ASSIGNMENT : Bagging & Boosting KNN & Stacking Assignment

QUESTION 1 : What is the fundamental idea behind ensemble techniques? How does bagging differ from boosting in terms of approach and objective?

ANSWER 1 : The **fundamental idea behind ensemble techniques** is to **combine multiple individual (weak) models** to create a **stronger, more accurate, and more robust predictive model**.
Instead of relying on a single model, ensemble methods aggregate the predictions of many models to reduce **variance**, **bias**, or **overfitting**, depending on the specific technique used.

🌳 1. Bagging (Bootstrap Aggregating)

Approach:

- Bagging builds **multiple independent models** (usually of the same type, like Decision Trees) on **different random subsets** of the training data.
- These subsets are created through **bootstrapping** — random sampling *with replacement*.
- The predictions from all models are then **averaged** (for regression) or **voted** (for classification).

Objective:

- **Reduce variance** (i.e., make the model more stable and less sensitive to data noise).
- Works well for **high-variance, low-bias** models like Decision Trees.

Example algorithms:

- Random Forest (ensemble of Decision Trees using bagging)

⚡ 2. Boosting

Approach:

- Boosting builds models **sequentially**, where each new model **focuses on correcting the errors** made by the previous models.
- It assigns **higher weights** to misclassified or poorly predicted samples so that future models pay more attention to them.
- The final model is a **weighted sum** of all weak learners.

Objective:

- **Reduce bias** and improve predictive accuracy by turning weak learners into a strong learner.
- Works well for **high-bias, low-variance** problems.

Example algorithms:

- AdaBoost, Gradient Boosting, XGBoost, LightGBM, CatBoost

🧠 Summary Table:

Feature	Bagging	Boosting
Model Training	Parallel (independent models)	Sequential (each depends on previous)
Data Sampling	Bootstrap samples (random subsets)	Weighted sampling based on errors
Objective	Reduce variance	Reduce bias
Model Combination	Simple averaging or voting	Weighted sum of weak learners
Typical Base Model	Decision Trees	Shallow Decision Trees (stumps)
Example Algorithms	Random Forest	AdaBoost, Gradient Boosting, XGBoost

---

✅ **In short:**

- **Bagging = Parallel learners → variance reduction.**
- **Boosting = Sequential learners → bias reduction.**

QUESTION 2 : Explain how the Random Forest Classifier reduces overfitting compared to a single decision tree. Mention the role of two key hyperparameters in this process.

ANSWER 2 : A **Random Forest Classifier** reduces overfitting compared to a single Decision Tree by combining the predictions of **many independent trees**, each trained on a **random subset** of the data and features.

Here's how it works and why it helps:

🔹 Fundamental Idea:

- A **single Decision Tree** tends to overfit — it learns noise and specific patterns from the training data, leading to poor generalization.
- A **Random Forest** builds **multiple trees (an ensemble)**, each slightly different, and then **averages** (for regression) or **takes a majority vote** (for classification).
  This averaging process **reduces variance** and prevents overfitting.

🔹 How Randomness Helps:

1. **Bootstrap Sampling (Bagging):**
    - Each tree is trained on a random sample (with replacement) of the training data.
    - This means each tree sees slightly different data → less correlation between trees → reduced overfitting.
2. **Random Feature Selection:**
    - At each split, the model considers only a random subset of features, not all.

- This prevents strong predictors from dominating every tree and adds diversity to the ensemble.

🔹 Two Key Hyperparameters That Help Control Overfitting:

1. **`n_estimators` (Number of Trees):**
   - More trees → better averaging and lower variance.
   - Beyond a certain point, adding more trees gives diminishing returns but still stabilizes the model.
2. **`max_features` (Number of Features Considered at Each Split):**
   - Smaller `max_features` → trees become more diverse → less overfitting.
   - Larger `max_features` → trees become more similar → higher risk of overfitting (like a single large tree).

✅ Summary:

Aspect	Decision Tree	Random Forest
Structure	Single, deep tree	Many shallow, random trees
Variance	High	Reduced via averaging
Overfitting	Common	Significantly reduced
Key Hyperparameters	–	`n_estimators`, `max_features`

In short:
Random Forest reduces overfitting by using **bagging** and **feature randomness** to create diverse trees and then **aggregating** their results.
The hyperparameters **`n_estimators`** (number of trees) and **`max_features`** (feature randomness) are key to controlling this process.

QUESTION 3 : What is Stacking in ensemble learning? How does it differ from traditional bagging/boosting methods? Provide a simple example use case.

ANSWER 3 : **Stacking (Stacked Generalization)** is an **ensemble learning technique** where **multiple different models (base learners)** are trained, and then their predictions are **combined using another model (meta-learner or blender)** to make the final prediction.

🔹 How Stacking Works:

1. **Train Base Models (Level-0 models):**
   Multiple algorithms (e.g., Decision Tree, SVM, Logistic Regression) are trained on the same dataset.

2. **Generate Meta Features:**
   Each base model makes predictions (on validation data). These predictions are treated as **new features**.

3. **Train Meta-Model (Level-1 model):**
   A new model (often a simple one like Linear Regression or Logistic Regression) learns how to best combine the base model outputs.

🔹 Difference from Bagging and Boosting:

Aspect	Bagging	Boosting	Stacking
Base Learners	Same algorithm (e.g., many Decision Trees in Random Forest)	Same algorithm trained sequentially (e.g., Decision Trees in AdaBoost, XGBoost)	Different algorithms can be combined
Training Style	Independent and parallel	Sequential (each model fixes errors of the previous)	Parallel at base level, then combined via meta-model
Combination Method	Averaging (for regression) or Voting (for classification)	Weighted combination	Learned combination using a meta-model
Goal	Reduce variance	Reduce bias	Leverage strengths of diverse models

---

🔹 Example Use Case:

Predicting loan default (classification problem)

- **Base Models (Level-0):**
    - Random Forest → captures non-linear relationships
    - Logistic Regression → good for linear trends
    - Gradient Boosting → strong on complex interactions
- **Meta-Model (Level-1):**
    - Logistic Regression → learns how to combine predictions from the three base models optimally

Result:
The stacked model often outperforms any individual model because it integrates their diverse perspectives.

✅ Summary:

Stacking = combining multiple different models through a meta-learner to improve predictive performance.
Unlike **bagging** and **boosting**, stacking **learns how to combine** model predictions instead of just averaging or weighting them.

QUESTION 4 : What is the OOB Score in Random Forest, and why is it useful? How does it help in model evaluation without a separate validation set?

ANSWER 4 : 🌲 **OOB (Out-of-Bag) Score in Random Forest**

Definition:
The **OOB (Out-of-Bag) Score** is an internal validation method used in **Random Forests** to estimate the model's performance **without needing a separate validation or test set**.

🔍 How It Works

When building each decision tree in a Random Forest:

- The algorithm uses **bootstrap sampling** — it randomly selects samples **with replacement** from the training data to train the tree.
- On average, about **63%** of the data points are selected (some are repeated).
- The remaining **~37%** of samples are **not used** to train that tree — these are called **Out-of-Bag (OOB) samples**.

After the forest is trained:

- Each data point is predicted **only by the trees that did not see it** during training (i.e., trees for which it was OOB).
- The final **OOB prediction** for each sample is obtained by **aggregating** these predictions (majority vote for classification, average for regression).
- The **OOB Score** is the accuracy (or R^2 for regression) computed from these OOB predictions.

💡 Why It's Useful

- ✅ **No need for a separate validation set:**
  The OOB Score acts like an internal cross-validation, saving data for training.
- ✅ **Efficient and unbiased estimate:**
  It provides an **unbiased estimate of model performance**, as each prediction comes from trees that haven't seen that data point.
- ✅ **Saves computation:**
  Since it reuses training data, no extra validation process is required.

🧮 Example

```
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier

# Load dataset
```

```
X, y = load_iris(return_X_y=True)

# Train Random Forest with OOB score enabled
rf = RandomForestClassifier(n_estimators=100, oob_score=True, random_state=42)
rf.fit(X, y)

# Print OOB score
print("OOB Score:", rf.oob_score_)
```

Output Example:

```
OOB Score: 0.96
```

📊 In Summary

Aspect	Description
Concept	Performance estimate using unseen (OOB) samples
Approx. size of OOB data per tree	~37% of total data
Main benefit	No need for a separate validation set
Used for	Estimating generalization accuracy efficiently
Similar to	Cross-validation, but faster and built-in

👉 **In short:**
The **OOB Score** provides a **built-in, unbiased validation measure** in Random Forests, letting you assess model performance **without holding out extra data**.

QUESTION 5 : Compare AdaBoost and Gradient Boosting in terms of: ● How they handle errors from weak learners ● Weight adjustment mechanism ● Typical use cases

ANSWER 5 : Here's a clear **comparison between AdaBoost and Gradient Boosting** across the requested points:

Aspect	AdaBoost (Adaptive Boosting)	Gradient Boosting
1. Handling errors from weak learners	Focuses more on **misclassified samples** from previous learners — increases their weights so the next learner pays more attention to them.	Focuses on **residual errors (prediction errors)** — each new learner tries to correct the residuals (differences between actual and predicted values) from the previous model.
2. Weight adjustment mechanism	Adjusts **sample weights** after each iteration: correctly classified samples get lower weights, and misclassified ones get higher weights.	Does **not modify sample weights** directly; instead, it fits new learners to minimize the **loss function's gradient** (i.e., learns from residuals).
3. Typical use cases	Works well with **simple base learners** (e.g., decision stumps) and when data is not too noisy. Often used for **classification tasks**.	More flexible and powerful — used in **both regression and classification**, and serves as the foundation for libraries like **XGBoost, LightGBM, and CatBoost**.

Summary:

- **AdaBoost** adjusts **weights of samples** to focus on hard-to-classify points.
- **Gradient Boosting** adjusts **model predictions** iteratively to minimize overall loss using gradient descent.
- **AdaBoost** is simpler and faster; **Gradient Boosting** is more general and powerful for complex, large-scale problems.

QUESTION 6 : Why does CatBoost perform well on categorical features without requiring extensive preprocessing? Briefly explain its handling of categorical variables.

ANSWER 6 : CatBoost performs well on **categorical features** because it has a **built-in, efficient way to handle them** — without the need for manual preprocessing like one-hot encoding or label encoding.

🔹 Key Idea:

CatBoost uses a technique called **“Ordered Target Statistics” (or Ordered Encoding)** to convert categorical values into numerical form **based on target values**, while **avoiding target leakage**.

🔹 How CatBoost Handles Categorical Variables:

1. **Target-Based Encoding:**
   Each categorical feature is transformed into a numeric value based on statistics of the target variable (e.g., mean target value for that category).

   [
   \text{Encoded Value} = \frac{\text{Sum of target values for the category} + \text{prior}}{\text{Count of occurrences of the category} + 1}
   ]

2. **Ordered Encoding to Prevent Leakage:**
   Instead of using the whole dataset (which could leak future information), CatBoost encodes each sample using only **data from previous samples** in a random permutation order.

3. **Efficient Combination of Categorical Features:**
   CatBoost can also create **combinations of categorical features** automatically, capturing complex interactions that would normally require manual feature engineering.

🔹 Why It Performs Well:

- No need for one-hot encoding (saves memory and time).
- Avoids overfitting and target leakage through ordered statistics.
- Automatically captures relationships between categorical and numerical features.

In short:
CatBoost natively handles categorical features by applying **ordered target-based encoding**, which makes it both **efficient and accurate** without heavy preprocessing.

QUESTION 7 : KNN Classifier Assignment: Wine Dataset Analysis with Optimization Task: 1. Load the Wine dataset (sklearn.datasets.load_wine()). 2. Split data into 70% train and 30% test. 3. Train a KNN classifier (default K=5) without scaling and evaluate using: a. Accuracy b. Precision, Recall, F1-Score (print classification report) 4. Apply StandardScaler, retrain KNN, and compare metrics. 5. Use GridSearchCV to find the best K (test K=1 to 20) and distance metric (Euclidean, Manhattan).

ANSWER 7 : Here's the full Python code for your **KNN Classifier Assignment on the Wine Dataset**, including all requested steps, evaluation metrics, scaling, and optimization using GridSearchCV 👇

```
# KNN Classifier Assignment: Wine Dataset Analysis with Optimization

from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report

# 1. Load the Wine dataset
wine = load_wine()
X, y = wine.data, wine.target

# 2. Split data into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# 3. Train KNN (K=5) without scaling
knn_default = KNeighborsClassifier(n_neighbors=5)
knn_default.fit(X_train, y_train)
y_pred_default = knn_default.predict(X_test)

print("=== Without Scaling ===")
print("Accuracy:", accuracy_score(y_test, y_pred_default))
print("\nClassification Report:\n", classification_report(y_test, y_pred_default))

# 4. Apply StandardScaler, retrain KNN, and compare
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

knn_scaled = KNeighborsClassifier(n_neighbors=5)
```

```
knn_scaled.fit(X_train_scaled, y_train)
y_pred_scaled = knn_scaled.predict(X_test_scaled)

print("\n=== With StandardScaler ===")
print("Accuracy:", accuracy_score(y_test, y_pred_scaled))
print("\nClassification Report:\n", classification_report(y_test, y_pred_scaled))

# 5. Hyperparameter tuning using GridSearchCV
param_grid = {
    'n_neighbors': list(range(1, 21)),  # K = 1 to 20
    'metric': ['euclidean', 'manhattan']
}

grid = GridSearchCV(
    KNeighborsClassifier(),
    param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)
grid.fit(X_train_scaled, y_train)

print("\n=== GridSearchCV Results ===")
print("Best Parameters:", grid.best_params_)
print("Best Cross-Validation Accuracy:", grid.best_score_)

# Evaluate best model on test data
best_knn = grid.best_estimator_
y_pred_best = best_knn.predict(X_test_scaled)

print("\n=== Optimized Model Evaluation ===")
print("Test Accuracy:", accuracy_score(y_test, y_pred_best))
print("\nClassification Report:\n", classification_report(y_test, y_pred_best))
```

🔍 Explanation of Steps

1. **Dataset** – Loads the Wine dataset from `sklearn.datasets`.
2. **Train-Test Split** – 70% for training and 30% for testing (stratified for class balance).
3. **Initial KNN (K=5)** – Trained on unscaled data; prints accuracy and classification report.

4. **Standardization** – Uses `StandardScaler` to normalize features, improving distance-based performance.
5. **Optimization** – Uses `GridSearchCV` to tune `n_neighbors` (1–20) and distance metrics (`euclidean`, `manhattan`).

QUESTION 8 : PCA + KNN with Variance Analysis and Visualization Task: 1. Load the Breast Cancer dataset (sklearn.datasets.load_breast_cancer()). 2. Apply PCA and plot the scree plot (explained variance ratio). 3. Retain 95% variance and transform the dataset. 4. Train KNN on the original data and PCA-transformed data, then compare accuracy. 5. Visualize the first two principal components using a scatter plot (color by class).

ANSWER 8 : Here's a **complete Python program** that performs PCA + KNN with variance analysis and visualization using the **Breast Cancer dataset**:

```
# PCA + KNN with Variance Analysis and Visualization

# ---------------------------------------------


from sklearn.datasets import load_breast_cancer

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from sklearn.neighbors import KNeighborsClassifier

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score

import matplotlib.pyplot as plt

import numpy as np


# 1. Load the dataset

data = load_breast_cancer()
```

```
X = data.data

y = data.target


# Standardize the data before PCA

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)


# 2. Apply PCA and plot the scree plot

pca = PCA()

pca.fit(X_scaled)

explained_variance = np.cumsum(pca.explained_variance_ratio_)


plt.figure(figsize=(8,5))

plt.plot(np.arange(1, len(explained_variance)+1), explained_variance, marker='o', color='b')

plt.title('Scree Plot (Cumulative Explained Variance)')

plt.xlabel('Number of Principal Components')

plt.ylabel('Cumulative Explained Variance Ratio')

plt.grid(True)

plt.show()


# 3. Retain 95% variance and transform the dataset

pca_95 = PCA(0.95)

X_pca = pca_95.fit_transform(X_scaled)

print(f"Number of components retained for 95% variance: {pca_95.n_components_}")
```

```
# 4. Train KNN on original and PCA-transformed data and compare accuracy

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.3, random_state=42)


# KNN on original data

knn_original = KNeighborsClassifier(n_neighbors=5)

knn_original.fit(X_train, y_train)

y_pred_original = knn_original.predict(X_test)

acc_original = accuracy_score(y_test, y_pred_original)


# KNN on PCA data

X_train_pca, X_test_pca, _, _ = train_test_split(X_pca, y, test_size=0.3, random_state=42)

knn_pca = KNeighborsClassifier(n_neighbors=5)

knn_pca.fit(X_train_pca, y_train)

y_pred_pca = knn_pca.predict(X_test_pca)

acc_pca = accuracy_score(y_test, y_pred_pca)


print(f"Accuracy on Original Data: {acc_original:.4f}")

print(f"Accuracy on PCA (95% variance) Data: {acc_pca:.4f}")


# 5. Visualize the first two principal components

pca_2 = PCA(n_components=2)

X_2D = pca_2.fit_transform(X_scaled)
```

```
plt.figure(figsize=(8,6))

plt.scatter(X_2D[:, 0], X_2D[:, 1], c=y, cmap='coolwarm', alpha=0.7)

plt.title('PCA - First Two Principal Components')

plt.xlabel('Principal Component 1')

plt.ylabel('Principal Component 2')

plt.colorbar(label='Class (0 = Malignant, 1 = Benign)')

plt.show()
```

🔍 Explanation:

1. **Dataset**: Loads the Breast Cancer dataset from `sklearn.datasets`.
2. **Scaling**: Standardization ensures PCA works properly (since it's variance-based).
3. **Scree Plot**: Shows how much variance each principal component explains (helps choose optimal number).
4. **PCA (95%)**: Retains components explaining **95%** of the total variance.
5. **KNN Comparison**:
    - Trains KNN (k=5) on **original scaled data** and **PCA-reduced data**.
    - Compares their **accuracy scores**.
6. **Visualization**: Uses a **scatter plot of first two principal components** colored by class to show class separability.

QUESTION 9 : KNN Regressor with Distance Metrics and K-Value Analysis Task: 1. Generate a synthetic regression dataset (sklearn.datasets.make_regression(n_samples=500, n_features=10)). 2. Train a KNN regressor with: a. Euclidean distance (K=5) b. Manhattan distance (K=5) c. Compare Mean Squared Error (MSE) for both. 3. Test K=1, 5, 10, 20, 50 and plot K vs. MSE to analyze bias-variance tradeoff.

ANSWER 9 : Here’s the complete **Python program** for the KNN Regressor with distance metrics and K-value analysis 👇

```
# KNN Regressor with Distance Metrics and K-Value Analysis

# -------------------------------------------------------


from sklearn.datasets import make_regression

from sklearn.model_selection import train_test_split

from sklearn.neighbors import KNeighborsRegressor

from sklearn.metrics import mean_squared_error

import matplotlib.pyplot as plt


# 1. Generate synthetic regression dataset

X, y = make_regression(n_samples=500, n_features=10, noise=10, random_state=42)


# Split into training and testing sets

X_train, X_test, y_test = train_test_split(X, y, test_size=0.3, random_state=42)


# 2a. Train KNN Regressor with Euclidean distance (K=5)

knn_euclidean = KNeighborsRegressor(n_neighbors=5, metric='euclidean')

knn_euclidean.fit(X_train, y_train)

y_pred_euclidean = knn_euclidean.predict(X_test)

mse_euclidean = mean_squared_error(y_test, y_pred_euclidean)


# 2b. Train KNN Regressor with Manhattan distance (K=5)

knn_manhattan = KNeighborsRegressor(n_neighbors=5, metric='manhattan')
```

```
knn_manhattan.fit(X_train, y_train)

y_pred_manhattan = knn_manhattan.predict(X_test)

mse_manhattan = mean_squared_error(y_test, y_pred_manhattan)


print("=== Distance Metric Comparison (K=5) ===")

print(f"Euclidean Distance MSE : {mse_euclidean:.3f}")

print(f"Manhattan Distance MSE : {mse_manhattan:.3f}")


# 3. Analyze bias-variance tradeoff by varying K

k_values = [1, 5, 10, 20, 50]

mse_values = []


for k in k_values:

    knn = KNeighborsRegressor(n_neighbors=k, metric='euclidean')

    knn.fit(X_train, y_train)

    y_pred = knn.predict(X_test)

    mse = mean_squared_error(y_test, y_pred)

    mse_values.append(mse)


# Plot K vs. MSE

plt.figure(figsize=(8, 5))

plt.plot(k_values, mse_values, marker='o', linestyle='-', color='blue')

plt.title("K vs Mean Squared Error (Euclidean Distance)")

plt.xlabel("Number of Neighbors (K)")
plt.ylabel("Mean Squared Error")

plt.grid(True)

plt.show()


# Interpretation

print("\nInterpretation:")

print("- Lower K (e.g., K=1) → Low bias, high variance (fits noise).")

print("- Higher K (e.g., K=50) → High bias, low variance (smoother predictions).")

print("- Optimal K usually balances bias and variance.")
```

✅ Explanation:

1. **Dataset Generation**
    - Uses `make_regression()` to create a synthetic regression dataset (500 samples, 10 features).
2. **Distance Metrics Comparison**
    - Trains two `KNeighborsRegressor` models:
        - **Euclidean distance:** default metric for continuous features.
        - **Manhattan distance:** uses absolute differences.
    - Compares **Mean Squared Error (MSE)** for both.
3. **Bias–Variance Analysis**
    - Tests `K = [1, 5, 10, 20, 50]` and plots **K vs. MSE**.
    - Demonstrates how increasing `K` increases bias but reduces variance.