CS 189 / 289 Introduction to Machine Learning
Fall 2024 Jennifer Listgarten, Saeed Saremi **HW5**

Due 11/08/24 11:59 pm PT

- Homework 5 consists of both written and coding questions.
- We prefer that you typeset your answers using LaTeX or other word processing software. If you haven't yet learned LaTeX, one of the crown jewels of computer science, now is a good time! Neatly handwritten and scanned solutions will also be accepted for the written questions.
- In all of the questions, **show your work**, not just the final answer.

Deliverables:

1. Submit a PDF of your homework to the Gradescope assignment entitled "HW 5 Written". Submit your code to the Gradescope assignment titled "HW 5 Code". **Please start each question on a new page.** If there are graphs, include those graphs in the correct sections. **Do not** put them in an appendix. We need each solution to be self-contained on pages of its own.
2. Please see the submission checklist at the end of this document for detailed submission guidelines.

1 *k*-Means Demo

Work through the entire ***Colab notebook*** [1].

Deliverables: Include a PDF export of the completed notebook in your write-up. In addition, submit the .ipynb file to the code assignment.

[1]https://drive.google.com/file/d/1RvCBdIIZUk-Z9E3dRXBHsixfSNhdPfWC/view?usp=drive$_l$*ink*

2 Exploring Bias & Variance with Ridge and OLS

Recall the statistical model for ridge regression from lecture. We have a design matrix $\mathbf{X}$, where the rows of $\mathbf{X} \in \mathbb{R}^{n\times d}$ are our data points $\mathbf{x}_i \in \mathbb{R}^d$. We assume a linear regression model

$$Y = \mathbf{X}\mathbf{w}^* + \mathbf{z}$$

where $\mathbf{w}^* \in \mathbb{R}^d$ is the true parameter we are trying to estimate, $\mathbf{z} = [z_1, \dots, z_n]^\top \sim \mathcal{N}(0, \sigma^2\mathbf{I}_n)$, and $Y = [y_1, \dots, y_n]^\top$ is the random variable representing our labels.

Throughout this problem, you may assume that $\mathbf{X}$ is full column rank. Given a realization of the labels $Y = \mathbf{y}$, recall these two estimators that we have studied so far:

$$\begin{aligned}\mathbf{w}_{\text{ols}} &= \min_{\mathbf{w}\in\mathbb{R}^d} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 \\ \mathbf{w}_{\text{ridge}} &= \min_{\mathbf{w}\in\mathbb{R}^d} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_2^2\end{aligned}$$

Also recall that the solutions for $\mathbf{w}_{\text{ols}}$ and $\mathbf{w}_{\text{ridge}}$ are

$$\begin{aligned}\mathbf{w}_{\text{ols}} &= (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y} \\ \mathbf{w}_{\text{ridge}} &= (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}\end{aligned}$$

(a) Let $\hat{\mathbf{w}} \in \mathbb{R}^d$ denote any estimator of $\mathbf{w}^*$. In the context of this problem, an estimator $\hat{\mathbf{w}} = \hat{\mathbf{w}}(Y)$ is any function which takes the data $\mathbf{X}$ and a realization of Y, and computes a guess for $\mathbf{w}^*$.

Define the MSE (mean squared error) of the estimator $\hat{\mathbf{w}}$ as

$$\text{MSE}(\hat{\mathbf{w}}) := \mathbb{E}\left[\|\hat{\mathbf{w}} - \mathbf{w}^*\|_2^2\right].$$

Above, the expectation is taken w.r.t. the randomness inherent in $\mathbf{z}$. Note that this is a multivariate generalization of the mean squared error we have seen previously.

Define $\hat{\boldsymbol{\mu}} := \mathbb{E}[\hat{\mathbf{w}}]$. Show that the MSE decomposes as such:

$$\text{MSE}(\hat{\mathbf{w}}) = \underbrace{\|\hat{\boldsymbol{\mu}} - \mathbf{w}^*\|_2^2}_{\text{Bias}(\hat{\mathbf{w}})^2} + \underbrace{\text{Tr}(\text{Cov}(\hat{\mathbf{w}}))}_{\text{Var}(\hat{\mathbf{w}})}$$

Note that this is a multivariate generalization of the bias-variance decomposition we have seen previously.

Hint: The inner product of two vectors is the trace of their outer product. Also, expectation and trace commute so $\mathbb{E}[\text{Tr}(A)] = \text{Tr}(\mathbb{E}[A])$ for any square matrix A.

(b) Show that

$$\begin{aligned}\mathbb{E}[\mathbf{w}_{\text{ols}}] &= \mathbf{w}^* \\ \mathbb{E}[\mathbf{w}_{\text{ridge}}] &= (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_d)^{-1}\mathbf{X}^\top\mathbf{X}\mathbf{w}^*\end{aligned}$$

That is, $\text{Bias}(\mathbf{w}_{\text{ols}}) = 0$, and hence $\mathbf{w}_{\text{ols}}$ is an *unbiased* estimator of $\mathbf{w}^*$, whereas $\mathbf{w}_{\text{ridge}}$ is a *biased* estimator of $\mathbf{w}^*$.

(c) Let $\{\gamma_i\}_{i=1}^d$ denote the d eigenvalues of the matrix $\mathbf{X}^\top\mathbf{X}$. Show that

$$\mathrm{Tr}(\mathrm{Cov}(\mathbf{w}_{\mathrm{ols}})) = \sigma^2 \sum_{i=1}^{d} \frac{1}{\gamma_i}, \qquad \mathrm{Tr}(\mathrm{Cov}(\mathbf{w}_{\mathrm{ridge}})) = \sigma^2 \sum_{i=1}^{d} \frac{\gamma_i}{(\gamma_i + \lambda)^2} .$$

Finally, use these formulas to conclude that

$$\mathrm{Var}(\mathbf{w}_{\mathrm{ridge}}) < \mathrm{Var}(\mathbf{w}_{\mathrm{ols}}) .$$

Note that this is opposite of the relationship between the bias terms.

Hint: Remember the relationship between the trace and the eigenvalues of a matrix. Also, for the ridge variance, consider writing $\mathbf{X}^\top\mathbf{X}$ in terms of its eigendecomposition $\mathbf{U}\mathbf{\Sigma}\mathbf{U}^\top$. Note that $\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_d$ has the eigendecomposition $\mathbf{U}(\mathbf{\Sigma} + \lambda\mathbf{I}_d)\mathbf{U}^\top$.

3 Running Time of k-Nearest neighbor Search Methods

The method of k-nearest neighbors is a fundamental conceptual building block of machine learning. A classic example is the k-nearest neighbor classifier, which is a non-parametric classifier that finds the k closest examples in the training set to the test example, and then outputs the most common label among them as its prediction. Generating predictions using this classifier requires an algorithm to find the k closest examples in a possibly large and high-dimensional dataset, which is known as the k-nearest neighbor search problem. More precisely, given a set of n points, $\mathcal{D} = \{\mathbf{x}_1 \ldots, \mathbf{x}_n\} \subseteq \mathbb{R}^d$ and a query point $\mathbf{z} \in \mathbb{R}^d$, the problem requires finding the k points in $\mathcal{D}$ that are the closest to $\mathbf{z}$ in Euclidean distance.

This problem explores the computational complexity of nearest-neighbor methods to show how naïve implementations perform very poorly as the dimensionality of the problem grows, but more sophisticated use of randomized techniques can do better.

Overall Hint: In this problem, reading later parts will help you know what you need to do in earlier parts in case you can't figure it out. So, read ahead before asking a question.

(a) Let's analyze the computational complexity of this algorithm. First, we consider the naïve exhaustive search algorithm, which computes the distance between $\mathbf{z}$ and all points in $\mathcal{D}$ and then returns the k points with the shortest distance. This algorithm first computes distances between the query and all points, then finds the k shortest distances using quickselect[2]. **What is the (average case) time complexity of running the overall algorithm for a single query?**

(b) Decades of research have focused on devising a way of preprocessing the data so that the k-nearest neighbors for each query can be found efficiently. "Efficient" means the time complexity of finding the k-nearest neighbors is lower than that of the naïve exhaustive search algorithm—meaning that the complexity must be *sublinear* in n.

Many efficient algorithms for k-nearest neighbor search rely on a divide-and-conquer strategy known as space partitioning. The idea is to divide the feature space into cells and maintain a data structure that keeps track of the points that lie in each. Then, to find the k-nearest neighbors of a query, these algorithms look up the cell that contains the query and obtain the subset of points in $\mathcal{D}$ that lie in the cell and adjacent cells. Adjacent cells must be included in case the query point is in the corner of its cell. Then, exhaustive search is performed on this subset to find the k points that are the closest to the query.

For simplicity, we'll consider the special case of $k = 1$ in the following questions, but note that the various algorithms we'll consider can be easily extended to the setting with arbitrary k. We first consider a simple partitioning scheme, where we place a Cartesian grid (a rectangular grid consisting of hypercubes) over the feature space.

How many cells need to be searched in total if the data points are one-dimensional? Two-dimensional? d-dimensional? If each cell contains one data point, what is the time com-

[2]Quickselect is a counterpart of quicksort that just picks the top k in an unordered list. Instead of taking $O(n \log n)$ like quicksort on average, it takes $O(n)$. Just realize that there is no point in recursively sorting things that for sure aren't going to be in the top k.

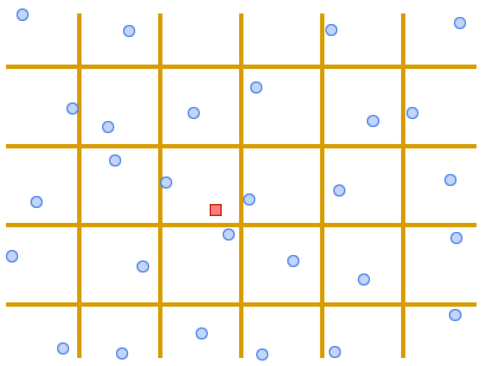


Figure 1: Illustration of the space partitioning scheme we consider. The data points are shown as blue circles and the query is shown as the red square. The cell boundaries are shown as gold lines.

plexity for finding the 1-nearest neighbor in terms of d, assuming accessing any cell takes constant time?

(c) In low dimensions, the divide-and-conquer method provides a significant speedup over naïve exhaustive search. However, in moderately high dimensions, its time complexity can grow quickly. In the high dimensional case, we modify our divide-and-conquer algorithm to use the naïve exhaustive search instead. This behavior arises in many settings, and is known as *the curse of dimensionality*. How do we overcome the curse of dimensionality? Since it arises from the need to search adjacent cells, what if we don't have cells at all?

Consider a new approach that simply projects all data points along a uniformly randomly chosen direction and keeps all projections of data points in a sorted list. To find the 1-nearest neighbor, the algorithm projects the query along the same direction used to project the data points and uses binary search to find the data point whose projection is closest to that of the query. Then it marches along the list to obtain c candidate points whose projections are the closest to the projection of the query. Finally, it performs exhaustive search over these points and returns the point that is the closest to the query. This is a simplified version of an algorithm known as Dynamic Continuous Indexing (DCI).

Because this algorithm is randomized (since it uses a randomly chosen direction), there is a non-zero probability that it returns the incorrect results. We are therefore interested in how many points we need to exhaustively search over to ensure the algorithm succeeds with high probability.

We first consider the probability that a data point that is originally far away appears closer to the query under projection than a data point that is originally close. Without loss of generality, we assume that the query is at the origin. Let $\mathbf{v}^l \in \mathbb{R}^d$ and $\mathbf{v}^s \in \mathbb{R}^d$ denote the far (long) and close (short) vectors respectively, and $\mathbf{u} \in S^{d-1} \subset \mathbb{R}^d$ is a vector drawn uniformly randomly on the unit sphere which serves as the random direction. Then the event of interest is when $\left\{|\langle \mathbf{v}^l, \mathbf{u}\rangle| \le |\langle \mathbf{v}^s, \mathbf{u}\rangle|\right\}$.

Assuming that $\mathbf{0}$, $\mathbf{v}^l$ and $\mathbf{v}^s$ are not collinear,[3] consider the plane spanned by $\mathbf{v}^l$ and $\mathbf{v}^s$, which

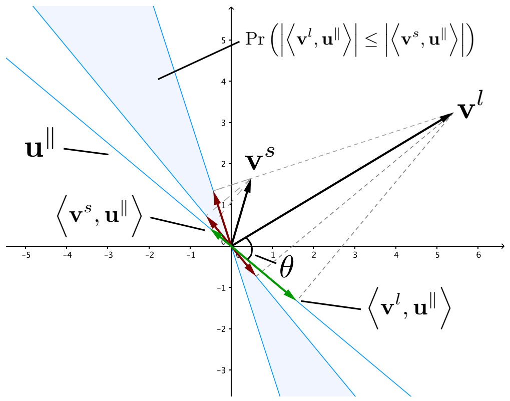
[3]If $\mathbf{v}^l$ and $\mathbf{v}^s$ are collinear, random projection will essentially always be able to tell which is which so we don't bother to analyze that case. Understanding why will help you do this problem.

we will denote as P. For any vector $\mathbf{w}$, we use $\mathbf{w}^{\parallel}$ and $\mathbf{w}^{\perp}$ to denote the components of $\mathbf{w}$ in P and $P^{\perp}$ such that $\mathbf{w} = \mathbf{w}^{\parallel} + \mathbf{w}^{\perp}$.

If we use θ denote the angle of $\mathbf{u}^{\parallel}$ relative to $\mathbf{v}^l$, show that

$$\Pr\left(|\langle \mathbf{v}^l, \mathbf{u}\rangle| \leq |\langle \mathbf{v}^s, \mathbf{u}\rangle|\right) \leq \Pr\left(|\cos(\theta)| \leq \frac{\|\mathbf{v}^s\|_2}{\|\mathbf{v}^l\|_2}\right).$$

Hint: For $\mathbf{w} \in \{\mathbf{v}^s, \mathbf{v}^l\}$, *because* $\mathbf{w}^{\perp} = 0$, $\langle \mathbf{w}, \mathbf{u}\rangle = \langle \mathbf{w}, \mathbf{u}^{\parallel}\rangle$.



$\Pr\left(\left|\left\langle \mathbf{v}^l, \mathbf{u}^{\parallel}\right\rangle\right| \leq \left|\left\langle \mathbf{v}^s, \mathbf{u}^{\parallel}\right\rangle\right|\right)$
$\mathbf{v}^l$
$\mathbf{v}^s$
$\mathbf{u}^{\parallel}$
$\left\langle \mathbf{v}^s, \mathbf{u}^{\parallel}\right\rangle$
θ
$\left\langle \mathbf{v}^l, \mathbf{u}^{\parallel}\right\rangle$

Figure 2: Examples of "good" and "bad" projection directions. The blue lines denote possible projection directions $\mathbf{u}^{\parallel}$. The isolated blue line represents a "good" projection direction, since the projection of $\mathbf{v}^l$ is longer than the projection of $\mathbf{v}^s$ (both shown in green), thereby preserving the relative order between $\mathbf{v}^l$ and $\mathbf{v}^s$ in terms of their lengths after projection. Any projection direction within the shaded region is a "bad" projection direction, since the projection of $\mathbf{v}^l$ would not be longer than the projection of $\mathbf{v}^s$, thereby inverting the relative order between $\mathbf{v}^l$ and $\mathbf{v}^s$ after projection (shown in red).

(d) The algorithm would fail to return the correct 1-nearest neighbor if more than $c - 1$ points appear closer to the query than the 1-nearest neighbor under projection.

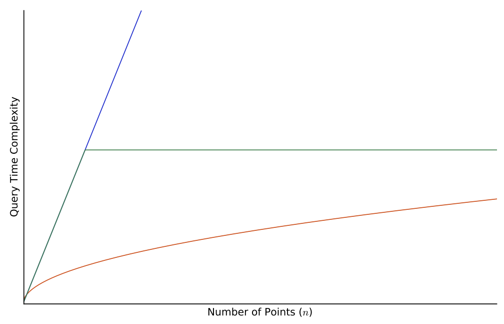
The following two statements will be useful:

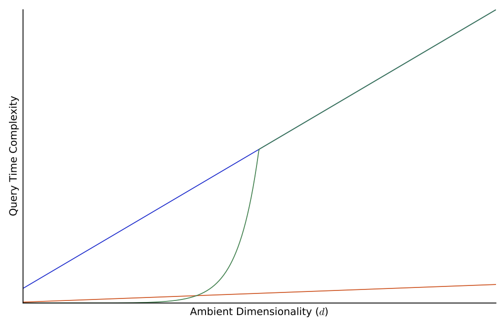
- For any set of events $\{E_i\}_{i=1}^N$, the probability that at least m of them occur is at most $\frac{1}{m}\sum_{i=1}^N \Pr(E_i)$.[4]
- $\Pr(|\cos\theta| \leq \|\mathbf{v}^s\|_2/\|\mathbf{v}^l\|_2) = 1 - \frac{2}{\pi}\cos^{-1}(\|\mathbf{v}^s\|_2/\|\mathbf{v}^l\|_2) \leq \|\mathbf{v}^s\|_2/\|\mathbf{v}^l\|_2$.

Using the first statement, derive an upper bound on the probability that the algorithm fails. Use $\mathbf{x}^{(i)}$ to denote the ith closest point to the query $\mathbf{z}$. Then use the second statement to simplify the expression.

(e) The following plots show the query time complexities of naïve exhaustive search, space partitioning, and DCI as functions of n and d. Curves of the same color correspond to the same algorithm. (Assume that the failure probability of DCI is small) **Which algorithm does each color correspond to?**

[4]This is a generalization of the union bound; the statement reduces to the union bound when $k' = 1$. (See this paper Ke Li and Jitendra Malik. Fast k-Nearest neighbor Search via Prioritized DCI. In *Proceedings of the 34th International Conference on Machine Learning*, pages 2081–2090, 2017.)


Query Time Complexity
Number of Points (n)


Query Time Complexity
Ambient Dimensionality (d)

4 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) 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 $\forall i \neq j$, $\text{Corr}(Z_i, Z_j) = \rho$. (If you don't remember the relationship between correlation and covariance from your prerequisite classes, please look it up.) Calculate the variance of the average of the random variables Z_i, written in terms of σ, ρ, and n.

What happens when n gets very large, and what does that tell us about the potential effectiveness of averaging? (…if ρ is large ($|\rho| \approx 1$)? …if ρ is very very small ($|\rho| \approx 0$)? …if ρ is middling ($|\rho| \approx 0.5$)?) We're not looking for anything too rigorous–qualitative reasoning using your derived variance is sufficient.

(c) **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? Please only consider the probability of a point not being selected in any ***one*** *of the subsamples (not all of the T subsamples).*

(ii) The number of decision trees T in the ensemble is usually chosen to trade off running time against reduced variance. (Typically, a dozen to several thousand trees are used.) The sample size n' has a smaller effect on running time, so our choice of n' is mainly governed by getting the best predictions. Although it's common practice to set $n' = n$, that isn't necessarily the best choice. How do you recommend we choose the hyperparameter n'?

5 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 additional decision tree techniques online (AdaBoost and XGBoost are particularly interesting!)

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.

5.1 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. After you are done, **attach your code in the appendix and select the appropriate pages when submitting to Gradescope.**

5.2 Implement a Random Forest

You are not allowed to use any off-the-shelf random forest implementation. However, you are allowed to now use library implementations for individual decision trees (we use sklearn in the starter code). If you use the starter code, you will mainly need to implement the superclass the random forest implementation inherits from, an implementation of bagged trees, which creates decision trees trained on different samples of the data. After you are done, **attach your code in the appendix and select the appropriate pages when submitting to Gradescope.**

5.3 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? ("No" is an acceptable response.)
5. Anything else cool you implemented? ("No" is an acceptable response.)

5.4 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).

5.5 Writeup Requirements for the Spam Dataset

1. 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) ("hot") ≥ 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.

2. 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.

5.6 Writeup Requirements for the Titanic Dataset

Train a shallow decision tree (minimum depth 3), 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–we also provide some starter code for this. If you're having too many package/environment issues, then you can also use the provided `__repr__` method to print the tree.

5.7 Test Set Predictions

Using your own classifiers, generate predictions on the test sets provided for both Spam and Titanic. You should use the `generate_submission` function provided in the starter code to ensure that your predictions are in the right format for Gradescope.

You may use any decision tree-based method that you implemented. Feel free to explore boosting methods if you wish, but these should not be required to meet the accuracy thresholds in Gradescope.

Grading for this part is as follows.

- **Titanic.** You will receive 100% if you meet 77% test set accuracy and 50% if you only meet 75% test set accuracy (no credit otherwise).
- **Spam.** You will receive 100% if you meet 80% test set accuracy and 50% if you only meet 78% test set accuracy (no credit otherwise).

You can submit to the Gradescope autograder as frequently as you wish.

6 Honor Code

1. **List all collaborators. If you worked alone, then you must explicitly state so.**
2. **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 the consequences of academic misconduct are *particularly severe*!

A Appendix

Titanic Dataset Details

Here's a brief overview of the fields in the Titanic dataset.

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)

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.fit(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. As you train your tree, your tree should create and configure subtrees to use for classification and store these internally. An instance of a **DecisionTree** class will be the root node of its resulting tree so you can directly reference its attributes and subtrees during inference time.

Each **DecisionTree** should have left and right pointers to its children, which are also trees, 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 **DecisionTree** or an entirely different class.

DecisionTree fields:

- `split_idx, thresh`: Two fields that detail 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.
- `pred`: If this field is set, the **DecisionTree** 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 prediction 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.
- `fit(data, labels)`: Grows a decision tree by constructing nodes. Using the entropy and information gain 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). 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!

B Submission Checklist

In your writeup for Question 1...

Did you include the **PDF export of the completed Colab notebook**?

In your writeup for Question 5...

1. Have you included the code for each part?
2. Have you included your generated plots and visualizations?

At the end of the writeup...

1. Have you completed all parts of the honor code (question 6)? Did you **list all collaborators** and **declare and sign** the statement?
2. Have you provided a code appendix including all code you wrote in completing the homework?

Executable Code Submission

1. Did you set seeds for all random utils? **You must ensure reproducibility of your results**.
2. Have you created an archive containing all ".py" and ".ipynb" files that you wrote or modified to generate your homework solutions? (This applies to both questions 1 and 5.)
3. Have you included your test set predictions for the **Spam** and **Titanic** datasets?
4. Have you removed all data and extraneous files from the archive?
5. 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 **HW 5 Written** and selected pages appropriately?
2. Have you submitted your executable code archive to the Gradescope assignment titled **HW 5 Code**?

Congratulations! You have completed Homework 5.