[0].write_pdf("%s-tree.pdf" % dataset)

    ##SPAM DATASET
    print("\nSPAM DATASET\n")
    np.random.seed(200)
    dataset = "spam"
    if dataset == "spam":
        features = [
            "pain", "private", "bank", "money", "drug", "spam", "prescription", "creative",
            "height", "featured", "differ", "width", "other", "energy", "business", "message",
            "volumes", "revision", "path", "meter", "memo", "planning", "pleased", "record", "out",
            "semicolon", "dollar", "sharp", "exclamation", "parenthesis", "square_bracket",
            "ampersand"
        ]
        assert len(features) == 32
        # Load spam data
        path_train = 'dataset/spam/spam_data.mat'
        data = scipy.io.loadmat(path_train)
        X = data['training_data']
        # print(X.shape)
        y = np.squeeze(data['training_labels'])
        Z = data['test_data']
        class_names = ["Ham", "Spam"]

    ### QUESTION 4.5 question 3: Varying maximum depths:
    # I tired depths from depth = 1 to depth = 50.
    #### Visualizing accuracies vs. depth.
    accuracies = []
    depths = [1,5,10,15,20,25,30,35,40,45,50]
    for depth in depths:
        base_dt = DecisionTree(max_depth=depth, feature_labels=features)
        #80/20 split -> 5280(0.20) = 1056 , 5280(0.80) = 4224
        training_accuracy, validation_accuracy = eval(X, y, 4224, base_dt)
        accuracies.append(validation_accuracy)
        print("Depth {} Validation Accuracy: {}".format(depth, validation_accuracy))
    fig, axes = plt.subplots(1, 1, figsize=(7, 7))
    axes.plot(depths, accuracies)
    axes.set_title("SPAM: Validation Accuracy vs. Depth of Decision Tree")
    axes.set_xlabel("Depth of Tree")
    axes.set_ylabel("Validation Accuracy")
    plt.show()
```

```
    #### QUESTION 4.4: Performance Evaluation:
    ### Training Base Decision Tree
    print("\nTraining Base Decision Tree")
    # Depth of 30 had the highest validation accuracy
    base_decision_tree = DecisionTree(max_depth=30, feature_labels=features)
    # 80/20 split -> 5280(0.20) = 1056 , 5280(0.80) = 4224
    training_accuracy, validation_accuracy = eval(X, y, 4224, base_decision_tree)
    print("SPAM Base Decision Tree Training Accuracy: {}, Validation Accuracy:
{}".format(training_accuracy, validation_accuracy))

    ### QUESTION 4.5 part 2: The Splits: For your decision tree, and for a data point of your choosing
from each class (spam and ham),
    # state the splits (i.e., which feature and which value of that feature to split on) your decision
tree made to classify it
    spam_sample = X[y==1, :][0,:].reshape(1, 32)
    ham_sample = X[y==0, :][4, :].reshape(1, 32)
    #### Predictions for spam sample
    print("\n")
    base_decision_tree.predict(spam_sample, verbose=True)
    print("Therefore this email was spam.\n")
    #### Predictions for ham sample
    base_decision_tree.predict(ham_sample, verbose=True)
    print("Therefore this email was ham.\n")

    ##### QUESTION 4.4: Performance Evaluation:
    ### Training Bagged Decision Tree
    print("\nTraining Bagged Decision Tree")
    # To use validation to find the best depth and number of trees, uncomment and run the following line
of code:
    # validate_bagged_decision_tree(X, y, features, sample_size=3000)
    # Depth 14 and N=80 trees works best
    bagged_decision_tree = BaggedTrees(maxdepth=14, n=80, features=features, sample_size=3000)
    # 80/20 split -> 5280(0.20) = 1056 , 5280(0.80) = 4224
    training_accuracy, validation_accuracy = eval(X, y, 4224, bagged_decision_tree)
    print("SPAM Bagged Decision Tree Training Accuracy: {}, Validation Accuracy:
{}".format(training_accuracy, validation_accuracy))

    #### QUESTION 4.4: Performance Evaluation:
    ### Training Random Forest
    print("\nTraining Random Forest")
    # To use validation to find the best depth and number of trees, uncomment and run the following line
of code:
    # validate_random_forest(X, y, features, math.ceil(math.sqrt(len(features))))
    # Depth 9 and N=100 trees works best
    random_forest = RandomForest(maxdepth=9, n=100, features=features, sample_size=3000,
m=math.ceil(math.sqrt(len(features))))
    # 80/20 split -> 5280(0.20) = 1056 , 5280(0.80) = 4224
    training_accuracy, validation_accuracy = eval(X, y, 4224, random_forest, filename="spam.csv", Z=Z)
    print("SPAM Random Forest Training Accuracy: {}, Validation Accuracy: {}".format(training_accuracy,
validation_accuracy))

#### QUESTION 4.6 Writeup Requirements for the Titanic Dataset:
# Train and visualize a Shallow Decision tree: I trained a depth 4 tree.
# I just printed the tree since I was not able to get any of the external libraries to do well.
# Basic decision tree
    print("\n\nPart 6: Visualize a shallow Decision Tree")
    dt = DecisionTree(max_depth=4, feature_labels=features)
    basic_val_acc = eval(X, y, 700, dt)
    print("Tree:")
    print(dt)
```