1 k-Means Demo

Work through the entire ***Colab notebook***.

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

Solution: See Apendix

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

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

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

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

(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.

Solution:

1. Expanding the MSE Definition:

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

2. Decomposing $\hat{\mathbf{w}}$ Around its Expected Value:

Let $\boldsymbol{\mu} := \mathbb{E}[\hat{\mathbf{w}}]$ represent the expected value of the estimator $\hat{\mathbf{w}}$. We can decompose $\hat{\mathbf{w}}$ as:

$$\hat{\mathbf{w}} - \mathbf{w}^* = (\hat{\mathbf{w}} - \boldsymbol{\mu}) + (\boldsymbol{\mu} - \mathbf{w}^*).$$

3. Expanding the Squared Norm Using This Decomposition:

By applying the identity $\|\mathbf{a} + \mathbf{b}\|_2^2 = \|\mathbf{a}\|_2^2 + \|\mathbf{b}\|_2^2 + 2\mathbf{a}^\top \mathbf{b}$, we get:

$$\|\hat{\mathbf{w}} - \mathbf{w}^*\|_2^2 = \|\hat{\mathbf{w}} - \boldsymbol{\mu}\|_2^2 + \|\boldsymbol{\mu} - \mathbf{w}^*\|_2^2 + 2(\hat{\mathbf{w}} - \boldsymbol{\mu})^\top (\boldsymbol{\mu} - \mathbf{w}^*).$$

4. Taking the Expectation:

The expectation of the cross term $2(\hat{\mathbf{w}} - \boldsymbol{\mu})^\top(\boldsymbol{\mu} - \mathbf{w}^*)$ is zero because $\hat{\mathbf{w}} - \boldsymbol{\mu}$ has mean zero (it is centered around $\boldsymbol{\mu}$). Therefore:

$$\mathbb{E}\left[\|\hat{\mathbf{w}} - \mathbf{w}^*\|_2^2\right] = \mathbb{E}\left[\|\hat{\mathbf{w}} - \boldsymbol{\mu}\|_2^2\right] + \|\boldsymbol{\mu} - \mathbf{w}^*\|_2^2.$$

5. Identifying the Bias and Variance Terms:

The term $\|\boldsymbol{\mu} - \mathbf{w}^*\|_2^2$ represents the **squared bias** of the estimator $\hat{\mathbf{w}}$, as it is the squared distance between the expected estimator $\boldsymbol{\mu}$ and the true parameter $\mathbf{w}^*$. The term $\mathbb{E}\left[\|\hat{\mathbf{w}} - \boldsymbol{\mu}\|_2^2\right]$ represents the **variance** of the estimator $\hat{\mathbf{w}}$.

6. Expressing the Variance Using the Covariance Matrix:

The variance term can be expressed as the trace of the covariance matrix of $\hat{\mathbf{w}}$:

$$\mathbb{E}\left[\|\hat{\mathbf{w}} - \boldsymbol{\mu}\|_2^2\right] = \mathrm{Tr}(\mathrm{Cov}(\hat{\mathbf{w}})).$$

7. Final MSE Decomposition:

Combining these, we get:

$$\mathrm{MSE}(\hat{\mathbf{w}}) = \|\boldsymbol{\mu} - \mathbf{w}^*\|_2^2 + \mathrm{Tr}(\mathrm{Cov}(\hat{\mathbf{w}})).$$

Thus,

$$\mathrm{MSE}(\hat{\mathbf{w}}) = \mathrm{Bias}(\hat{\mathbf{w}})^2 + \mathrm{Var}(\hat{\mathbf{w}}),$$

where: $\mathrm{Bias}(\hat{\mathbf{w}})^2 = \|\boldsymbol{\mu} - \mathbf{w}^*\|_2^2$, $\mathrm{Var}(\hat{\mathbf{w}}) = \mathrm{Tr}(\mathrm{Cov}(\hat{\mathbf{w}}))$.

(b) 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 $\mathbf{w}^*$.

Solution:

Step 1: Expectation of $\mathbf{w}_{\text{ols}}$

Substitute $\mathbf{y} = \mathbf{X}\mathbf{w}^* + \mathbf{z}$ into the OLS estimator: $\mathbf{w}_{\text{ols}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top(\mathbf{X}\mathbf{w}^* + \mathbf{z})$.

Expanding this, we get $\mathbf{w}_{\text{ols}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{X}\,\mathbf{w}^* + (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{z}$.

Now, take the expectation of $\mathbf{w}_{\text{ols}}$: $\mathbb{E}[\mathbf{w}_{\text{ols}}] = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{X}\,\mathbf{w}^* + (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbb{E}[\mathbf{z}]$.

Since $\mathbb{E}[\mathbf{z}] = \mathbf{0}$, we have $\mathbb{E}[\mathbf{w}_{\text{ols}}] = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{X}\,\mathbf{w}^* = \mathbf{w}^*$.

Thus, $\mathbf{w}_{\text{ols}}$ is an unbiased estimator of $\mathbf{w}^*$.

Step 2: Expectation of $\mathbf{w}_{\text{ridge}}$

Now, consider the ridge estimator: $\mathbf{w}_{\text{ridge}} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_d)^{-1}\mathbf{X}^\top\mathbf{y}$.

Substitute $\mathbf{y} = \mathbf{X}\mathbf{w}^* + \mathbf{z}$: $\mathbf{w}_{\text{ridge}} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_d)^{-1}\mathbf{X}^\top(\mathbf{X}\mathbf{w}^* + \mathbf{z})$.

Expanding this, we get $\mathbf{w}_{\text{ridge}} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_d)^{-1}\mathbf{X}^\top\mathbf{X}\,\mathbf{w}^* + (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_d)^{-1}\mathbf{X}^\top\mathbf{z}$.

Take the expectation of $\mathbf{w}_{\text{ridge}}$: $\mathbb{E}[\mathbf{w}_{\text{ridge}}] = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_d)^{-1}\mathbf{X}^\top\mathbf{X}\,\mathbf{w}^* + (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_d)^{-1}\mathbf{X}^\top\mathbb{E}[\mathbf{z}]$.

Since $\mathbb{E}[\mathbf{z}] = \mathbf{0}$, the expectation simplifies to $\mathbb{E}[\mathbf{w}_{\text{ridge}}] = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_d)^{-1}\mathbf{X}^\top\mathbf{X}\,\mathbf{w}^*$.

This expression shows that $\mathbb{E}[\mathbf{w}_{\text{ridge}}] \neq \mathbf{w}^*$ unless $\lambda = 0$. Thus, $\mathbf{w}_{\text{ridge}}$ is a biased estimator of $\mathbf{w}^*$.

Conclusion

$\mathbf{w}_{\text{ols}}$ is an unbiased estimator of $\mathbf{w}^*$ because $\mathbb{E}[\mathbf{w}_{\text{ols}}] = \mathbf{w}^*$.

$\mathbf{w}_{\text{ridge}}$ is a biased estimator of $\mathbf{w}^*$ because $\mathbb{E}[\mathbf{w}_{\text{ridge}}] = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_d)^{-1}\mathbf{X}^\top\mathbf{X}\,\mathbf{w}^*$, which depends on λ.

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

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

Finally, use these formulas to conclude that

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

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

Solution:

Solution to Part (c)

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

Step 1: Trace of Cov($\mathbf{w}_{\text{ols}}$)

The OLS estimator for $\mathbf{w}$ is $\mathbf{w}_{\text{ols}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$.

The covariance of $\mathbf{w}_{\text{ols}}$ is given by $\mathrm{Cov}(\mathbf{w}_{\text{ols}}) = \sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}$.

To find the trace of $\mathrm{Cov}(\mathbf{w}_{\text{ols}})$, we use the fact that the trace of a matrix is the sum of its eigenvalues. If $\{\gamma_i\}_{i=1}^d$ are the eigenvalues of $\mathbf{X}^\top\mathbf{X}$, then the eigenvalues of $(\mathbf{X}^\top\mathbf{X})^{-1}$ are $\left\{\frac{1}{\gamma_i}\right\}_{i=1}^d$. Therefore, $\mathrm{Tr}(\mathrm{Cov}(\mathbf{w}_{\text{ols}})) = \mathrm{Tr}\left(\sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}\right) = \sigma^2\sum_{i=1}^d \frac{1}{\gamma_i}$.

Step 2: Trace of Cov($\mathbf{w}_{\text{ridge}}$)

The Ridge estimator for $\mathbf{w}$ is $\mathbf{w}_{\text{ridge}} = (\mathbf{X}^\top\mathbf{X}+\lambda\mathbf{I}_d)^{-1}\mathbf{X}^\top\mathbf{y}$.

The covariance of $\mathbf{w}_{\text{ridge}}$ is $\mathrm{Cov}(\mathbf{w}_{\text{ridge}}) = \sigma^2(\mathbf{X}^\top\mathbf{X}+\lambda\mathbf{I}_d)^{-1}\mathbf{X}^\top\mathbf{X}(\mathbf{X}^\top\mathbf{X}+\lambda\mathbf{I}_d)^{-1}$.

To simplify, let's consider the eigendecomposition of $\mathbf{X}^\top\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{U}^\top$, where $\mathbf{\Sigma} = \mathrm{diag}(\gamma_1, \gamma_2, \ldots, \gamma_d)$ and $\mathbf{U}$ is an orthogonal matrix of eigenvectors.

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

The inverse is then $(\mathbf{X}^\top\mathbf{X}+\lambda\mathbf{I}_d)^{-1} = \mathbf{U}(\mathbf{\Sigma}+\lambda\mathbf{I}_d)^{-1}\mathbf{U}^\top$.

The covariance matrix $\mathrm{Cov}(\mathbf{w}_{\text{ridge}})$ can now be expressed as $\mathrm{Cov}(\mathbf{w}_{\text{ridge}}) = \sigma^2\mathbf{U}(\mathbf{\Sigma}+\lambda\mathbf{I}_d)^{-1}\mathbf{\Sigma}(\mathbf{\Sigma}+\lambda\mathbf{I}_d)^{-1}\mathbf{U}^\top$.

The eigenvalues of $\mathrm{Cov}(\mathbf{w}_{\text{ridge}})$ are thus $\frac{\sigma^2\gamma_i}{(\gamma_i+\lambda)^2}$, for $i = 1, \ldots, d$. Therefore, $\mathrm{Tr}(\mathrm{Cov}(\mathbf{w}_{\text{ridge}})) = \sigma^2\sum_{i=1}^d \frac{\gamma_i}{(\gamma_i+\lambda)^2}$.

Step 3: Comparing Var($\mathbf{w}_{\text{ridge}}$) and Var($\mathbf{w}_{\text{ols}}$)

Now we can compare $\mathrm{Tr}(\mathrm{Cov}(\mathbf{w}_{\text{ols}}))$ and $\mathrm{Tr}(\mathrm{Cov}(\mathbf{w}_{\text{ridge}}))$.

1. For $\mathrm{Tr}(\mathrm{Cov}(\mathbf{w}_{\text{ols}}))$, we have $\mathrm{Tr}(\mathrm{Cov}(\mathbf{w}_{\text{ols}})) = \sigma^2\sum_{i=1}^d \frac{1}{\gamma_i}$.

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

Since $\frac{\gamma_i}{(\gamma_i+\lambda)^2} < \frac{1}{\gamma_i}$ for each i (because $\lambda > 0$), it follows that $\mathrm{Tr}(\mathrm{Cov}(\mathbf{w}_{\text{ridge}})) < \mathrm{Tr}(\mathrm{Cov}(\mathbf{w}_{\text{ols}}))$.

Thus, we conclude that $\mathrm{Var}(\mathbf{w}_{\text{ridge}}) < \mathrm{Var}(\mathbf{w}_{\text{ols}})$.

This result shows that the Ridge estimator has lower variance than the OLS estimator, which is the opposite of the relationship between their biases.

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?**

Solution:

Step 1: Computing Distances

Given a query point $\mathbf{z} \in \mathbb{R}^d$, we need to compute the Euclidean distance from $\mathbf{z}$ to each of the n points in the dataset $D = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$.

Each distance computation involves calculating the difference between the coordinates of $\mathbf{z}$ and a point $\mathbf{x}_i$, squaring each component, summing them up, and taking the square root. The complexity of computing the distance from $\mathbf{z}$ to a single point in D is $\mathcal{O}(d)$, as there are d components to process. Since we need to compute the distance for each of the n points, the total complexity for this step is $\mathcal{O}(nd)$.

Step 2: Finding the k Nearest Neighbors

Once we have all n distances, we need to identify the k points with the shortest distances

Using the Quickselect algorithm to find the k-smallest elements has an average-case time complexity of $\mathcal{O}(n)$. This complexity holds because Quickselect partitions the distances in linear time, focusing only on finding the k-smallest distances without fully sorting the list.

Step 3: Total Complexity

Combining both steps, the total average-case time complexity for a single query in this naive exhaustive search method is:

$$\mathcal{O}(nd) + \mathcal{O}(n) = \mathcal{O}(nd).$$

[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.

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

Solution:

Step 1: One-dimensional Case

In one dimension, the feature space is divided into intervals (cells) along a line. When searching for the 1-nearest neighbor of a query point $\mathbf{z}$, we need to check the cell containing $\mathbf{z}$ and the adjacent cells on either side, in case $\mathbf{z}$ is close to the boundary of its cell.

Number of cells to search: 3 (the cell containing $\mathbf{z}$ and its two neighbors).

Step 2: Two-dimensional Case

In two dimensions, the feature space is divided into a grid of squares. For a query point $\mathbf{z}$, we need to check the cell containing $\mathbf{z}$ and its adjacent cells, which includes the cells to the left, right, above, below, and diagonal neighbors.

Number of cells to search: $3 \times 3 = 9$ cells (the cell containing $\mathbf{z}$ and its 8 adjacent cells).

Step 3: d-dimensional Case

In d-dimensions, the feature space is divided into a grid of d-dimensional hypercubes. For a query point $\mathbf{z}$, we need to check the cell containing $\mathbf{z}$ and all its adjacent cells along each dimension.

Number of cells to search: In d-dimensions, each cell has 3^d possible configurations, because along each dimension we consider the cell containing $\mathbf{z}$ and the two adjacent cells (one on each side).

$$\text{Total cells to search} = 3^d.$$

Final Answer

Thus, the time complexity for finding the 1-nearest neighbor in a grid-based partitioning scheme, assuming constant-time access to each cell, is $\mathcal{O}(3^d)$ with respect to the dimensionality d.

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

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 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

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

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

Solution:

Step 1: Decomposing w Relative to $\mathbf{v}^l$

To analyze this probability, consider the 2-dimensional plane P spanned by $\mathbf{v}^l$ and $\mathbf{v}^s$. We can decompose any vector $\mathbf{w}$ in P into components parallel and orthogonal to $\mathbf{v}^l$: $\mathbf{w} = \mathbf{w}^\parallel + \mathbf{w}^\perp$, where: - $\mathbf{w}^\parallel$ is the component of $\mathbf{w}$ in the direction of $\mathbf{v}^l$, - $\mathbf{w}^\perp$ is orthogonal to $\mathbf{v}^l$ within the plane P.

Step 2: Projecting onto u and Analyzing the Probability

Let θ denote the angle between $\mathbf{w}^\parallel$ and $\mathbf{v}^l$. When we project $\mathbf{w}$ onto the random direction $\mathbf{u}$, the length of the projection depends on the angle θ between $\mathbf{u}$ and $\mathbf{v}^l$.

Since $\mathbf{u}$ is chosen randomly on the unit sphere, $\cos(\theta)$ (the cosine of the angle between $\mathbf{u}$ and $\mathbf{v}^l$) is uniformly distributed between -1 and 1.

The probability we want is $\Pr\left(\left|\langle \mathbf{v}^l, \mathbf{u}\rangle\right| \leq |\langle \mathbf{v}^s, \mathbf{u}\rangle|\right)$.

Using the definition of the inner product, this becomes:

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

[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.

Step 3: Expressing the Probability in Terms of $\cos(\theta)$

Rearranging the inequality, we get:

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

Since $\cos(\theta)$ is uniformly distributed between -1 and 1, the probability that $|\cos(\theta)|$ falls within this range is given by twice the length of the interval $\left[0, \frac{\|\mathbf{v}^s\|_2}{\|\mathbf{v}^l\|_2}\right]$:

$$\Pr\left(|\cos(\theta)| \leq \frac{\|\mathbf{v}^s\|_2}{\|\mathbf{v}^l\|_2}\right) = 2 \cdot \frac{\|\mathbf{v}^s\|_2}{\|\mathbf{v}^l\|_2}.$$

(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 $\frac{1}{m}\sum_{i=1}^N \Pr(E_i)$.[3]
- $\Pr(|\cos\theta| \le \|\mathbf{v}^s\|_2/\|\mathbf{v}^l\|_2) = 1 - \frac{2}{\pi}\cos^{-1}(\|\mathbf{v}^s\|_2/\|\mathbf{v}^l\|_2) \le \|\mathbf{v}^s\|_2/\|\mathbf{v}^l\|_2$.

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.

Solution:

Let: $\mathbf{x}^{(1)}$ denote the true 1-nearest neighbor to the query $\mathbf{z}$.

-$\mathbf{x}^{(i)}$ for $i = 2, \ldots, n$ denote other points in the dataset, ordered by increasing distance from $\mathbf{z}$.

$E_i :=$ {The projection of $\mathbf{x}^{(i)}$ is closer to the projection of $\mathbf{z}$ than $\mathbf{x}^{(1)}$ under projection}.

To bound the probability that more than $c-1$ points are closer to the query than the true 1-nearest neighbor, we will use the union bound and the given statements.

Step 1: Applying the Union Bound

Using the first statement, we know that the probability that at least c of the events $\{E_i\}_{i=2}^n$ occur is at most:

$$\Pr\left(\bigcup_{i=1}^n E_i\right) \le \frac{1}{c}\sum_{i=2}^n \Pr(E_i).$$

Step 2: Bounding $\Pr(E_i)$ using the Second Statement

From the second statement, we have:

$$\Pr\left(|\cos\theta| \le \frac{\|\mathbf{v}^s\|_2}{\|\mathbf{v}^l\|_2}\right) \le \frac{\|\mathbf{v}^s\|_2}{\|\mathbf{v}^l\|_2},$$

where $\|\mathbf{v}^s\|$ and $\|\mathbf{v}^l\|$ denote the distances from the query to the "short" (near) and "long" (far) points respectively.

In our context, for each event E_i, $\mathbf{v}^s$ would correspond to $\mathbf{x}^{(1)}$ (the nearest neighbor), and $\mathbf{v}^l$ corresponds to $\mathbf{x}^{(i)}$. Thus, we can bound $\Pr(E_i)$ as:

$$\Pr(E_i) \le \frac{\|\mathbf{x}^{(1)}\|_2}{\|\mathbf{x}^{(i)}\|_2}.$$

Step 3: Combining the Results

Using the bound on $\Pr(E_i)$, we get:

$$\sum_{i=2}^n \Pr(E_i) \le \sum_{i=2}^n \frac{\|\mathbf{x}^{(1)}\|_2}{\|\mathbf{x}^{(i)}\|_2}.$$

Therefore, the probability that the algorithm fails (i.e., at least c points appear closer under projection) is at most:

$$\Pr(\text{Algorithm fails}) \le \frac{1}{c}\sum_{i=2}^n \frac{\|\mathbf{x}^{(1)}\|_2}{\|\mathbf{x}^{(i)}\|_2}.$$

[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.)

(e) The following plots (see the homework PDF) 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?**

Solution:

Blue Line - Naive Exhaustive Search: The blue line represents the naive exhaustive search algorithm, which calculates the distance from a query point to each point in the dataset. This results in a time complexity of $\mathcal{O}(nd)$, where n is the dataset size and d is the dimensionality of each point.

In the plot of complexity as a function of n, this linear relationship with n creates a steep, direct upward trend in the blue line, indicating that as the dataset grows, the query time scales linearly. In the second plot (complexity as a function of d), we see a similarly linear trend due to the direct dependence on d. Because naive search does not use any optimization or space partitioning, it scales linearly with both n and d, resulting in consistently steep slopes across both plots.

Green Line - Space Partitioning (e.g., KD-trees): The green line corresponds to space partitioning algorithms like KD-trees, which divide the data space into smaller regions, optimizing query time particularly in low-dimensional settings.

In the plot of complexity as a function of n, space partitioning initially achieves sublinear complexity, represented by a flat or gently increasing green line when d is low, indicating efficient querying. However, as d rises, space partitioning suffers from the "curse of dimensionality," causing its complexity to approach $\mathcal{O}(nd)$ and the green line to grow more steeply with n. In the plot of complexity as a function of d, space partitioning is highly effective in low d, but as dimensionality increases, the performance deteriorates sharply, leading to an exponential rise as d grows. This pattern explains why the green line is initially efficient but becomes steeper and eventually similar to the blue line in both plots as d increases.

Orange Line - Dynamic Continuous Indexing (DCI): The orange line represents Dynamic Continuous Indexing (DCI), designed for efficiency in high-dimensional data. DCI uses random projections to reduce the effective dimensionality of the dataset, which enables sublinear scaling in n and helps maintain low complexity as both n and d increase.

In the plot of complexity as a function of n, DCI's use of projections results in a relatively stable and low growth in query time, so the orange line remains flat and lower than the others, making it especially suitable for large datasets. In the plot of complexity as a function of d, DCI is less sensitive to increases in dimensionality compared to space partitioning, resulting in a relatively stable curve across d values. This stability across both plots illustrates why DCI is well-suited for handling high-dimensional, large datasets, with the orange line exhibiting the least steep and most consistent trend in both dimensions.

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.)

Solution:).

Step 1: Calculating the Expectation of $\bar{Y}$

Since expectation is a linear operator, we can calculate $\mathbb{E}[\bar{Y}]$ as follows:

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

Since $\mathbb{E}[Y_i] = \mu$ for all i, we have:

$$\mathbb{E}[\bar{Y}] = \frac{1}{n}\sum_{i=1}^n \mu = \frac{n\mu}{n} = \mu.$$

Thus, the expectation of the average $\bar{Y}$ is $\mathbb{E}[\bar{Y}] = \mu$.

Step 2: Calculating the Variance of $\bar{Y}$

Since Y_i are uncorrelated, the variance of the sum is the sum of the variances. Therefore, we can calculate $\text{Var}(\bar{Y})$ as follows:

$$\text{Var}(\bar{Y}) = \text{Var}\left(\frac{1}{n}\sum_{i=1}^n Y_i\right) = \frac{1}{n^2}\sum_{i=1}^n \text{Var}(Y_i).$$

Since $\text{Var}(Y_i) = \sigma^2$ for all i, we have:

$$\text{Var}(\bar{Y}) = \frac{1}{n^2}\sum_{i=1}^n \sigma^2 = \frac{n\sigma^2}{n^2} = \frac{\sigma^2}{n}.$$

Thus, the variance of the average $\bar{Y}$ is $\text{Var}(\bar{Y}) = \frac{\sigma^2}{n}$.

(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.

Solution:

Step 1: Calculate the Variance of the Average

Define the average of these random variables as $\bar{Z} = \frac{1}{n}\sum_{i=1}^n Z_i$.

We want to calculate $\text{Var}(\bar{Z})$. First, expand $\bar{Z}$ as follows:

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

Using the formula for the variance of a sum of random variables, we get

$$\text{Var}\left(\sum_{i=1}^n Z_i\right) = \sum_{i=1}^n \text{Var}(Z_i) + \sum_{i\neq j} \text{Cov}(Z_i, Z_j).$$

Since $\text{Var}(Z_i) = \sigma^2$ for all i, and $\text{Cov}(Z_i, Z_j) = \rho\sigma^2$ for $i \neq j$, we can substitute these values:

$$\text{Var}\left(\sum_{i=1}^n Z_i\right) = n\sigma^2 + \sum_{i\neq j} \rho\sigma^2.$$

There are $n(n-1)$ pairs (i, j) where $i \neq j$, so we get

$$\text{Var}\left(\sum_{i=1}^n Z_i\right) = n\sigma^2 + n(n-1)\rho\sigma^2 = \sigma^2\left(n + n(n-1)\rho\right).$$

Now, substituting back, we find

$$\text{Var}(\bar{Z}) = \frac{1}{n^2}\sigma^2\left(n + n(n-1)\rho\right) = \frac{\sigma^2}{n}\left(1 + (n-1)\rho\right).$$

Step 2: Analyze the Behavior as n Gets Large

The variance of the average $\bar{Z}$ is

$$\text{Var}(\bar{Z}) = \frac{\sigma^2}{n}\left(1 + (n-1)\rho\right).$$

Now, consider how this behaves for different values of ρ:

1. If ρ is large ($|\rho| \approx 1$). When ρ is close to 1, the variance does not decrease significantly as n increases, because the term $(n-1)\rho$ dominates. In this case, averaging does not effectively reduce the variance, as the variables are highly correlated and do not provide much independent information.

2. If ρ is very small ($|\rho| \approx 0$). When ρ is close to 0, the variance becomes $\frac{\sigma^2}{n}$, which decreases as n increases. This is the ideal case for averaging, as uncorrelated variables contribute to a significant reduction in variance.

3. If ρ is middling ($|\rho| \approx 0.5$). For intermediate values of ρ, the variance reduction is partial. As n grows, the $(n-1)\rho$ term contributes moderately to the variance, so averaging provides some reduction in variance, but not as effectively as in the uncorrelated case.

(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).*

Solution:

Step 1: Calculation of the Probability of Not Being Selected

Consider a single data point in the dataset. The probability that this particular data point is **not selected** in one draw is:

$$1 - \frac{1}{n}.$$

Since we are sampling with replacement and creating a subsample of size n, there are n draws in total. The probability that this data point is not selected in any of the n draws is:

$$\left(1 - \frac{1}{n}\right)^n.$$

Step 2: Taking the Limit as $n \to \infty$

As n becomes large, we can approximate $\left(1 - \frac{1}{n}\right)^n$ using the limit:

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

Since $e^{-1} \approx 0.3679$, we find that:

$$\left(1 - \frac{1}{n}\right)^n \approx 0.37 \quad \text{for large } n.$$

(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'?

Solution:

Using a larger n' means that each tree has access to more data points, which generally results in lower variance. However, setting n' close to n may reduce the diversity among trees in the ensemble, as each tree will be trained on nearly identical samples. Reduced diversity can limit the effectiveness of bagging, as the benefits of averaging are most pronounced when the individual trees are diverse.

Conversely, using a smaller n' may introduce slightly higher bias in the individual trees because each tree is trained on a smaller subset of the data. However, this smaller sample size can increase the diversity among trees, helping the ensemble capture different patterns in the data, which may ultimately improve generalization.

A practical approach to selecting n' is to use cross-validation or out-of-bag (OOB) error estimation to evaluate the performance of the model for different values of n'. By examining the accuracy, error, or other relevant metrics on validation data across a range of n' values, we can identify the value that yields the best balance between accuracy and computational efficiency.

In summart, I recommend beginning with $n' = n$ as a baseline and then experimenting with smaller values to observe their impact on both performance and training time. Adjusting n' allows control over the level of diversity in the ensemble, which can be crucial in maximizing the effectiveness of bagging in ensemble learning. For large datasets, setting n' to a fraction of n (e.g., 0.8n or 0.9n) is often effective in maintaining high accuracy while improving efficiency. For smaller datasets, keeping n' close to n might be preferable to ensure that each tree has access to a wide variety of samples.

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 (in the homework PDF) 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.)

Solution:

1. **How did you deal with categorical features and missing values?**
   Categorical features were processed using one-hot encoding for specific columns (like `pclass`, `sex`, and `embarked` in the Titanic dataset), while missing values were handled by filling numerical columns with the median and categorical columns with the mode. This was achieved through a custom preprocessing function.
2. **What was your stopping criterion?**
   The stopping criterion for the decision tree included reaching the maximum depth or encountering a node where all samples belonged to the same class. Additionally, if no split improved information gain, the node was set as a leaf.
3. **How did you implement random forests?**
   The random forest was implemented as an ensemble of decision trees, each trained on a bootstrap sample of the data with a random subset of features selected at each split (based on the `max_features` parameter). We used `BaggedTrees` as a superclass for our `RandomForest` class, inheriting functionality for fitting and predicting with multiple decision trees.
4. **Did you do anything special to speed up training?**
   No specific optimizations were applied beyond leveraging the inherent parallelism in random forests by training each tree independently on a bootstrap sample. Random feature selection at each split helped to reduce overfitting and slightly reduce training time for individual trees.
5. **Anything else cool you implemented?**
   Yes, the code includes a method to visualize the decision tree structure using Graphviz, allowing us to save the tree in a `.pdf` format. Additionally, we implemented functions to calculate entropy and information gain, as well as custom handling for missing data and one-hot encoding, which were useful for preprocessing and understanding model splits in depth.

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 × 2 classifiers × training/validation).

Solution:

Dataset: titanic

Decision Tree - Training Accuracy: 0.8235, Validation Accuracy: 0.8000

Random Forest - Training Accuracy: 0.8360, Validation Accuracy: 0.7750

Dataset: spam

Decision Tree - Training Accuracy: 0.8064, Validation Accuracy: 0.8039

Random Forest - Training Accuracy: 0.8151, Validation Accuracy: 0.8116

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.

   Solution:

```
Task 1: Decision Tree Paths for Specific Data Points

Spam Email Classification Path:
(1) exclamation >= 1.0 (Value: 1.0)
(2) ampersand < 1.0 (Value: 0.0)
(3) meter < 1.0 (Value: 0.0)
(4) money < 1.0 (Value: 0.0)
(5) prescription < 1.0 (Value: 0.0)
Therefore, this email was classified as spam.

Ham Email Classification Path:
(1) exclamation < 1.0 (Value: 0.0)
(2) meter < 1.0 (Value: 0.0)
(3) parenthesis >= 1.0 (Value: 4.0)
(4) private < 1.0 (Value: 0.0)
(5) dollar < 1.0 (Value: 0.0)
Therefore, this email was classified as ham.
```

2. For random forests, find and state the most common splits made at the root node of the trees. For example:

   (a) (“viagra”) ≥ 3 (20 trees)
   (b) (“thanks”) < 4 (15 trees)
   (c) (“nigeria”) ≥ 1 (5 trees)

   Solution:

```
Task 2: Most Common Splits at Root Node in Random Forest
Most Common Root Splits:
Feature: money (15 trees)
Feature: meter (15 trees)
Feature: exclamation (13 trees)
Feature: pain (8 trees)
```

```
Feature: volumes (8 trees)
Feature: featured (8 trees)
Feature: ampersand (7 trees)
Feature: parenthesis (5 trees)
Feature: dollar (4 trees)
Feature: creative (4 trees)
Feature: spam (3 trees)
Feature: prescription (3 trees)
Feature: private (2 trees)
Feature: sharp (2 trees)
Feature: differ (1 trees)
Feature: width (1 trees)
Feature: other (1 trees)
```

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.

   Solution:

```
Task 3: Decision Tree Depth vs. Validation Accuracy
Plot saved as 'decision_tree_depth_accuracy.png'.

Maximum validation accuracy of 0.8200 achieved at depth = 35

Observation:
As the depth increases, the training accuracy generally improves, indicating that the model is fitting
However, the validation accuracy may peak at a certain depth and then decrease, suggesting overfitting
```

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.

Solution: See Appendix

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.**

   Solution: N/A

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*!

   Solution: Zhe Wee (Derrick) Ng

7 Appendix

1. **Q1**

2. **Q5.1 and Q5.2**

```
import numpy as np
import pandas as pd
import scipy.io
from collections import Counter
from sklearn.tree import DecisionTreeClassifier
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.model_selection import cross_val_score
import random

random.seed(246810)
np.random.seed(246810)
eps = 1e-5  # a small number

class DecisionTree:

    def __init__(self, max_depth=3, feature_labels=None):
        self.max_depth = max_depth
        self.features = feature_labels
        self.left, self.right = None, None
        self.split_idx, self.thresh = None, None
        self.pred = None

    @staticmethod
    def entropy(y):
        y = y.astype(int)  # Ensure y is of integer type
        counts = np.bincount(y)
        probabilities = counts / len(y)
        return -np.sum([p * np.log2(p) for p in probabilities if p > 0])

    @staticmethod
    def information_gain(X_column, y, thresh):
        parent_entropy = DecisionTree.entropy(y)

        left_indices = X_column < thresh
        right_indices = ~left_indices

        if len(y[left_indices]) == 0 or len(y[right_indices]) == 0:
            return 0

        n = len(y)
        n_left = np.sum(left_indices)
        n_right = n - n_left

        e_left = DecisionTree.entropy(y[left_indices])
        e_right = DecisionTree.entropy(y[right_indices])

        child_entropy = (n_left / n) * e_left + (n_right / n) * e_right
        return parent_entropy - child_entropy

    def split(self, X, y, feature_idx, thresh):
        left_indices = X[:, feature_idx] < thresh
        right_indices = ~left_indices
        return X[left_indices], y[left_indices], X[right_indices], y[right_indices]

    def fit(self, X, y, depth=0):
        y = y.astype(int)  # Ensure y is of integer type
        if depth >= self.max_depth or len(set(y)) == 1:
            self.pred = Counter(y).most_common(1)[0][0]
            return

        best_gain = 0
        best_split = None

        n_features = X.shape[1]
        for feature_idx in range(n_features):
            X_column = X[:, feature_idx]
            thresholds = np.unique(X_column)
            for thresh in thresholds:
                gain = self.information_gain(X_column, y, thresh)
                if gain > best_gain + eps:
                    best_gain = gain
                    best_split = {
                        'feature_idx': feature_idx,
                        'thresh': thresh
                    }

        if best_gain == 0 or best_split is None:
            self.pred = Counter(y).most_common(1)[0][0]
            return

        self.split_idx = best_split['feature_idx']
        self.thresh = best_split['thresh']

        X_left, y_left, X_right, y_right = self.split(X, y, self.split_idx, self.thresh)

        if len(y_left) == 0 or len(y_right) == 0:
            self.pred = Counter(y).most_common(1)[0][0]
            return

        self.left = DecisionTree(self.max_depth, self.features)
        self.left.fit(X_left, y_left, depth + 1)

        self.right = DecisionTree(self.max_depth, self.features)
        self.right.fit(X_right, y_right, depth + 1)
    def predict(self, X):
        if self.pred is not None:
            return np.array([self.pred] * len(X))
        else:
            left_indices = X[:, self.split_idx] < self.thresh
            right_indices = ~left_indices

            y_pred = np.empty(len(X), dtype=int)

            if np.any(left_indices):
                y_pred[left_indices] = self.left.predict(X[left_indices])
            if np.any(right_indices):
                y_pred[right_indices] = self.right.predict(X[right_indices])

            return y_pred

    def _to_graphviz(self, node_id):
        if self.split_idx is None:
            return f'{node_id} [label="Prediction: {self.pred}"];\n'
        else:
            feature_name = self.features[self.split_idx] if self.features else f"Feature {self.split_i
            graph = f'{node_id} [label="{feature_name} < {self.thresh:.2f}"];\n'
            left_id = node_id * 2 + 1
            right_id = node_id * 2 + 2
            if self.left is not None:
                graph += f'{node_id} -> {left_id} [label="Yes"];\n'
                graph += self.left._to_graphviz(left_id)
            if self.right is not None:
                graph += f'{node_id} -> {right_id} [label="No"];\n'
                graph += self.right._to_graphviz(right_id)
            return graph

    def to_graphviz(self):
        graph = "digraph Tree {\nnode [shape=box];\n"
        graph += self._to_graphviz(0)
        graph += "}\n"
        return graph

    def __repr__(self):
        if self.pred is not None:
            return f"Prediction: {self.pred}"
        else:
            feature_name = self.features[self.split_idx] if self.features else f"Feature {self.split_i
            return f"[{feature_name} < {self.thresh:.2f}: {self.left} | {self.right}]"

class BaggedTrees(BaseEstimator, ClassifierMixin):

    def __init__(self, params=None, n_estimators=200):
        self.params = params if params is not None else {}
        self.n_estimators = n_estimators
        self.decision_trees = []

    def fit(self, X, y):
        y = y.astype(int)  # Ensure y is of integer type
        self.classes_ = np.unique(y)  # Set classes_
        self.decision_trees = []
        n_samples = X.shape[0]
        for i in range(self.n_estimators):
            bootstrap_indices = np.random.choice(n_samples, n_samples, replace=True)
            X_bootstrap = X[bootstrap_indices]
            y_bootstrap = y[bootstrap_indices]
            tree = DecisionTreeClassifier(random_state=i, **self.params)
            tree.fit(X_bootstrap, y_bootstrap)
            self.decision_trees.append(tree)

    def predict(self, X):
        predictions = np.array([tree.predict(X) for tree in self.decision_trees])
        # Compute the majority vote
        y_pred = np.apply_along_axis(lambda x: np.bincount(x).argmax(), axis=0, arr=predictions.astype
        return y_pred

class RandomForest(BaggedTrees):

    def __init__(self, params=None, n_estimators=200, max_features='sqrt'):
        self.params = params if params is not None else {}
        self.n_estimators = n_estimators
        self.max_features = max_features

        self.params['max_features'] = self.max_features

        super().__init__(params=self.params, n_estimators=self.n_estimators)

def preprocess(data, onehot_cols=[]):
    data = data.copy()
    # Fill missing numerical values with median
    num_cols = data.select_dtypes(include=['float64', 'int64']).columns
    data[num_cols] = data[num_cols].fillna(data[num_cols].median())
    # Fill missing categorical values with mode
    cat_cols = data.select_dtypes(include=['object']).columns
    data[cat_cols] = data[cat_cols].fillna(data[cat_cols].mode().iloc[0])
    # One-hot encode specified columns
    data = pd.get_dummies(data, columns=onehot_cols, drop_first=True)
    return data

def evaluate(clf, X, y, features):
    y = y.astype(int)  # Ensure y is of integer type
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    print("Cross-validation scores:", scores)
```

```
        print("Mean accuracy:", np.nanmean(scores))
        if hasattr(clf, "decision_trees"):
            first_splits = []
            for tree in clf.decision_trees:
                if hasattr(tree, 'tree_'):
                    if tree.tree_.feature[0] != -2:  # -2 indicates leaf node
                        first_splits.append(tree.tree_.feature[0])
            counter = Counter(first_splits)
            first_splits_info = [
                (features[idx], count) for idx, count in counter.most_common()
            ]
            print("Most common first splits:", first_splits_info)

    def generate_submission(predictions, dataset="titanic"):
        df = pd.DataFrame({'Category': predictions})
        df.index += 1
        df.to_csv(f'predictions_{dataset}.csv', index_label='Id')

    if __name__ == "__main__":
        dataset = "titanic"
        params = {"max_depth": 5, "min_samples_leaf": 10}
        n_estimators = 100

        if dataset == "titanic":
            # Load data using pandas
            path_train = 'datasets/titanic/titanic_training.csv'
            data = pd.read_csv(path_train)

            # Print column names to verify
            print("Columns in training data:", data.columns.tolist())

            # Update label column name to 'survived' as per your dataset
            label_column = 'survived'

            if label_column not in data.columns:
                raise ValueError(f"The label column '{label_column}' is not found in the data.")

            # Drop rows where 'survived' is NaN
            data = data.dropna(subset=[label_column])

            # Convert labels to integers
            y = data[label_column].astype(int)

            # Drop 'survived', 'ticket', and 'cabin' columns from X
            X = data.drop([label_column, 'ticket', 'cabin'], axis=1)

            print("\n\nPart (b): preprocessing the Titanic dataset")
            onehot_cols = ['pclass', 'sex', 'embarked']  # Adjusted to match your dataset
            # Preprocess training data
            X = preprocess(X, onehot_cols=onehot_cols)

            # Load and preprocess test data
            path_test = 'datasets/titanic/titanic_testing_data.csv'
            test_data = pd.read_csv(path_test)

            # Drop 'ticket' and 'cabin' columns from test data
            test_data = test_data.drop(['ticket', 'cabin'], axis=1)

            Z = preprocess(test_data, onehot_cols=onehot_cols)

            # Align features between training and test data
            X, Z = X.align(Z, join='left', axis=1, fill_value=0)

            features = X.columns.tolist()

        elif dataset == "spam":
            # Load spam data
            path_train = 'datasets/spam_data/spam_data.mat'
            data = scipy.io.loadmat(path_train)
            X = pd.DataFrame(data['training_data'])
            y = np.squeeze(data['training_labels']).astype(int)
            Z = pd.DataFrame(data['test_data'])
            features = [f'feature_{i}' for i in range(X.shape[1])]

        else:
            raise NotImplementedError(f"Dataset {dataset} not handled")

        print("Features:", features)
        print("Train/test size:", X.shape, Z.shape)

        # Decision Tree
        print("\n\nDecision Tree")
        dt = DecisionTree(max_depth=3, feature_labels=features)
        dt.fit(X.values, y.values)

        # Visualize Decision Tree
        print("\n\nTree Structure")
        print(dt)
        graph = dt.to_graphviz()
        with open(f"{dataset}-basic-tree.dot", "w") as f:
            f.write(graph)

        # Random Forest
        print("\n\nRandom Forest")
        rf = RandomForest(params=params, n_estimators=n_estimators)
        rf.fit(X.values, y.values)
        evaluate(rf, X.values, y.values, features)
```

```
# Generate Test Predictions
print("\n\nGenerate Test Predictions")
pred = rf.predict(Z.values).astype(int)  # Ensure predictions are integers
generate_submission(pred, dataset)
```

3. **Q5.4**

```
def main():
    datasets = ["titanic", "spam"]
    params = {"max_depth": 5, "min_samples_leaf": 10}
    n_estimators = 100

    results = {}

    for dataset in datasets:
        print(f"\nProcessing dataset: {dataset}")

        if dataset == "titanic":
            # Load data
            path_train = 'datasets/titanic/titanic_training.csv'
            data = pd.read_csv(path_train)

            # Update label column name
            label_column = 'survived'
            data = data.dropna(subset=[label_column])
            y = data[label_column].astype(int)
            X = data.drop([label_column, 'ticket', 'cabin'], axis=1)

            onehot_cols = ['pclass', 'sex', 'embarked']
            X = preprocess(X, onehot_cols=onehot_cols)

            # Split into training and validation sets
            X_train, X_val, y_train, y_val = train_test_split(
                X.values, y.values, test_size=0.2, random_state=42
            )

            features = X.columns.tolist()

        elif dataset == "spam":
            # Load data
            path_train = 'datasets/spam_data/spam_data.mat'
            data = scipy.io.loadmat(path_train)
            X = pd.DataFrame(data['training_data'])
            y = np.squeeze(data['training_labels']).astype(int)

            # Split into training and validation sets
            X_train, X_val, y_train, y_val = train_test_split(
                X.values, y, test_size=0.2, random_state=42
            )

            features = [f'feature_{i}' for i in range(X.shape[1])]

        else:
            raise NotImplementedError(f"Dataset {dataset} not handled")
        # Initialize results dictionary
        results[dataset] = {}

        # Decision Tree
        print("\nDecision Tree")
        dt = DecisionTree(max_depth=5, feature_labels=features)
        dt.fit(X_train, y_train)
        train_acc_dt, val_acc_dt = evaluate_model(dt, X_train, y_train, X_val, y_val)
        print(f"Training Accuracy (Decision Tree): {train_acc_dt:.4f}")
        print(f"Validation Accuracy (Decision Tree): {val_acc_dt:.4f}")
        results[dataset]['Decision Tree'] = (train_acc_dt, val_acc_dt)

        # Random Forest
        print("\nRandom Forest")
        rf = RandomForest(params=params, n_estimators=n_estimators)
        rf.fit(X_train, y_train)
        train_acc_rf, val_acc_rf = evaluate_model(rf, X_train, y_train, X_val, y_val)
        print(f"Training Accuracy (Random Forest): {train_acc_rf:.4f}")
        print(f"Validation Accuracy (Random Forest): {val_acc_rf:.4f}")
        results[dataset]['Random Forest'] = (train_acc_rf, val_acc_rf)

    # Report results
    print("\n\nSummary of Results:")
    for dataset in datasets:
        print(f"\nDataset: {dataset}")
        for model in ['Decision Tree', 'Random Forest']:
            train_acc, val_acc = results[dataset][model]
            print(f"{model} - Training Accuracy: {train_acc:.4f}, Validation Accuracy: {val_acc:.4f}")

if __name__ == "__main__":
    main()
```

4. **Q5.6**