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 probelms can be answered in a few lines of math, code, and words.

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.

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

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

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

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:

```python
X_demeaned = X - X.mean(axis=0)
covX = 1/499 * X_demeaned.T @ X_demeaned
eigenvalues, eigenvectors = np.linalg.eig(covX)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)
```

The eigenvalues and eigenvectors are:

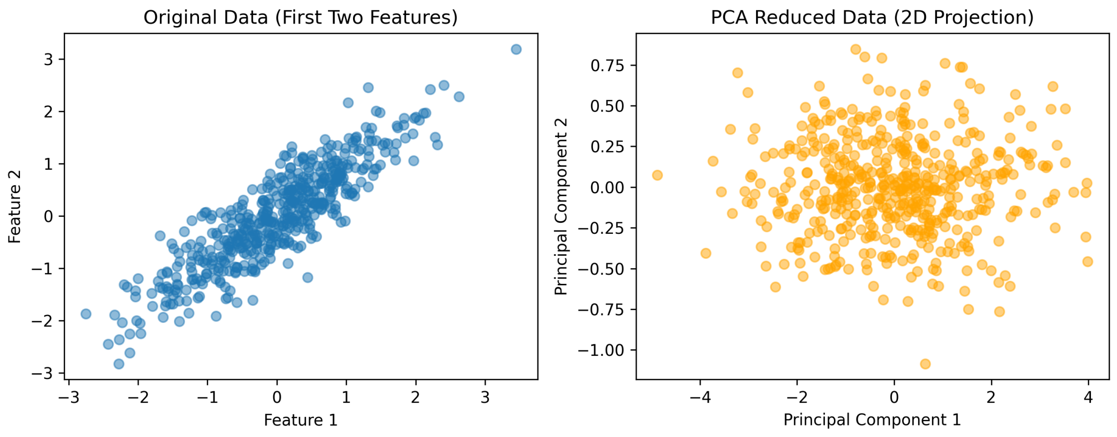
$$\lambda = (2.29077816,\ 0.08782584,\ 0.31300638)$$

$$Q = \begin{bmatrix} -0.5859901 & -0.57470129 & -0.57125653 \\ -0.60499383 & 0.77928614 & -0.16338781 \\ -0.53907148 & -0.24986304 & 0.8043447 \end{bmatrix}$$

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:



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:

The components and explained variance from `sklearn` match the manual calculations (up to sign differences in eigenvectors).

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:

```python
import numpy as np
import pandas as pd
import statsmodels.api as sm
from sklearn.decomposition import PCA

np.random.seed(42)
n = 500

X_high_var = np.random.normal(0, 10, (n, 7))
X_low_var = np.random.normal(0, 1, (n, 3))
X = np.hstack([X_high_var, X_low_var])

true_beta = np.array([0]*7 + [2, -2, 2])
y = X @ true_beta + np.random.normal(0, 0.5, n)
```

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:

The explained variance ratios are

$$\begin{gathered}(0.16772843,\ 0.16159361,\ 0.15786878,\ 0.14254598,\ 0.12969401,\\ 0.12647038,\ 0.10982575,\ 0.00148884,\ 0.00142307,\ 0.00136117)\end{gathered}$$

The first 7 components capture the majority of the variance, as they correspond to the high variance features.

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:The R^2 of this model is very low (0.006), indicating poor predictive power. Capturing the majority of the variance that are irrelvant on predicting y does not ensure good predictive power.

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:

Using the last 3 principal components, we get $R^2 \approx 0.98$.

2.4: Conclusion

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

Answer:

Simply selecting the top k principal components for a regression model might be dangerous because the components that capture the most variance are not necessarily the most predictive of the response variable. In this example, the low variance components actually contained the signal for predicting y, while the high variance components were mostly noise. Therefore, relying solely on variance to select components can lead to poor predictive performance.

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:

```python
import pandas as pd
np.random.seed(42)
n_points = 500
grid_size = 4
limit = 10
step = (limit * 2) / grid_size

all_points = []
all_labels = []

for i in range(grid_size):
    for j in range(grid_size):
        label = (i + j) % 2
        x_min, x_max = -limit + i * step, -limit + (i + 1) * step
        y_min, y_max = -limit + j * step, -limit + (j + 1) * step
        points_x = np.random.uniform(x_min, x_max, int(n_points/16))
        points_y = np.random.uniform(y_min, y_max, int(n_points/16))
        all_points.append(np.vstack((points_x, points_y)).T)
        all_labels.extend([label] * int(n_points/16))

X = np.vstack(all_points)
y = np.array(all_labels)
```

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?

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:

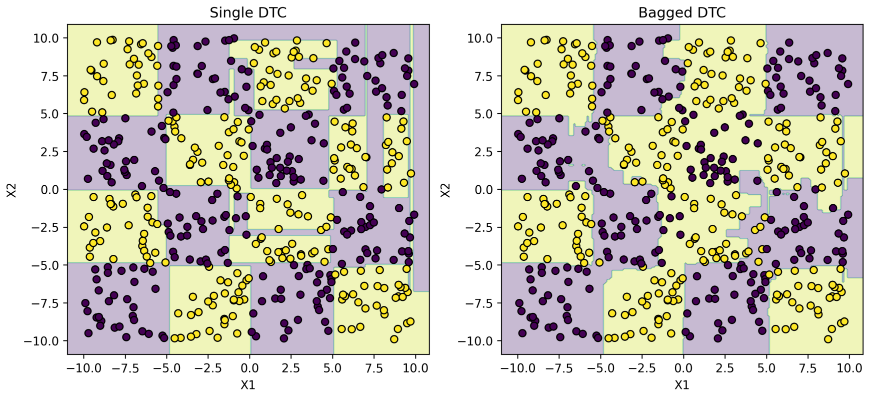


Figure 1: Decision boundary of the Bagging ensemble

3.1 The decision boundary of single decision tree look quite complex and overfit the training data.

3.2 The decision boundary of the Bagging ensemble is much smoother and generalizes better compared to the single decision tree.

```python
class BaggingDTC:
    def __init__(self, B=100):
        self.B = B
        self.trees = []

    def fit(self, X, y):
        n_samples = X.shape[0]
        for b in range(self.B):
            indices = np.random.choice(n_samples, size=n_samples, replace=True)
            X_bag = X[indices]
            y_bag = y[indices]
            tree = DecisionTreeClassifier(max_depth=None, random_state=42)
            tree.fit(X_bag, y_bag)
            self.trees.append(tree)

    def predict(self, X):
        predictions = np.array([tree.predict(X) for tree in self.trees])
        majority_votes = np.apply_along_axis(lambda x: np.bincount(x).argmax(),
    axis=0, arr=predictions)
        return majority_votes
```

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:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor

np.random.seed(42)
X = np.linspace(0, 10, 100).reshape(-1, 1)
# y is exponential, which OLS struggles with
y = np.exp(X.ravel() / 3) + np.random.normal(0, 0.5, 100)
```

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:

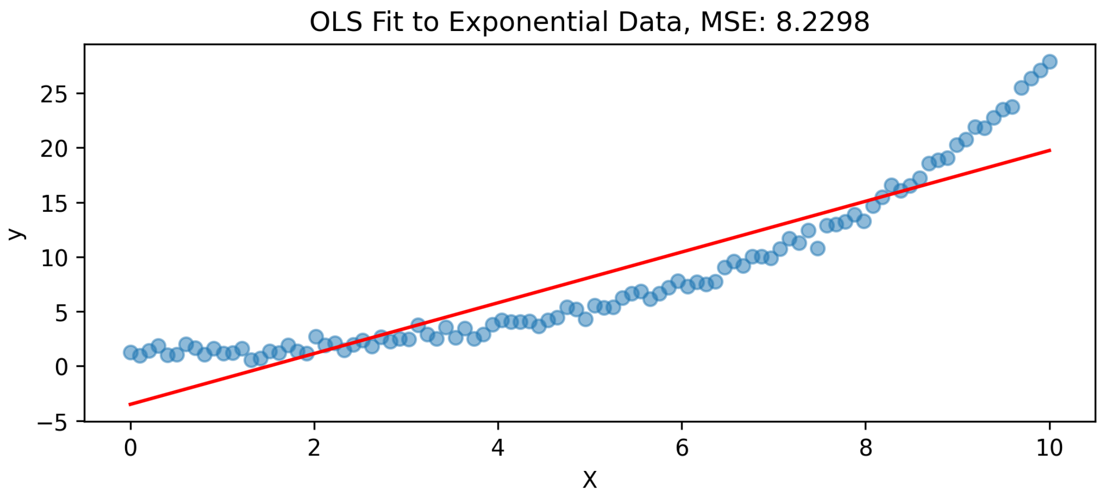


Figure 2: OLS Fit to Exponential Data

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:

```python
pred = ols.fittedvalues.copy()
eta = 0.1
for _ in range(50):
    resid_n = y - pred
    tree = DecisionTreeRegressor(max_depth=1).fit(X, resid_n)
    pred += eta * tree.predict(X)
```

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:

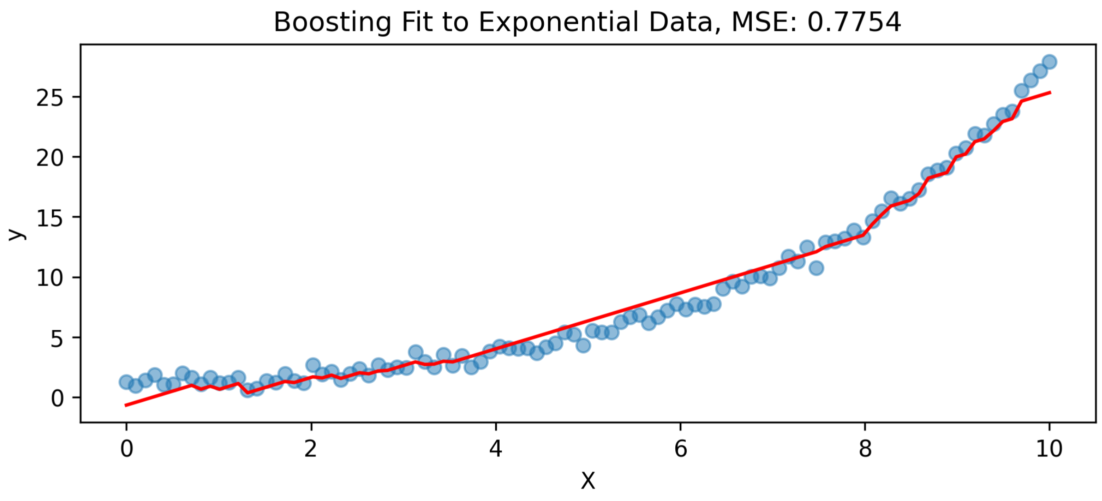


Figure 3: Boosting Fit to Exponential Data

The final MSE after boosting is approximately 10 times lower than the OLS MSE. The boosting process iteratively corrected the bias of the original linear model by fitting small decision trees to the residuals, allowing the final prediction to closely follow the non-linear pattern of the data.

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).

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:

With $70 - 30$ train-test split, we get: Random Forest Accuracy: 0.8926 AdaBoost Accuracy: 0.9128

In-sample accuracy is 1, since the two hyperparameter settings result with large enough models to overfit the training data perfectly.

5.2: Conceptual Question

Explain in 2-3 sentences:

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

Answer:

Random Forest reduces variance by averaging predictions from multiple decision trees trained on different bootstrapped samples of the data, which smooths out fluctuations due to random noise. However, each individual tree may still be biased, and averaging them does not correct this inherent bias well.

Boosting reduces bias by sequentially fitting models to the residuals of previous models, allowing the ensemble to correct errors and capture complex patterns that a single model might miss. On the other hand, since boosting focuses on correcting errors, it can be prone to overfitting if not properly regularized.

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:

```python
from sklearn.datasets import make_moons
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)
X, _ = make_moons(n_samples=300, noise=0.05)
```

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 = -1`.
- Loop through each point P in X:
  - If P is visited, continue.
  - Mark P as visited.
  - Find all neighbors of P within distance ϵ.
  - If number of neighbors $<$ MinPts, mark P as Noise (Cluster `-1`).
  - If number of neighbors $\geq$ 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 ($\geq$ MinPts), add those neighbors to the queue.
      - If Q is not yet assigned to a cluster, assign it to `cluster_id`.

Answer:

```python
def cdist(a, b):
    return np.sqrt(((a[:, np.newaxis, :] - b[np.newaxis, :, :]) ** 2).sum(axis=2))

```

```python
eps = 0.2
min_pts = 5
labels = np.full(X_moons.shape[0], -2) # -2: unvisited, -1: noise
cluster_id = -1
dists = cdist(X_moons, X_moons)

for i in range(len(X_moons)):
    if labels[i] != -2: continue
    neighbors = np.where(dists[i] < eps)[0]
    if len(neighbors) < min_pts:
        labels[i] = -1
    else:
        cluster_id += 1
        labels[i] = cluster_id
        queue = list(neighbors)
        while queue:
            q = queue.pop(0)
            if labels[q] != -2: continue # if already assigned to a cluster, skip
            labels[q] = cluster_id  # assign
            q_neighbors = np.where(dists[q] < eps)[0]
            if len(q_neighbors) >= min_pts:
                queue.extend(q_neighbors)
```

6.2: 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:Both look the same.

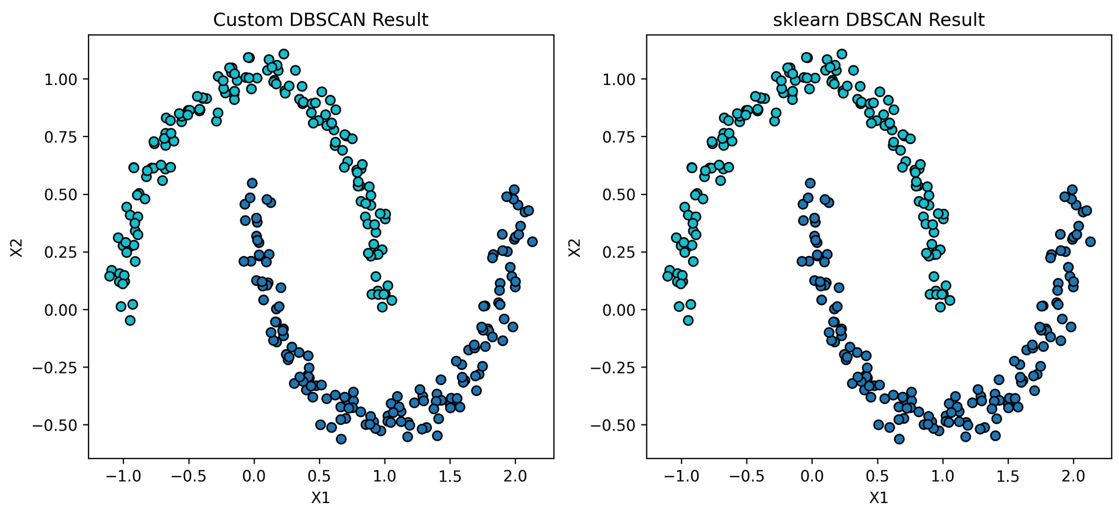


Figure 4: DBSCAN Clustering Results