

## Instructions

- Homework 4 is due December 21st at 16:00 Chicago Time.
    - We will not accept any submissions past 16:00:00, even if they are only one second late.
  - You **must** upload the following files to the class Canvas:
    - LASTNAME\_FIRSTNAME.pdf
    - LASTNAME\_FIRSTNAME.ipynb
  - Your code notebook **must** be runnable using my environment outlines in class 1 (Python 3.14, and the `requirements.txt`).
  - You **must** use this template file and fill out your solutions for the written portion.
  - Please note that your last name and first name should match what you appear on Canvas as.
  - Include code snippets where required, as well as math and equations.
  - Be *concise* where possible, all of the homework problems can be answered in a few lines of math, code, and words.
-

## Problem 1: PCA from Scratch

In this problem, you will implement Principal Component Analysis (PCA) “by hand” (using basic linear algebra functions) and compare it to the standard implementation.

Generate a dataset with  $n = 500$  observations and 3 highly correlated features.

$$\mu = [0, 0, 0]$$

$$\Sigma = \begin{pmatrix} 1 & 0.9 & 0.7 \\ 0.9 & 1 & 0.8 \\ 0.7 & 0.8 & 1 \end{pmatrix}$$

```
1 import numpy as np
2 np.random.seed(42)
3 mean = [0, 0, 0]
4 cov = [[1, 0.9, 0.7], [0.9, 1, 0.8], [0.7, 0.8, 1]]
5 X = np.random.multivariate_normal(mean, cov, 500)
```

### Problem 1.1: Eigendecomposition

- Calculate the covariance matrix of the data  $X$  manually (recall  $\text{Cov}(X) = \frac{1}{n-1}X^T X$  if  $X$  is centered).
- Find the eigenvalues and eigenvectors of this covariance matrix using `np.linalg.eig`.
- Report the eigenvalues and eigenvectors.

*Answer:*

### Problem 1.2: Projection

- Project the data  $X$  onto the principal components (the eigenvectors).
- Plot the original data (pick any 2 dimensions) and the projected data (PC1 vs PC2).

*Answer:*

### Problem 1.3: Verification

- Use `sklearn.decomposition.PCA` to perform PCA on the same data.
- Compare the components (eigenvectors) and explained variance (eigenvalues) from `sklearn` with your manual calculation. Are they the same?

*Answer:*

## Problem 2: The Pitfalls of Principal Component Regression (PCR)

A common misconception is that the principal components with the largest variance are always the most useful features for prediction. This problem illustrates a scenario where this is **not** true.

Generate the following dataset:

```
1 import numpy as np
2 import pandas as pd
3 import statsmodels.api as sm
4 from sklearn.decomposition import PCA
5
6 np.random.seed(42)
7 n = 500
8
9 X_high_var = np.random.normal(0, 10, (n, 7))
10 X_low_var = np.random.normal(0, 1, (n, 3))
11 X = np.hstack([X_high_var, X_low_var])
12
13 true_beta = np.array([0]*7 + [2, -2, 2])
14 y = X @ true_beta + np.random.normal(0, 0.5, n)
```

### Problem 2.1: PCA Analysis

Run PCA on the raw input matrix  $X$ . You must center the data first ( $\tilde{X} = X - \bar{X}$ ).

- Report the explained variance ratio for all 10 components.
- Which components capture the majority of the variance?

*Answer:*

### Problem 2.2: PCR - Regressing on High Variance Components

Fit an OLS model using **only** the first 7 principal components to predict  $y$ .

- Report the  $R^2$  of this model.
- Does capturing the majority of the variance ensure good predictive power?

*Answer:*

### Problem 2.3: PCR - Regressing on Low Variance Components

Fit an OLS model using **only** the last 3 principal components (PC8, PC9, PC10) to predict  $y$ .

- Report the  $R^2$  of this model.

*Answer:*

### Problem 2.4: Conclusion

Based on your results, explain why simply selecting the top  $k$  principal components for a regression model might be dangerous.

*Answer:*

### Problem 3: Bagging from Scratch

In class, we discussed how Bagging (Bootstrap Aggregating) reduces variance by averaging predictions from multiple models trained on bootstrapped samples. You will implement this manually.

Generate a synthetic “checkerboard” dataset:

```
1 import pandas as pd
2 np.random.seed(42)
3 n_points = 500
4 grid_size = 4
5 limit = 10
6 step = (limit * 2) / grid_size
7
8 all_points = []
9 all_labels = []
10
11 for i in range(grid_size):
12     for j in range(grid_size):
13         label = (i + j) % 2
14         x_min, x_max = -limit + i * step, -limit + (i + 1) * step
15         y_min, y_max = -limit + j * step, -limit + (j + 1) * step
16         points_x = np.random.uniform(x_min, x_max, int(n_points/16))
17         points_y = np.random.uniform(y_min, y_max, int(n_points/16))
18         all_points.append(np.vstack((points_x, points_y)).T)
19         all_labels.extend([label] * int(n_points/16))
20
21 X = np.vstack(all_points)
22 y = np.array(all_labels)
```

### Problem 3.1: Single Decision Tree

Fit a single `DecisionTreeClassifier` (from `sklearn`) with `max_depth=None` (or a high number like 10) to the data.

- Visualize the decision boundary (you can use the `meshgrid` method shown in class).
- Does the decision boundary look smooth? Does it look like it's overfitting?

*Answer:*

### Problem 3.2: Manual Bagging Implementation

Implement Bagging manually (do **not** use `BaggingClassifier`).

- Create a loop that runs  $B = 100$  times.
- Inside the loop:
  1. Create a bootstrap sample of the dataset (sample  $N$  rows with replacement).
  2. Fit a new `DecisionTreeClassifier` (`max_depth=5`) on this bootstrap sample.
  3. Store the trained tree in a list.
- To predict for a new point (or the meshgrid for plotting), get predictions from all 100 trees and take the majority vote (average).

Visualize the decision boundary of your manual Bagging ensemble. Compare it to the single decision tree in 3.1. Is it smoother?

*Answer:*

## Problem 4: Gradient Boosting from Scratch

We often use OLS because it is simple and interpretable. However, it can suffer from high bias if the true relationship is non-linear. In this problem, you will implement Gradient Boosting manually to correct the bias of an OLS model.

Generate the following non-linear dataset:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.linear_model import LinearRegression
4 from sklearn.tree import DecisionTreeRegressor
5
6 np.random.seed(42)
7 X = np.linspace(0, 10, 100).reshape(-1, 1)
8 # y is exponential, which OLS struggles with
9 y = np.exp(X.ravel() / 3) + np.random.normal(0, 0.5, 100)
```

### Problem 4.1: The Base Model (OLS)

- Fit a standard Linear Regression (OLS) model to  $X$  and  $y$ .
- Calculate and report the Mean Squared Error (MSE).
- Plot the data and the OLS prediction line. Observe how the straight line fails to capture the curve (high bias).

*Answer:*

### Problem 4.2: Boosting with Residuals

Now, we will boost this model by iteratively fitting the residuals with weak learners (shallow trees). Implement the following algorithm:

1. Initialize predictions with the OLS model:  $F_0(x) = \hat{y}_{OLS}$ .
2. Set learning rate  $\eta = 0.1$ .
3. Loop 50 times ( $m = 1$  to 50):
  - Calculate residuals:  $r_m = y - F_{m-1}(x)$ .
  - Fit a `DecisionTreeRegressor` with `max_depth=1` to the data  $(X, r_m)$ . This is your weak learner  $h_m(x)$ .
  - Update predictions:  $F_m(x) = F_{m-1}(x) + \eta \cdot h_m(x)$ .

**Answer:**

### Problem 4.3: Results

- Plot the final boosted prediction  $F_{50}(x)$  against the original data.
- Calculate the final MSE. Compare it to the OLS MSE.
- Explain intuitively what the boosting process did to the original linear line.

**Answer:**

## Problem 5: Boosting vs. Random Forest

Using the same dataset as Problem 3 (the checkerboard), we will compare Random Forest (which uses Bagging + feature splitting) and AdaBoost (which reduces bias).

### Problem 5.1: Model Comparison

- Fit a `RandomForestClassifier` with 100 estimators.
- Fit an `AdaBoostClassifier` with 50 estimators (base estimator can be a Tree with depth 3).
- Report the accuracy of both models on the dataset (since we didn't do a train/test split, reporting training accuracy is fine for this illustrative comparison).

**Answer:**

### Problem 5.2: Conceptual Question

Explain in 2-3 sentences:

- Why does Random Forest help reduce *variance*?
- Why does Boosting help reduce *bias*?

**Answer:**

## Problem 6: Clustering from Scratch (DBSCAN Flavor)

Standard K-Means clustering assumes that clusters are spherical and roughly the same size. DBSCAN is an alternative that finds clusters of arbitrary shape. You will implement a simplified version of this.

Generate the “Moons” dataset:

```
1 from sklearn.datasets import make_moons
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 np.random.seed(42)
6 X, _ = make_moons(n_samples=300, noise=0.05)
```

### Problem 6.1: The Algorithm

It may be helpful to cache the pairwise distances between points to speed up the algorithm, you can do this using `scipy.spatial.distance.cdist` or numpy operations.

Implement the following simplified DBSCAN logic:

- Set  $\epsilon = 0.2$  and `MinPts = 5`.
- Initialize all points as unvisited.
- Initialize `cluster_id = 0`.
- Loop through each point  $P$  in  $X$ :
  - If  $P$  is visited, continue.
  - Mark  $P$  as visited.
  - Find all neighbors of  $P$  within distance  $\epsilon$ .
  - If number of neighbors  $< \text{MinPts}$ , mark  $P$  as Noise (Cluster -1).
  - If number of neighbors  $\geq \text{MinPts}$ :
    - \* Increment `cluster_id`.
    - \* Assign  $P$  to `cluster_id`.
    - \* Expand the cluster (Breadth-First Search): Add all neighbors to a queue. For each point  $Q$  in the queue:
      - If  $Q$  is unvisited, mark it visited.
      - If  $Q$  has enough neighbors ( $\geq \text{MinPts}$ ), add those neighbors to the queue.
      - If  $Q$  is not yet assigned to a cluster, assign it to `cluster_id`.

**Answer:**

**Problem 6.3: Visualization**

- Plot the points  $X$ , colored by their assigned cluster labels.
- Verify using `sklearn.cluster.DBSCAN` with the same parameters. Do the plots look the same?

*Answer:*