CS 189 / 289 Introduction to Machine Learning
Fall 2023 Jennifer Listgarten, Jitendra Malik

HW5

Due 11/3/23 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 Write-Up". Submit your code to the Gradescope assignment titled "HW5 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.
    - In your write-up, please state with whom you worked on the homework. This should be on its own page and should be the first page that you submit.
    - In your write-up, please copy the following statement and sign your signature underneath. If you are using LaTeX, you can type your full name underneath instead. We want to make it *extra* clear so that no one inadvertently cheats.

      "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."
    - **Replicate all of your code in an appendix**. Begin code for each coding question on 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 the appendix to correct questions.

1 Exploring Bias Variance between Ridge and Least Squares Regression

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, $\mathbf{z} = [z_1, \ldots, z_n]^\top \sim \mathcal{N}(0, \sigma^2 \mathbf{I}_n)$, and $Y = [y_1, \ldots, y_n]^\top$ 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:

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

(a) Write the solution for $\mathbf{w}_{\text{ols}}, \mathbf{w}_{\text{ridge}}$. No need to derive it

(b) Let $\hat{\mathbf{w}} \in \mathrm{R}^d$ 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 of $\mathbf{w}^*$.

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

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

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:

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

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.

(c) Show that

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

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 $\mathbf{w}^*$.

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

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

Finally, use these formulas to conclude that

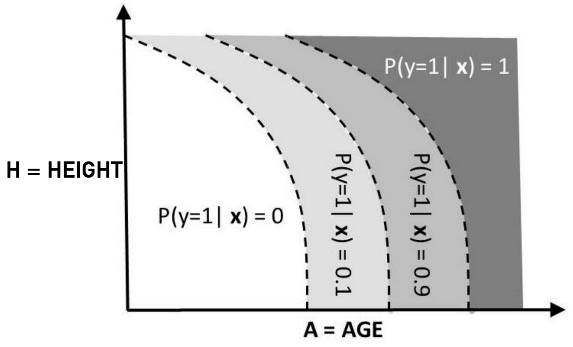
$$\operatorname{Var}(\mathbf{w}_{\text{ridge}}) < \operatorname{Var}(\mathbf{w}_{\text{ols}}) .$$

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 $\mathbf{U}(\mathbf{\Sigma} + \lambda\mathbf{I}_d)\mathbf{U}^\top$.

2 Nearest Neighbors and Bayes Rrisk

A census of Middle Earth is conducted and the data from it is used to check the extent to which we can predict if someone is an orc (class 0) or an elf (class 1) based on their age and height. We design $\mathbf{x} = [A, H]^T$ to be our feature vector, where A indicates the candidate's age and H their height. The census includes enough data that we can accurately estimate the posterior probability $P(y = 1|\mathbf{x})$ (and consequently $P(y = 0 \mid \mathbf{x}) = 1 - P(y = 1 \mid \mathbf{x})$).



We label the regions above from left to right as R_1 to R_4 respectively. For illustrative simplicity, we have made the conditional probability of being an elf piecewise constant over each of the regions (that is, $P(y = 1 \mid \mathbf{x})$ has the same value in region R_i).

(a) What is the Bayes risk in each of the four regions? Assume a 0-1 loss function.

Here, the *Bayes risk* means the probability of error of the optimum Bayes classifier that knows the underlying pattern perfectly. The Bayes risk is the counterpart of the "irreducible error" (from the bias-variance decomposition) since it reflects an intrinsic uncertainty that we cannot remove when we are trying to do prediction based on the given information.

(b) Assume that the training data is dense enough so that all the nearest neighbors of a test sample lie in the same region as the test sample itself. Now, consider a test sample $\mathbf{x}$ which falls in region R_i. For $i \in 1, 2, 3, 4$, what is the probability that $\mathbf{x}$ and its nearest neighbor have different labels y?

Here, we are assuming that the training data has labels generated by the same random process that generates the labels at test time, and that the labels on training points are independent of the labels on test points.

(c) What is the 1-nearest neighbor classification error rate in each region?

(d) Now we generalize the problem above.

$$R1 : P(y = 1|x) = 0$$
$$R2 : P(y = 1|x) = p$$
$$R3 : P(y = 1|x) = 1 - p$$
$$R4 : P(y = 1|x) = 1$$

where p is a probability between 0 and 0.5.

Calculate the answers to the previous questions under this generalization. Can you see why the classification performance of 1-nearest neighbor with sufficiently dense training data is never worse than twice the Bayes risk?

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[1]. **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-

[1]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.

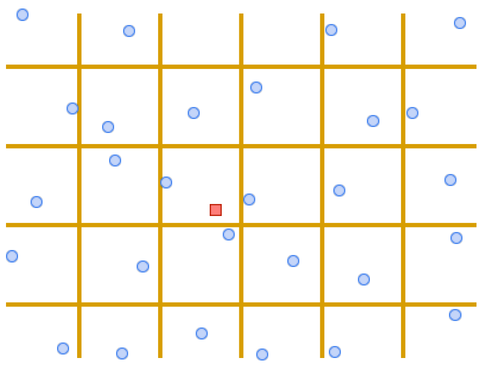


Figure 1: Illustration of the space partitioning scheme we consider. The data points are shown as blue circles and the query is shown as the red square. The cell boundaries are shown as gold lines.

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 $\tilde{k}$ 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 $\left\{ |\langle \mathbf{v}^l, \mathbf{u} \rangle| \leq |\langle \mathbf{v}^s, \mathbf{u} \rangle| \right\}$.

Assuming that $\mathbf{0}$, $\mathbf{v}^l$ and $\mathbf{v}^s$ are not collinear,[2] consider the plane spanned by $\mathbf{v}^l$ and $\mathbf{v}^s$, which

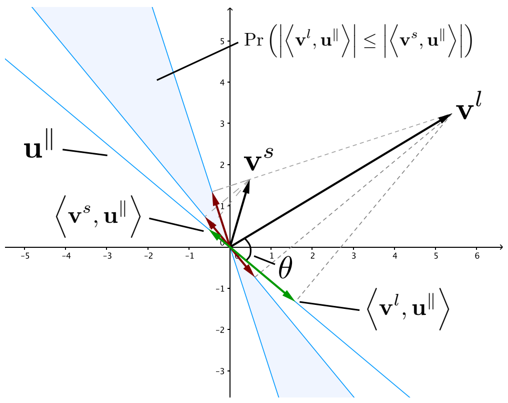
[2]If $\mathbf{v}^l$ and $\mathbf{v}^s$ are collinear, random projection will essentially always be able to tell which is which so we don't bother to analyze that case. Understanding why will help you do this problem.

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

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

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



$\Pr\left(\left|\left\langle \mathbf{v}^l, \mathbf{u}^{\parallel}\right\rangle\right| \le \left|\left\langle \mathbf{v}^s, \mathbf{u}^{\parallel}\right\rangle\right|\right)$
$\mathbf{v}^l$
$\mathbf{u}^{\parallel}$
$\mathbf{v}^s$
$\left\langle \mathbf{v}^s, \mathbf{u}^{\parallel}\right\rangle$
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 $\tilde{k} - 1$ points appear closer to the query than the 1-nearest neighbor under projection.

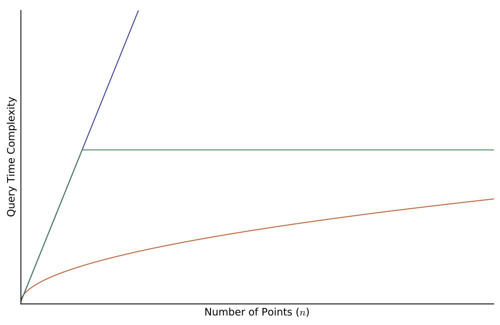
The following two statements will be useful:

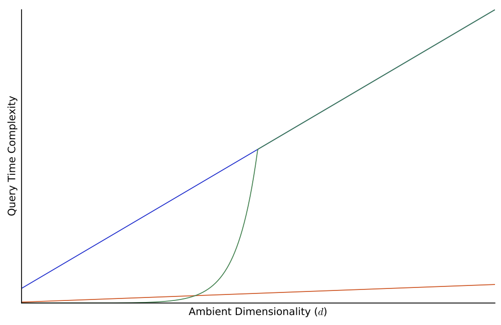
- For any set of events $\{E_i\}_{i=1}^{N}$, the probability that at least m of them occur is at most $\frac{1}{m}\sum_{i=1}^{N} \Pr(E_i)$.[3]
- $\Pr\left(|\cos\theta| \le \|\mathbf{v}^s\|_2 / \|\mathbf{v}^l\|_2\right) = 1 - \frac{2}{\pi}\cos^{-1}\left(\|\mathbf{v}^s\|_2 / \|\mathbf{v}^l\|_2\right)$.

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?**

[3]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 Decision Trees

Consider constructing a decision tree on data with d features and n training points where each feature is real-valued and each label takes one of m possible values. The splits are two-way, and are chosen to maximize the information gain. We only consider splits that form a linear boundary parallel to one of the axes. We will only consider a standalone decision tree and not a random forest (hence no randomization). Recall the definition of information gain:

$$IG(\mathbf{node}) = H(S) - \frac{|S_l|H(S_l) + |S_r|H(S_r)}{|S_l| + |S_r|}$$

where S is set of samples considered at **node**, S_l is the set of samples remaining in the left sub-tree after **node**, S_r is the set of samples remaining in the right sub-tree after **node**, and $H(S)$ is the entropy over a set of samples:

$$H(S) = -\sum_{i=1}^{C} p_i \log(p_i)$$

Here, C is the number of classes, and p_i is the proportion of samples in S labeled as class i.

(a) Prove or give a counter-example: In any path from the root to a leaf, the same feature will never be split on twice.

(b) Prove or give a counter-example: The information gain at the root is at least as much as the information gain at any other node.
Hint: Think about the XOR function.

(c) Suppose that a learning algorithm is trying to find a consistent hypothesis when the labels are actually being generated randomly. There are d Boolean features and 1 Boolean label, and n examples are drawn uniformly from the set of 2^{d+1} possible examples *with* replacement. Calculate the probability of finding a contradiction in the sampled data. For ease of computation, you don't have to consider the case where identical samples (samples with the same features and same label) are drawn from the distribution.
Hint: a contradiction is reached if two samples with identical features but different labels are drawn.

(d) Intuitively, how does the bias-variance trade-off relate to the depth of a decision tree?

5 Decision Trees and Random Forests

In this problem, you will implement decision trees and random forests for classification on two datasets:

1. Titanic Dataset: predict if a person survived the sinking of the Titanic
2. Spam Dataset: predict if a message is spam

In lecture, you were given a basic introduction to decision trees and how such trees are learned from training data. You were also introduced to random forests. Feel free to research different decision-tree training techniques online.

(a) **Implement the *information gain* (i.e., entropy of the parent node minus the weighted sum of entropy of the child nodes) splitting rule for greedy decision tree learning. Include your code below.**

See `decision_tree_starter.py` for the recommended starter code. It contains a simplified decision tree implementation that splits only on one feature at a time, using the functions you implement.

Note: The sample implementation assumes that all features are continuous. You may convert all your features to be continuous or augment the implementation to handle discrete features.

Note: You should NOT use any software package other than NumPy for part a.

(b) Before applying the decision-tree learning algorithm to the Titanic dataset, we will first pre-process the dataset. In the real-world, pre-processing the data is a very important step since real-life data can be quite imperfect. However, to make this problem easier, we have provided some code to preprocess the data. Look at the code and **describe how we deal with the following problems**:

- Some data points are misssing class labels;
- Some features are not numerical values;
- Some data points are missing some features.

Now **train a shallow decision tree on the Titanic dataset.** (for example, a depth 3 tree, although you may choose any depth that looks good).

Data Processing for Titanic Here is a brief overview of the fields in the Titanic dataset.

(a) survived - 1 is survived; 0 is not. This is the class label.
(b) pclass - Measure of socioeconomic status: 1 is upper, 2 is middle, 3 is lower.
(c) sex - Male/Female
(d) age - Fractional if less than 1.
(e) sibsp - Number of siblings/spouses aboard the Titanic

(f) parch - Number of parents/children aboard the Titanic

(g) ticket - Ticket number

(h) fare - Fare.

(i) cabin - Cabin number.

(j) embarked - Port of Embarkation (C = Cherbourg, Q = Queenstown, S = Southampton)

(c) **Use `sklearn` to train a shallow decision tree on the Titanic and spam datasets.** (for example, a depth 3 tree, although you may choose any depth that looks good). **Include your code for it below.** Additionally, **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.

We provide you a code snippet to draw the tree using `pydot` and `graphviz`. If it is hard for you to install these dependencies, you need to draw the diagram by hand. If you need draw the diagram by hand, you may use the following function, added to the `DecisionTree` class for visualization:

```
    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__())
```

(d) From this point forward, you are allowed to use `sklearn.tree.*` and the classes we have imported for you below in the starter code snippets. You are NOT allowed to use other functions from `sklearn`. **Implement bagged trees as follows:** for each tree up to *n*, sample *with replacement* from the original training set until you have as many samples as the training set. Fit a decision tree for each sampling. **Include the code for your implementation of bagged trees below.** Here is some optional starter code:

```
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.base import BaseEstimator, ClassifierMixin


class BaggedTrees(BaseEstimator, ClassifierMixin):

    def __init__(self, params=None, n=200):
        if params is None:
            params = {}
        self.params = params
        self.n = n
        self.decision_trees = [
            DecisionTreeClassifier(random_state=i,
                                   **self.params) for i in
            range(self.n)]

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

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

(e) **Apply bagged trees to the Titanic and spam datasets. Find and state the most common splits made at the root node of the different trees.** For example:

  (a) ("thanks") < 4 (15 trees)
  (b) ("nigeria") ≥ 1 (5 trees)

Data format for Spam The preprocessed spam dataset given to you as part of the homework in `spam_data.mat` consists of 11,029 email messages, from which 32 features have been extracted as follows:

- 25 features giving the frequency (count) of words in a given message which match the following words: 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.
- 7 features giving the frequency (count) of characters in the email that match the following characters: `;`, `$`, `#`, `!`, `(`, `[`, `&`.

The dataset consists of a training set size 5172 and a test set of size 5857.

(f) **Implement random forests as follows:** again, for each tree in the forest, sample *with replacement* from the original training set until you have as many samples as the training set. Learn a decision tree for each sample, this time using a randomly sampled subset of the features (instead of the full set of features) to find the best split on the data. Let `m` denote the number of features to subsample. **Include the code for your implementation of random forests below.** Here is some optional starter code:

```
class RandomForest(BaggedTrees):

    def __init__(self, params=None, n=200, m=1):
        if params is None:
            params = {}
        # TODO implement function
        pass
```

(g) **Apply bagged random forests to the Titanic and spam datasets. Find and state the most common splits made at the root node of the different trees.**

(h) **Summarize the performance evaluation of: a single decision tree, bagged trees, and random forests**. For each of the 2 datasets, report your training and validation accuracies. You should use a 3-fold cross validation, i.e., splitting the dataset into 3 parts. You should be reporting 24 numbers (2 datasets $\times$ 3 classifiers $\times$ (3 + 1) for 3 cross validation accuracies (1 per split) and 1 training accuracy (for the whole dataset)). Describe qualitatively which types of trees and forests performed best. Detail any parameters that worked well for you. **In addition, for each of the** 2 **datasets, train your best model and submit your predictions on the test data to Gradescope**. Your best Titanic classifier should exceed 73% accuracy and your best Spam classifier should exceed 76% accuracy for full points.

(i) You should submit

- your answers, plots, and code as part of your homework PDF write-up
- a .zip file as part of the homework code submission, containing (a) your code, (b) a file, named `predictions_titanic.csv`, of your Titanic test predictions, and (c) a file, named `predictions_spam.csv`, of your spam test predictions.