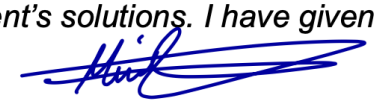
Hiva Mohammadzadeh

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.”



CS 189 Introduction to Machine Learning
Spring 2023 Jonathan Shewchuk

HW5

Due: Friday, March 24 at 11:59 pm

Submit your predictions for the test sets to Kaggle as early as possible. Include your Kaggle scores in your write-up (see below). The Kaggle competition for this assignment can be found at

- Spam: `https://www.kaggle.com/c/hw5-spam-competition-cs189sp23`
- Titanic: `https://www.kaggle.com/c/hw5-titantic-competition-cs189sp23`

Write-up: Submit your solution in **PDF** format to "Homework 5 Write-Up" on Gradescope.

- State your name, and if you have discussed this homework with anyone (other than GSIs), list the names *of them all*.
- Begin the solution for each question in a new page. Do not put content for different questions in the same page. You may use multiple pages for a question if required.
- If you include figures, graphs or tables for a question, any explanations should accompany them in *the same page*. Do NOT put these in an appendix!
- **Only PDF uploads to Gradescope will be accepted**. You may use LaTeX or Word to typeset your solution or scan a neatly handwritten solution to produce the PDF.
- **Replicate all your code in an appendix**. Begin code for each coding question in a fresh page. Do not put code from multiple questions in the same page. When you upload this PDF on Gradescope, *make sure* that you assign the relevant pages of your code from appendix to correct questions.

Code: Additionally, submit all your code as a .zip file to "Homework 5 Code" on Gradescope.

- **Set a seed for all pseudo-random numbers generated in your code.** This ensures your results are replicated when readers run your code.
- Include a README with your name, student ID, the values of the random seed (above) you used, and instructions for running (and compiling, if appropriate) your code.
- Do NOT provide any data files, but supply instructions on how to add data to your code.
- Code that the readers can't run because it requires exorbitant memory or execution time might not receive marks.
- Code submitted here must match that in the PDF Write-up, and produce the *exact* output submitted to Kaggle. Inconsistent or incomplete code might not receive marks.

1 Honor Code

Declare and sign the following statement:

"I certify that all solutions in this document are entirely my own and that I have not looked at anyone else's solution. I have given credit to all external sources I consulted."

Signature : _____

While discussions are encouraged, *everything* in your solution must be your (and only your) creation. Furthermore, all external material (i.e., *anything* outside lectures and assigned readings, including figures and pictures) should be cited properly. We wish to remind you that consequences of academic misconduct are *particularly severe*!

2 Random Forest Motivation

Ensemble learning is a general technique to combat overfitting, by combining the predictions of many varied models into a single prediction based on their average or majority vote.

(a) **The motivation of averaging.** Consider a set of uncorrelated random variables $\{Y_i\}_{i=1}^n$ with mean μ and variance σ^2. Calculate the expectation and variance of their average. (In the context of ensemble methods, these Y_i's are analogous to the prediction made by classifier i.)

(b) **Ensemble Learning – Bagging.** In lecture, we covered bagging (Bootstrap AGGregatING). Bagging is a randomized method for creating many different learners from the same data set.

Given a training set of size n, generate T random subsamples, each of size n', by sampling with replacement. Some points may be chosen multiple times, while some may not be chosen at all. If $n' = n$, around 63% are chosen, and the remaining 37% are called out-of-bag (OOB) sample points.

(i) Why 63%?
Hint: when n is very large, what is the probability that a sample point won't be selected?

(ii) If we use bagging to train our model, how do you recommend we choose the hyperparameter T? (Recall, T is the number of decision trees in the ensemble and the number of subsamples; typically, a dozen to several thousand trees are used, depending on the size and nature of the training set.)

(c) In part (a), we see that averaging reduces variance for uncorrelated classifiers. Real-world prediction will of course not be completely uncorrelated, but reducing correlation among decision trees will generally reduce the final variance. Reconsider a set of correlated random variables $\{Z_i\}_{i=1}^n$ with mean μ and variance σ^2, where each $Z_i \in \mathbb{R}$ is a scalar. Suppose $\forall i \neq j$, $\text{Corr}(Z_i, Z_j) = \rho$. Calculate the variance of their average.

(2) a) Calculate the expectation and variance of the average of the set of uncorrelated random variables.

uncorrelated variables $= \{Y_i\}_{i=1}^n$

Expectation $= E\left[\frac{1}{n}\sum_{i=1}^{n} Y_i\right] = \frac{1}{n}\sum_{i=1}^{n} E[Y_i] = \frac{1}{n} \cdot n \cdot \mu = \boxed{\mu}$ → same expectation.

Variance $= Var\left(\frac{1}{n}\sum_{i=1}^{n} Y_i\right) = \frac{1}{n^2}\sum_{i=1}^{n} Var(Y_i) = \frac{1}{n^2} \cdot n \cdot \sigma^2 = \boxed{\frac{\sigma^2}{n}}$

---

b) If $n' = n$, around 63% are chosen and the remaining 37% are OOB samples.

i) Why 63%? hint: $n \to \infty$, whats the probability that a sample is not selected?

probability of not being selected $= (1 - \frac{1}{n})^n$

as $n \to \infty$, $(1 - \frac{1}{n})^n \Rightarrow (1 - \frac{1}{\infty})^{\infty} = \frac{1}{e} = 0.3679$ which is approximately 37%.

So, probability of being chosen = 1 − probability of not chosen = 1 − 37% = 63%

---

ii) If we use bagging, how do we choose the hyper parameter T?

We can use validation accuracy and cross validation to set T just like we set any other hyperparameter.

---

c) Reducing correlation among decision trees will generally reduce the final variance. Calculate the variance of the average of the new set of correlated random variables.

Now have correlated random variables $= \{z_i\}_{i=1}^n$ and $\forall i \neq j, Corr(z_i, z_j) = \rho$.

Variance $= Var\left(\frac{1}{n}\sum_{i=1}^{n} z_i\right) = \frac{1}{n^2} Var\left(\sum_{i=1}^{n} z_i\right) = \frac{1}{n^2}\left(\underbrace{\sum_{i=1}^{n} Var(z_i)}_{\sigma^2 / n} + \sum_{1 \le i < j \le n} Cov(z_i, z_j)\right) = \frac{1}{n^2}\left((n \cdot \sigma^2) + (n(n-1)\sigma^2\rho)\right)$

$= \frac{1}{n^2}(n\sigma^2) + \frac{1}{n^2}(n(n-1)\sigma^2\rho) = \boxed{\frac{\sigma^2}{n} + \frac{(n-1)\sigma^2\rho}{n}}$

3 Gaussian Kernels

In this question, we will look at training a binary classifier with a Gaussian kernel. Specifically, given a labeled dataset $S = \{(x_i, y_i)\}_{i=1}^n \subseteq \mathbb{R}^d \times \{\pm 1\}$ and a kernel function $k(x_1, x_2)$, we consider classifiers of the form

$$\widehat{f}(x) = \text{sign}\left(\sum_{i=1}^n a_i\, k(x_i, x)\right),$$

where we define sign(u) to be 1 if $u \geq 0$ or -1 if $u < 0$. To choose the weights $a_i, i = 1, ..., n$, we consider the least-squares problem

$$a \in \arg\min_{a \in \mathbb{R}^n} \|Ka - y\|_2^2, \tag{1}$$

where $K = (k(x_i, x_j))_{i=1,\,j=1}^n$ is the kernel matrix and y is the vector of labels. We will work with the Gaussian kernel. Recall that the Gaussian kernel with bandwidth $\sigma > 0$ is

$$k(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|_2^2}{2\sigma^2}\right).$$

(a) When the bandwidth parameter $\sigma \to 0$, observe that the off-diagonal entries of the kernel matrix K tend to zero. Consider a two-point dataset S ($n = 2$) with $(x_1, y_1) = (1, 1)$ and $(x_2, y_2) = (-1, -1)$. If we assume that as $\sigma \to 0$, the off-diagonal entries of K approach zero (and the diagonal entries are unmodified), what is the optimal solution of a for the optimization problem (1) and what is the classifier $\widehat{f}(x)$?

(b) Now we consider the regime when the bandwidth parameter $\sigma \to \infty$. Observe in this regime, the off-diagonal entries of the kernel matrix K approach one. Given a dataset S, suppose we solve the optimization problem (1) with all the off-diagonal entries of K equal to one (and the diagonal entries unmodified). Prove that if the number of $+1$ labels in S equals the number of -1 labels in S, then $a = \mathbf{0}$ is an optimal solution of (1). What is the resulting classifier $\widehat{f}(x)$?

(c) Now we consider the regime when the bandwidth parameter is large but finite. Consider again the two-point dataset S with $(x_1, y_1) = (1, 1)$ and $(x_2, y_2) = (-1, -1)$. When $\sigma \gg 1$, we can approximate $k(x_1, x_2) \approx 1 + \frac{x_1 x_2}{2\sigma^2}$. Show that the solution of the optimization problem (1) with the kernel $k_a(x_1, x_2) = 1 + \frac{x_1 x_2}{2\sigma^2}$ is $a = (\sigma^2, -\sigma^2)$. What is the classifier $\widehat{f}(x)$?

optional

Hint: By Cramer's Rule, the inverse of a 2×2 *matrix is* $\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$.

(3) $\hat{f}(x) = \text{sign}\left(\sum_{i=1}^{n} a_i k(x_i, x)\right)$. $\text{sign}(u) = \begin{cases} 1 & \text{if } u \geq 0 \\ -1 & \text{if } u < 0 \end{cases}$. $a \in \underset{a \in \mathbb{R}^n}{\text{argmin}} \|ka - y\|_2^2$. $k(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|_2^2}{2\sigma^2}\right)$

a) What is the optimal solution of a for the optimization problem?

we're assuming that as $\sigma \to 0$, $k \to I$. and I is full rank. So,

our optimization problem is $a \in \underset{a \in \mathbb{R}^n}{\text{argmin}} \|ka - y\|_2^2$ which has the optimal solution of $\boxed{a = (k^T k)^{-1} k^T y = y}$

And what is the classifier $\hat{f}(x)$?

$a = y$. so, $\hat{f}(x) = \text{sign}\left(\sum_{i=1}^{n} a_i k(x_i, x)\right) = \text{sign}\left(\sum_{i=1}^{n} y_i k(x_i, x)\right)$

$\hat{f}(x) = 1(x \geq 0) - 1(x < 0)$

---

b) Prove that if the number of +1 labels in S equal the number of -1 labels in S, then $a = 0$ is an optimal solution of the optimization problem.

Assuming that as $\sigma \to \infty$, $k \to 1^{(n \times n)}$ → Not full rank → have to use normal equations and closed form.

So, $k^T k a = k^T y$ $\quad k = 1^{(n \times n)} \to k_i^T y = \sum_j y_j = 0$ And we have equal number of +1 and -1 labels. So, $k^T y = 0$

and solution is $k^T k a = 0$ and $a = 0$ is the optimal solution. Therefore proven. ✓

what is the resulting classifier $\hat{f}(x)$?

when $a = 0$, $\hat{f}(x) = \text{sign}(0) = \boxed{1}$.

4 Decision Trees for Classification

In this problem, you will implement decision trees and random forests for classification on two datasets: 1) the spam dataset and 2) a Titanic dataset to predict survivors of the infamous disaster. The data is with the assignment. See the Appendix for more information on its contents and some suggestions on data structure design.

In lectures, you were given a basic introduction to decision trees and how such trees are trained. You were also introduced to random forests. Feel free to research different decision tree techniques online. You do not have to implement boosting (which we will learn late this semester), although it might help with Kaggle.

For your convenience, we provide starter code which includes preprocessing and some decision tree functionality already implemented. Feel free to use (or not to use) this code in your implementation.

4.1 Implement Decision Trees

We expect you to implement the tree data structure yourself; you are not allowed to use a pre-existing decision tree implementation. The Titanic dataset is not "cleaned"—that is, there are missing values—so you can use external libraries for data preprocessing and tree visualization (in fact, we recommend it). Removing examples with missing features is not a good option; there is not enough data to justify throwing some of it away. Be aware that some of the later questions might require special functionality that you need to implement (e.g., maximum depth stopping criterion, visualizing the tree, tracing the path of a sample point through the tree). You can use any programming language you wish as long as we can read and run your code with minimal effort. If you choose to use our starter code, a skeleton structure of the decision tree implementation is provided, and you will decide how to fill it in. In this part of your writeup, **include your decision tree code.**

4.2 Implement a Random Forest

You are not allowed to use any off-the-shelf random forest implementation. If you architected your code well, this part should be a (relatively) easy encapsulation of the previous part. In this part of your writeup, **include your random forest code.**

4.3 Describe implementation details

We aren't looking for an essay; 1–2 sentences per question is enough.

1. How did you deal with categorical features and missing values?
2. What was your stopping criterion?
3. How did you implement random forests?
4. Did you do anything special to speed up training?
5. Anything else cool you implemented?

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 $\times$ 2 classifiers $\times$ training/validation). In addition,

for both datasets, train your best model and submit your predictions to Kaggle. Include your Kaggle display name and your public scores on each dataset. You should be reporting 2 Kaggle scores.

4.5 Writeup Requirements for the Spam Dataset

1. (Optional) If you use any other features or feature transformations, explain what you did in your report. You may choose to use something like bag-of-words. You can implement any custom feature extraction code in `featurize.py`, which will save your features to a `.mat` file.

2. 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. An example of what this might look like:

   (a) ("viagra") ≥ 2
   (b) ("thanks") < 1
   (c) ("nigeria") ≥ 3
   (d) Therefore this email was spam.

   (a) ("budget") ≥ 2
   (b) ("spreadsheet") ≥ 1
   (c) Therefore this email was ham.

3. Generate a random 80/20 training/validation split. Train decision trees with varying maximum depths (try going from depth = 1 to depth = 40) with all other hyperparameters fixed. Plot your validation accuracies as a function of the depth. Which depth had the highest validation accuracy? Write 1–2 sentences explaining the behavior you observe in your plot. If you find that you need to plot more depths, feel free to do so.

4.6 Writeup Requirements for the Titanic Dataset

Train a very shallow decision tree (for example, a depth 3 tree, although you may choose any depth that looks good) and visualize your tree. Include for each non-leaf node the feature name and the split rule, and include for leaf nodes the class your decision tree would assign. You can use any visualization method you want, from simple printing to an external library; the `rcviz` library on github works well.

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.
```

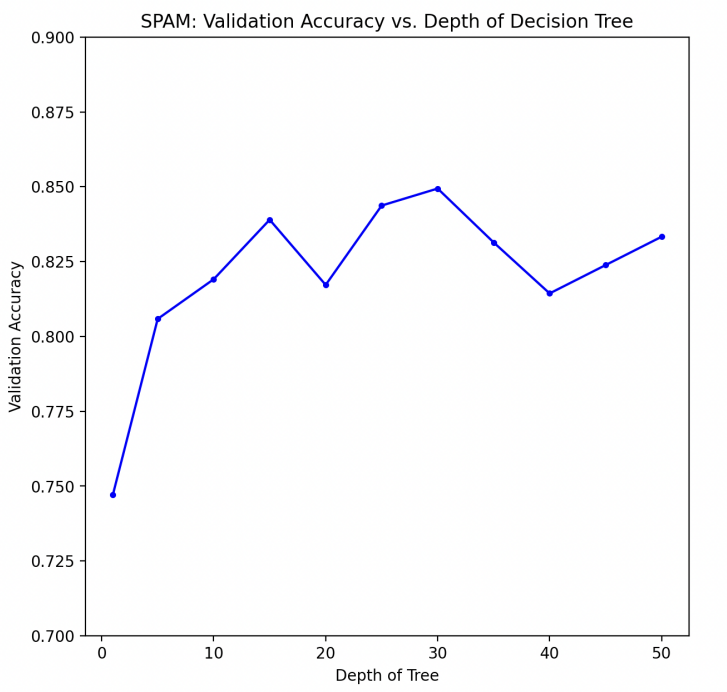
3. Varying maximum depths:

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: 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
```

```
    # 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)
```

A Appendix

Data Processing for Titanic

Here's a brief overview of the fields in the Titanic dataset. You will need to preprocess the dataset into a form usable by your decision tree code.

1. survived: the label we want to predict. 1 indicates the person survived, whereas 0 indicates the person died.
2. pclass: Measure of socioeconomic status. 1 is upper, 2 is middle, 3 is lower.
3. age: Fractional if less than 1.
4. sex: Male/female.
5. sibsp: Number of siblings/spouses aboard the Titanic.
6. parch: Number of parents/children aboard the Titanic.
7. ticket: Ticket number.
8. fare: Fare.
9. cabin: Cabin number.
10. embarked: Port of embarkation (C = Cherbourg, Q = Queenstown, S = Southampton)

You will face two challenges you did not have to deal with in previous datasets:

1. Categorical variables. Most of the data you've dealt with so far has been continuous-valued. Some features in this dataset represent types/categories. Here are two possible ways to deal with categorical variables:

   (a) (Easy) In the feature extraction phase, map categories to binary variables. For example suppose feature 2 takes on three possible values: 'TA', 'lecturer', and 'professor'. In the data matrix, these categories would be mapped to three binary variables. These would be columns 2, 3, and 4 of the data matrix. Column 2 would be a boolean feature $\{0, 1\}$ representing the TA category, and so on. In other words, 'TA' is represented by $[1, 0, 0]$, 'lecturer' is represented by $[0, 1, 0]$, and 'professor' is represented by $[0, 0, 1]$. Note that this expands the number of columns in your data matrix. This is called "vectorizing," or "one-hot encoding" the categorical feature.

   (b) (Hard, but more generalizable) Keep the categories as strings or map the categories to indices (e.g. 'TA', 'lecturer', 'professor' get mapped to $0, 1, 2$). Then implement functionality in decision trees to determine split rules based on the subsets of categorical variables that maximize information gain. You cannot treat these as normal continuous-valued features because ordering has no meaning for these categories (the fact that $0 < 1 < 2$ has no significance when $0, 1, 2$ are discrete categories).

2. Missing values. Some data points are missing features. In the `csv` files, these are represented by the value '`?`'. You have three approaches:

(a) (Easiest) If a data point is missing some features, remove it from the data matrix (**this is useful for your first code draft, but your submission must not do this**).

(b) (Easy) Infer the value of the feature from all the other values of that feature (e.g., fill it in with the mean, median, or mode of the feature. Think about which of these is the best choice and why).

(c) (Hard, but more powerful). Use k-nearest neighbors to impute feature values based on the nearest neighbors of a data point. In your distance metric you will need to define the distance to a missing value.

(d) (Hardest, but more powerful) Implement within your decision tree functionality to handle missing feature values based on the current node. There are many ways this can be done. You might infer missing values based on the mean/median/mode of the feature values of data points sorted to the current node. Another possibility is assigning probabilities to each possible value of the missing feature, then sorting fractional (weighted) data points to each child (you would need to associate each data point with a weight in the tree).

For Python:

It is recommended you use the following classes to write, read, and process data:

```
csv.DictReader
sklearn.feature_extraction.DictVectorizer (vectorizing categorical variables)
    (There's also sklearn.preprocessingOneHotEncoder, but it's much less clean)
sklearn.preprocessing.LabelEncoder
    (if you choose to discretize but not vectorize categorical variables)
sklearn.preprocessing.Imputer
    (for inferring missing feature values in the preprocessing phase)
```

If you use `csv.DictReader`, it will automatically parse out the header line in the `csv` file (first line of the file) and assign values to fields in a dictionary. This can then be consumed by `DictVectorizer` to binarize categorical variables.

To speed up your work, you might want to store your cleaned features in a file, so that you don't need to preprocess every time you run your code.

Approximate Expected Performance

For spam, with a single decision tree, we got 79.9% validation accuracy. With a random forest, we get around 80.4% validation accuracy on Titanic. You might not do quite this well. We will post cutoffs on Piazza.

Suggested Architecture

This is a complicated coding project. You should put in some thought about how to structure your program so your decision trees don't end up as horrific forest fires of technical debt. Here is a rough, **optional** spec that only covers the barebones decision tree structure. This is only for your benefit—writing clean code will make your life easier, but we won't grade you on it. There are many different ways to implement this.

Your decision trees ideally should have a well-encapsulated interface like this:

```
classifier = DecisionTree(params)
classifier.train(train_data, train_labels)
predictions = classifier.predict(test_data)
```

where `train_data` and `test_data` are 2D matrices (rows are data, columns are features).

A decision tree (or **DecisionTree**) is a binary tree composed of **Nodes**. You first initialize it with the necessary parameters (which depend on what techniques you implement). As you train your tree, your tree should create and configure **Nodes** to use for classification and store these nodes internally. Your **DecisionTree** will store the root node of the resulting tree so you can use it in classification.

Each **Node** has left and right pointers to its children, which are also nodes, though some (like leaf nodes) won't have any children. Each node has a split rule that, during classification, tells you when you should continue traversing to the left or to the right child of the node. Leaf nodes, instead of containing a split rule, should simply contain a label of what class to classify a data point as. Leaf nodes can either be a special configuration of regular **Nodes** or an entirely different class.

Node fields:

- `split_rule`: A length 2 tuple that details what feature to split on at a node, as well as the threshold value at which you should split. The former can be encoded as an integer index into your data point's feature vector.
- `left`: The left child of the current node.
- `right`: The right child of the current node.
- `label`: If this field is set, the **Node** is a leaf node, and the field contains the label with which you should classify a data point as, assuming you reached this node during your classification tree traversal. Typically, the label is the mode of the labels of the training data points arriving at this node.

DecisionTree methods:

- `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.
- `information_gain(features, labels, threshold)`: A method that takes in some feature of the data, the labels and a threshold, and compute the information gain of a split using the threshold.
- `entropy(label)`: A method that takes in the labels of data stored at a node and compute the entropy (or Gini impurity).
- `fit(data, labels)`: Grows a decision tree by constructing nodes. Using the entropy and segmenter methods, it attempts to find a configuration of nodes that best splits the input data. This function figures out the split rules that each node should have and figures out when to stop growing the tree and insert a leaf node. There are many ways to implement this, but eventually your DecisionTree should store the root node of the resulting tree so you can use the tree for classification later on. Since the height of your DecisionTree shouldn't be astronomically large (you may want to cap the height—if you do, the max height would be a hyperparameter), this method is best implemented recursively.
- `predict(data)`: Given a data point, traverse the tree to find the best label to classify the data point as. Start at the root node you stored and evaluate split rules at each node as you traverse until you reach a leaf node, then choose that leaf node's label as your output label.

Random forests can be implemented without code duplication by storing groups of decision trees. You will have to train each tree on different subsets of the data (data bagging) and train nodes in each tree on different subsets of features (attribute bagging). Most of this functionality should be handled by a random forest class, except attribute bagging, which may need to be implemented in the decision tree class. Hopefully, the spec above gives you a good jumping-off point as you start to implement your decision trees. Again, it's highly recommended to think through design before coding.

Happy hacking!

Submission Checklist

Please ensure you have completed the following before your final submission.

At the beginning of your writeup...

1. Have you copied and hand-signed the honor code specified in Question 1?
2. Have you listed all students (Names and ID numbers) that you collaborated with?

In your writeup for Question 4...

1. Have you included your **Kaggle Score** and **Kaggle Username**?
2. Have you included your generated plots and visualizations?

At the end of the writeup...

1. Have you provided a code appendix including all code you wrote in solving the homework?

Executable Code Submission

1. Have you created an archive containing all ".py" files that you wrote or modified to generate your homework solutions?
2. Have you removed all data and extraneous files from the archive?
3. Have you included a `README` in your archive containing any special instructions to reproduce your results?

Submissions

1. Have you submitted your written solutions to the Gradescope assignment titled **HW5 Write-Up** and selected pages appropriately?
2. Have you submitted your executable code archive to the Gradescope assignment titled **HW5 Code**?
3. Have you submitted your test set predictions for **Spam** and **Titanic** dataset to the appropriate Kaggle challenges?
4. Is your Kaggle submission in integer format? Submissions in decimal format will recieve a score of zero!

Congratulations! You have completed Homework 5.