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.

Q1 (a)

(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.)

$$\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n} Y_i\right] = \frac{1}{n}\mathbb{E}\left[\sum_{i=1}^{n} Y_i\right]$$

$$= \frac{1}{n}\left(\mathbb{E}[Y_1] + \mathbb{E}[Y_2] + \cdots + \mathbb{E}[Y_n]\right) \quad \text{lin. of Exp.}$$

$$= \frac{1}{n}\left(\mu + \cdots + \mu\right)$$

$$= \frac{n\mu}{n}$$

$$= \boxed{\mu}$$

$$\mathrm{Var}\left(\frac{1}{n}\sum_{i=1}^{n} Y_i\right) = \frac{1}{n^2}\mathrm{Var}\left(\sum_{i=1}^{n} Y_i\right)$$

$$= \frac{1}{n^2}\sum_{i=1}^{n}\mathrm{Var}(Y_i)$$

$$= \frac{1}{n^2}\sum_{i=1}^{n}\sigma^2$$

$$= \frac{n\sigma^2}{n^2}$$

$$= \boxed{\frac{\sigma^2}{n}}$$

Q1 (b)

(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.)

(i)

Let A = sample point i is not selected

When $n = n'$

$$P(A) = \left(\frac{n-1}{n}\right)^n$$

$$\rightarrow \lim_{n\to\infty} \left(\frac{n-1}{n}\right)^n = \lim_{n\to\infty} \left(1 - \frac{1}{n}\right)^n$$

$$= \frac{1}{e}$$

$$\approx 0.3679$$

$$\Rightarrow P(\text{sample point i is selected}) = 1 - P(A) = 1 - 0.37$$

$$\approx 0.63$$

$$\Rightarrow P(\text{i is selected}) = 63\%$$

(ii)

If you train on the training set and test using K-fold cross validation, you can find the optimal T value.

Q1 (c)

(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.

$$\text{Cov}(Z_i, Z_j) = \mathbb{E}[(Z_i - \mu_i)(Z_j - \mu_j)]$$

$$\text{Var}\left(\frac{1}{n}\sum_{i=1}^{n} Z_i\right) = \frac{1}{n^2}\text{Var}\left(\sum_{i=1}^{n} Z_i\right)$$

$$= \frac{1}{n^2}\mathbb{E}\left[(Z_1 - \mu) + \cdots + (Z_n - \mu)\right]^2$$

$$= \frac{1}{n^2}\left(n\text{var}(Z_1) + n(n-1)\text{cov}(Z_1, Z_2)\right)$$

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

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

$$= \frac{\sigma^2}{n} + \frac{n^2\rho - n\rho}{n^2}\sigma^2$$

$$= \frac{\sigma^2}{n} + \frac{n\rho - \rho}{n}\sigma^2$$

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

As $n \to \infty$, $\rho\sigma^2$ is left

$$\Rightarrow \rho\sigma^2$$

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 $\widehat{f}(x)$?

optional ?

(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 $\widehat{f}(x)$?

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

Q3 (a)

(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 $\widehat{f}(x)$?

$$x = [[1], [-1]] \qquad y = [1, -1]$$

$$a \in \underset{a \in \mathbb{R}^n}{\operatorname{argmin}} \| Ka - y \|_2^2$$

$$k(x_1, x_1) = \exp\left(\frac{\| x_1 - x_1 \|_2^2}{2\sigma^2}\right) = \exp(0) = 1 = K(x_2, x_2)$$

$$k(x_1, x_2) = \exp\left(\frac{\| x_1 - x_2 \|_2^2}{2\sigma^2}\right) = \exp\left(\frac{\| 2 \|_2^2}{2\sigma}\right)$$

$$= \exp\left(\frac{2}{\sigma^2}\right) \longrightarrow 0 \text{ as } \sigma \to 0$$

$K_{1,2} = K_{2,1}$

Thus, when $\sigma \to 0$, $K = I_{2\times 2} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

Taking the gradient and setting it equal to 0:

$$\| Ka - y \|_2^2 = (Ka - y)^T (Ka - y)$$

$$= (a^T K^T - y^T)(Ka - y)$$

$$= a^T K^T K a - 2a^T K^T y + y^T y = g(x)$$

$$\nabla g(x) = 2K^T K a - 2K^T y = 0$$

$$= K^T K a = K^T y$$

$$\Longrightarrow a^* = (K^T K)^{-1} K^T y$$

Since $K = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$,

$$a^* = \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right)^{-1} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

$$= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

$$\Rightarrow \boxed{a^* = \begin{bmatrix} 1 \\ -1 \end{bmatrix}}$$

$$\hat{f}(x) = \operatorname{sign}\left(\sum_{i=1}^{2} a_i K(x_i, x) \right) = \operatorname{sign}\left(1 \cdot K(x_1, x) + (-1) \cdot K(x_2, x) \right)$$

$$= \operatorname{sign}\left(K(x_1, x) - K(x_2, x) \right)$$

$$\Rightarrow \hat{f}(x) = \begin{cases} 1 & \text{if } K(x_1, x) \geq K(x_2, x) \\ -1 & \text{else} \end{cases}$$

Q3 (b.)

(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 $\widehat{f}(x)$?

Prove if +1 labels = -1 labels, then $a = 0$ is optimum

Base Case: $n = 2$

By part (a.) we know when $\sigma \to \infty$, $K_{i,i} = 1$

By our givens, $K_{i,j} = 1$ (off diagonal entries)

Thus, if $\sigma \to \infty$, the kernel matrix is equal to the ones matrix $K = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$

If the number of +1 labels = # -1 labels for $n = 2$,

$$y = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

By (a.) $(K^T K)^{-1} a^* = K^T y \to \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix} a^* = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix}$

$$= \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} 1 - 1 \\ 1 - 1 \end{pmatrix}$$

$$= 2a_1 + 2a_2 = 0$$

$$2a_1 + 2a_2 = 0$$

$$= 2a_1 = -2a_2$$

$$= a_1 = -a_2$$

The only way $a_1 = -a_2$ is if $a_1 = 0 = a_2$. Thus,

$a^* = \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \vec{0}$ so the optimal solution is $a^* = \vec{0}$

for $n = 2$. Thus, proved for the base case.

The resulting classifier for $\hat{f}(x)$ is

$$\hat{f}(x) = \text{sign}\left(\sum_{i=1}^{2} a_i K(x_i, x)\right) = \text{sign}\left(0 \cdot K(x_1, x) + 0 \cdot K(x_2, x)\right)$$

$$= \text{sign}(0)$$

$$= 1$$

$$\Rightarrow \quad \hat{f}(x) = 1$$

Inductive Step: Suppose the statement holds for $n = m$. We want to show that it follows that $n = m+2$ holds true.

Note: we must have an even number of rows in order for the problem description to hold. I.e. to ensure that there are an equal number of +1 and -1 labels, $n = 2p$ for some p. Thus, $n = m$ and m is even so $n = m+2$ is also even

By our base case, we saw K is the ones matrix, $K \in \mathbb{R}^{2\times 2} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$

Extending this to the $m+1 \times m+1$ case, $K_{i,i} = 1$ and $K_{i,j} = 1$ so K is still the ones matrix, $K = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ \vdots & \ddots & & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}$

By (a.), $(K^T K)^{-1} a^* = K^T y$ where y has an equal amount of 1 and -1 labels

$$\Rightarrow \quad \left(\begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}\begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}\right)^{-1} \begin{pmatrix} a_1 \\ \vdots \\ a_{m+2} \end{pmatrix} = \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}\begin{bmatrix} 1 \\ \vdots \\ -1 \end{bmatrix}$$

$$\longrightarrow \quad \begin{bmatrix} m+2 & \cdots & m+2 \\ \vdots & \ddots & \vdots \\ m+2 & \cdots & m+2 \end{bmatrix}\begin{bmatrix} a_1 \\ \vdots \\ a_{m+2} \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

← each entry is 0 since $K^T y = \vec{0}$. The +1 and -1 labels cancel out

$\longrightarrow$ $m+2$ system of equations

$$\begin{cases} (m+2)a_1 + \cdots + (m+2)a_m + (m+2)a_{m+1} + (m+2)a_{m+2} = 0 \\ (m+2)a_1 + \cdots + (m+2)a_m + (m+2)a_{m+1} + (m+2)a_{m+2} = 0 \\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ (m+2)a_1 + \cdots + (m+2)a_m + (m+2)a_{m+1} + (m+2)a_{m+2} = 0 \end{cases}$$

By our inductive hypothesis, we know when $n = m$, the statement holds, so for a^* where $n = m$, $a_i^* = 0$ for $1 \leq i \leq m$.
Thus, we can cancel out all of the a_i $1 \leq i \leq m$ terms to 0.

$\longrightarrow$

$$\begin{cases} \cancel{(m+2)a_1 + \cdots + (m+2)a_m} + (m+2)a_{m+1} + (m+2)a_{m+2} = 0 \\ \cancel{(m+2)a_1 + \cdots + (m+2)a_m} + (m+2)a_{m+1} + (m+2)a_{m+2} = 0 \\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ \cancel{(m+2)a_1 + \cdots + (m+2)a_m} + (m+2)a_{m+1} + (m+2)a_{m+2} = 0 \end{cases}$$

(each cancelled block $\to 0$)

Thus, we are left with $m+2$ equations:

$$\longrightarrow (m+2)a_{m+1} + (m+2)a_{m+2} = 0$$

$$\longrightarrow \cancel{(m+2)}\, a_{m+1} = -\cancel{(m+2)}\, a_{m+2}$$

$$\longrightarrow a_{m+1} = -a_{m+2}$$

The only way $a_{m+1} = -a_{m+2}$ is if $a_{m+1} = 0 = -a_{m+2}$.
Thus, $a_{m+1} = 0 = a_{m+2}$ so $a_i = 0$ for $1 \leq i \leq m+2$ and

$$a^* = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \vec{0} \quad \text{for } n = m+2.$$

Thus, the statement holds for $n = m+2$. Hence, $a^* = \vec{0}$ when the # of $+1$ labels = # -1 labels.

It follows that $\hat{f}(x) = 1$ since

$$\hat{f}(x) = \text{sign}\left(\sum_{i=1}^{n} a_i K(x_i, x) \right) = \text{sign}\left(\sum_{i=1}^{n} 0 \cdot K(x_i, x) \right)$$

$$= \text{sign}(0)$$

$$= +1$$

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.

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.

Q4 (1)

Decision Tree Code

```
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
        # adding max number of features m
        self.m = m


    @staticmethod
    def information_gain(X, y, thresh):
        # TODO: implement information gain function
        # H(S) - H(after)

        splitl = np.where(X <= thresh)
        splitr = np.where(X > thresh)

        # Splitting the labels based on the threshold
        sl = y[splitl]
        sr = y[splitr]

        HS = DecisionTree.entropy(y)
        HSL = DecisionTree.entropy(sl)
        HSR = DecisionTree.entropy(sr)

        H_after = ((len(sl) * HSL) + (len(sr) * HSR)) / (len(sl) + len(sr))

        return (HS - H_after)

    @staticmethod
    def gini_impurity(X, y, thresh):
        # TODO: implement gini impurity function
        pass
```

```
  @staticmethod
  def entropy(y):
      # TODO: implement entropy function
      H = 0
      sorted_y, unique_counts = np.unique(y, return_counts = True)
      num_classes = len(sorted_y)
      for i in unique_counts:
          pc = i / num_classes
          product = pc * np.log2(pc)
          H += product
      return H

  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.

         # picking m random features out of total d features and selecting only those features for X
         data = X
         if self.m:
             m_feature_ids = np.random.choice(X.shape[1], size = self.m, replace = False)
             X = data[:, m_feature_ids]
         else:
             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])
         ])

         if self.m:
             for i in range(self.m):
                 gains.append([self.information_gain(X[:, i], y, t) for t in thresh[i, :]])
         else:
             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]
         if self.m:
             self.split_idx = m_feature_ids[self.split_idx]

         # splitting on the original data
         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)
             self.left.fit(X0, y0)
             self.right = DecisionTree(
                 max_depth=self.max_depth - 1, feature_labels=self.features)
             self.right.fit(X1, y1)
         else:
             self.max_depth = 0
             self.data, self.labels = X, y
             self.pred = stats.mode(y).mode[0]
     else:
         self.data, self.labels = X, y
         self.pred = stats.mode(y).mode[0]
     return self

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

Q4 (2.)

Random Forest

```
class BaggedTrees(BaseEstimator, ClassifierMixin):
    def __init__(self, m_depth = 3, n = 200, features = None):
        self.n = n
        self.m_depth = m_depth
        self.decision_trees = [
            DecisionTree(max_depth = m_depth, feature_labels = features) for i in range(self.n)]

    def fit(self, X, y):
        # TODO: implement function
        for n in range(self.n):
            row = X.shape[0]
            sample = np.random.choice(np.arange(row), size = row, replace = True)
            X_train = X[sample]
            y_train = y[sample]
            self.decision_trees[n].fit(X_train, y_train)
        return self

    def predict(self, X):
        # TODO: implement function
        y_hats = []
        for i in range(self.n):
            pred = self.decision_trees[i].predict(X)
            y_hats.append(pred)

        return np.round(np.mean(y_hats, axis=0))
```

```
class RandomForest(BaggedTrees):
    def __init__(self, m_depth, features, n=200, m=1):
        # TODO: implement function
        self.params = params
        self.m = m
        self.n = n
        self.decision_trees = [
            DecisionTree(max_depth = m_depth, feature_labels = features, m = m) for i in range(self.n)]
```

4.3 Describe Implementation Details

1.) The process to clean the data consisted of removing rows where labels (whether they survived or not) were completely missing. This ensured that every row would have a corresponding label. In addition, hotcoding allowed categorical variables such as gender or embarked to be mapped to binary vectors of zeros and ones. This helped deal with the catagorical features. In addition, to account for missing data values, we could use the mode of each feature and input that into each missing data value.

2.) I used the maximum depth as my stopping criterion. A depth that is too high can result in overfitting, so I had to make sure to use a depth that would not increase the bias too much.

3.) I added a m value to my decision tree class to choose m features before splitting between features. Additionally, since my Random Forest inherited my BaggedTrees class, I simply created a list of decision trees (a forest if you will) using m features. For my BaggedTrees class, I sampled with replacement and trained on each sample. I had my forest pick the the optimal sample from my BaggedTrees class.

4.) I did not do anything special to optimize speed.

5.) No

Q4.4 Performance Evaluation

```
decisionTree = DecisionTree(max_depth = 6)
randomForest = RandomForest(m_depth = 5, features = feat, n=200, m=4)

decisionTree.fit(spam_training_data, spam_training_labels)
randomForest.fit(spam_training_data, spam_training_labels)

spam_training_pred = decisionTree.predict(spam_training_data)
spam_val_pred = decisionTree.predict(spam_val_data)
spam_training_pred_forest = randomForest.predict(spam_training_data)
spam_val_pred_forest = randomForest.predict(spam_val_data)

# Training accuracies
spam_training_acc = np.sum(spam_training_pred == spam_training_labels.flatten()) / spam_training_data.shape[0]
spam_training_acc_forest = np.sum(spam_training_pred_forest == spam_training_labels.flatten()) / spam_training_

# Validation accuracies
spam_val_acc = np.sum(spam_val_pred == spam_val_labels.flatten()) / spam_val_data.shape[0]
spam_val_acc_forest = np.sum(spam_val_pred_forest == spam_val_labels.flatten()) / spam_val_data.shape[0]

print(f"Training accuracy for SPAM using a Decision Tree: {spam_training_acc}.")
print(f"Validation accuracy for SPAM using a Decision Tree: {spam_val_acc}.")

print(f"Training accuracy for SPAM using a Random Forest: {spam_training_acc_forest}.")
print(f"Validation accuracy for SPAM using a Random Forest: {spam_val_acc_forest}.")
```

```
Training accuracy for SPAM using a Decision Tree: 0.790719696969697.
Validation accuracy for SPAM using a Decision Tree: 0.7935606060606061.
Training accuracy for SPAM using a Random Forest: 0.7831439393939394.
Validation accuracy for SPAM using a Random Forest: 0.7945075757575758.
```

```
decisionTree = DecisionTree(max_depth = 5)
randomForest = RandomForest(m_depth = 6, features = titanicFeatures, n=200, m=4)

decisionTree.fit(titanic_training_data, titanic_training_labels)
randomForest.fit(titanic_training_data, titanic_training_labels)

titanic_training_pred = decisionTree.predict(titanic_training_data)
titanic_val_pred = decisionTree.predict(titanic_val_data)
titanic_training_pred_forest = randomForest.predict(titanic_training_data)
titanic_val_pred_forest = randomForest.predict(titanic_val_data)

# Training accuracies
titanic_training_acc = np.sum(titanic_training_pred == titanic_training_labels.flatten()) / titanic_training_d
titanic_training_acc_forest = np.sum(titanic_training_pred_forest == titanic_training_labels.flatten()) / tita

# Validation accuracies
titanic_val_acc = np.sum(titanic_val_pred == titanic_val_labels.flatten()) / titanic_val_data.shape[0]
titanic_val_acc_forest = np.sum(titanic_val_pred_forest == titanic_val_labels.flatten()) / titanic_val_data.sh

print(f"Training accuracy for Titanic using a Decision Tree: {titanic_training_acc}.")
print(f"Validation accuracy for Titanic using a Decision Tree: {titanic_val_acc}.")

print(f"Training accuracy for Titanic using a Random Forest: {titanic_training_acc_forest}.")
print(f"Validation accuracy for Titanic using a Random Forest: {titanic_val_acc_forest}.")
```

```
Training accuracy for Titanic using a Decision Tree: 0.65625.
Validation accuracy for Titanic using a Decision Tree: 0.5778894472361809.
Training accuracy for Titanic using a Random Forest: 0.78375.
Validation accuracy for Titanic using a Random Forest: 0.7487437185929648.
```

Kaggle Username: nadaleek

SPAM Score: 0.80394

Titanic Score: 0.73548

4.5 Writeup Spam

drug > 1e-5

exclamation > 1e-5

semicolon < 1e-5

perscription < 1e-5

spam > 1e-5

money > 1e-5

Therefore, the prediction is spam

business > 1e-5

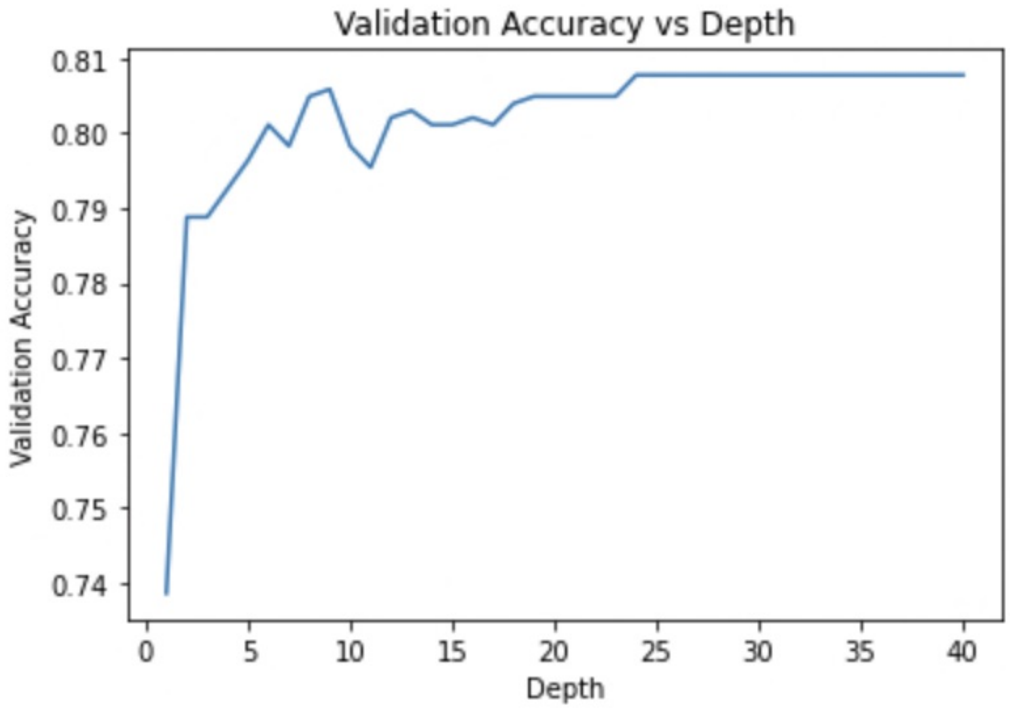
message > 1e-5

pain < 1e-5

ampersand < 1e-5

other < 1e-5

Therefore, this prediction is ham




```
The optimal depth is 23
```

4.6 Writeup for the Titanic Dataset

```
titanicModel = DecisionTree(max_depth = 3)
mod = titanicModel.fit(titanicData, titanicLabels)
```

```
fig = plt.figure(figsize=(25,20))
_ = sklearn.tree.plot_tree(titanicModel, feature_names = titanicFeatures)
plt.show()
```