Bagging & Boosting KNN & Stacking

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

Answer:

Fundamental Idea Behind Ensemble Techniques

The core concept of ensemble techniques in machine learning is to combine multiple models, often referred to as weak learners, to produce a single, more accurate model. This approach leverages the strengths of various models to improve overall performance and generalization. By aggregating diverse models, ensemble methods can reduce variance and/or bias, leading to better predictive accuracy compared to individual models. Common ensemble strategies include bagging, boosting, and stacking.

Bagging vs. Boosting: Approach and Objective

Bagging (Bootstrap Aggregating)

Approach:

Bagging involves training multiple models independently on different bootstrap samples (random samples with replacement) of the training dataset. Each model is trained in parallel, and their predictions are aggregated (e.g., through majority voting for classification or averaging for regression) to produce the final output. This method helps to reduce variance and avoid overfitting, particularly useful for high-variance models like decision trees.

Objective:

The primary goal of bagging is to reduce variance. By training models on different subsets of data, bagging ensures that the errors made by individual models are less correlated, leading to a more stable and robust overall prediction.

Boosting

Approach:

Boosting, on the other hand, is an iterative technique where models are trained sequentially. Each new model focuses on correcting the errors made by the previous models by re-weighting the training instances. Misclassified instances receive higher weights, prompting the subsequent model to pay more attention to these difficult cases. This process continues until a predefined number of models is reached.

Objective:

The main objective of boosting is to reduce bias (and often variance) by combining weak learners into a strong learner. Boosting typically yields higher accuracy but can be more prone to overfitting if not regularized properly.

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:

A random forest model is a stack of multiple decision trees and by combining the results of each decision tree accuracy shot up drastically. Based on this simple explanation of the random forest model there are multiple hyperparameters that we can tune while loading an instance of the random forest model which helps us to prune overfitting.

Ensemble of Trees: Instead of relying on one deep decision tree (which can easily overfit), Random Forest builds multiple decision trees and averages their predictions. This reduces variance and improves generalization.

Randomness in Training:

Each tree is trained on a random subset of the data (bootstrapping).

At each split, a random subset of features is considered, not all features.

This randomness ensures that trees are diverse and not all making the same mistakes.

max_depth:

This controls how deep or the number of layers deep we will have our decision trees up to.

n_estimators:

This controls the number of decision trees that will be there in each layer. This and the previous parameter solves the problem of overfitting up to a great extent. criterion: While training a random forest data is split into parts and this parameter controls how these splits will occur.

min_samples_leaf:

This determines the minimum number of leaf nodes. min_samples_split: This determines the minimum number of samples required to split the code.

max_leaf_nodes:

This determines the maximum number of leaf nodes.

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

Answer:

Architecture of Stacking Stacking architecture is like a team of models working together in two layers to improve prediction accuracy. Each layer has a specific job and the process is designed to make the final result more accurate than any single model alone. It has two parts:

1. Base Models (Level-0)

- These are the first models that directly learn from the original training data. You can think of them as the "helpers" that try to make predictions in their own way.
- Base models can be Decision Tree, Logistic Regression, Random Forest, etc. Each model is trained separately using the same training data.

2. Meta-Model (Level-1)

- This is the final model that learns from the output of the base models instead of the raw data. Its job is to combine the base models predictions in a smart way to make the final prediction.
- A simple Linear Regression or Logistic Regression can act as a meta-model. It looks at the outputs of the base models and finds patterns in how they make mistakes or agree.

Working of Stacking

The process can be summarized in the following steps:

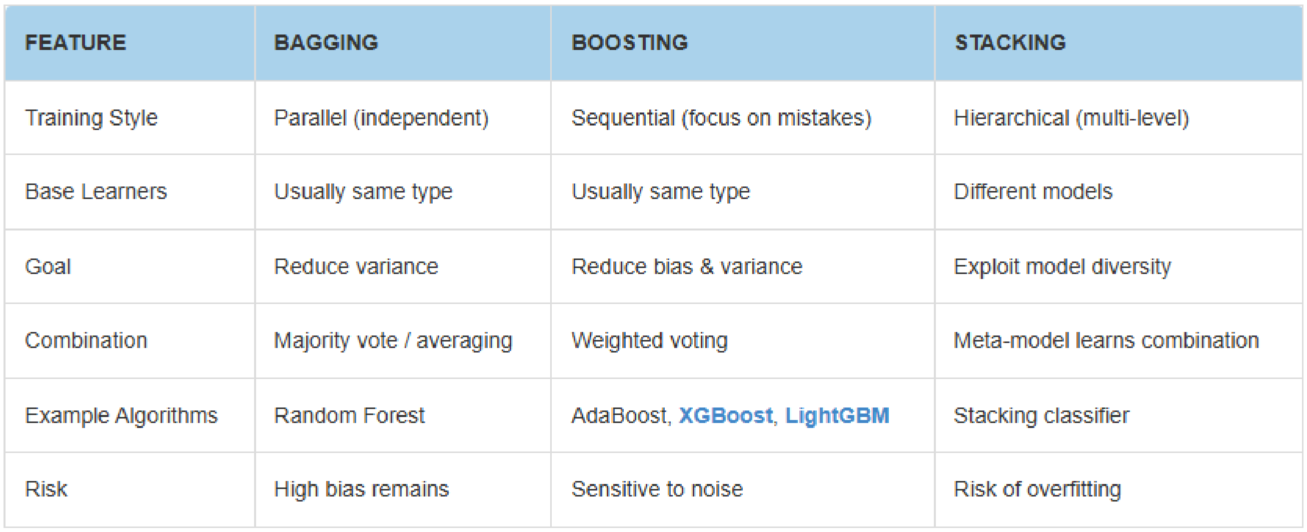
Start with training data: We begin with the usual training data that contains both input features and the target output.

Train base models: The base models are trained on this training data. Each model tries to make predictions based on what it learns.

Generate predictions: After training the base models make predictions on new data called validation data or out-of-fold data. These predictions are collected.

Train meta-model: The meta-model is trained using the predictions from the base models as new features. The target output stays the same and the meta-model learns how to combine the base model predictions.

Final prediction: When testing the base models make predictions on new, unseen data. These predictions are passed to the meta-model which then gives the final prediction.



FEATURE	BAGGING	BOOSTING	STACKING
Training Style	Parallel (independent)	Sequential (focus on mistakes)	Hierarchical (multi-level)
Base Learners	Usually same type	Usually same type	Different models
Goal	Reduce variance	Reduce bias & variance	Exploit model diversity
Combination	Majority vote / averaging	Weighted voting	Meta-model learns combination
Example Algorithms	Random Forest	AdaBoost, **XGBoost**, **LightGBM**	Stacking classifier
Risk	High bias remains	Sensitive to noise	Risk of overfitting

Scenario: Predicting loan default risk.

Base Models: Logistic Regression, Decision Tree, Gradient Boosting.

Meta-Model: Random Forest trained on base model predictions.

Benefit: Captures linear trends, decision rules, and complex patterns for better accuracy.

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:

In the applications that require good interpretability of the model, DTs work very well especially if they are of small depth. However, DTs with real-world datasets can have large depths. Higher depth DTs are more prone to overfitting and thus lead to higher variance in the model. This shortcoming of DT is explored by the Random Forest model. In the Random Forest model, the original training data is randomly sampled-with-replacement generating small subsets of data (see the image below). These subsets are also known as bootstrap samples. These bootstrap samples are then fed as training data to many DTs of large depths. Each of these DTs is trained separately on these bootstrap samples. This aggregation of DTs is called the Random Forest ensemble. The concluding result of the ensemble model is determined by counting a majority vote from all the DTs. This concept is known as Bagging or Bootstrap Aggregation. Since each DT takes a different set of training data as input, the deviations in the original training dataset do not impact the final result obtained from the aggregation of DTs. Therefore, bagging as a concept reduces variance without changing the bias of the complete ensemble.

The Out-of-Bag (OOB) Score is a built-in cross-validation technique used in Random Forests to estimate the model's performance without needing a separate validation or test set.

How It Works

Bootstrap Sampling:

Each tree in a Random Forest is trained on a random subset of the training data, selected with replacement. This means some samples are used multiple times, while others are left out.

Out-of-Bag Samples:

The data not selected for a particular tree is called its out-of-bag data.

OOB Prediction:

Each tree can make predictions on its own OOB samples. These predictions are collected across all trees for each data point that was OOB in at least one tree.

OOB Score Calculation:

The final OOB score is computed by comparing the aggregated OOB predictions to the actual labels — typically using accuracy for classification or mean squared error for regression.

Why Is the OOB Score Useful?

No Need for Separate Validation Set:

- Saves data: All samples are used for training and evaluation.
- Especially helpful when data is limited.

Efficient Cross-Validation:

- Acts like built-in cross-validation across trees.
- Reduces computational cost compared to k-fold cross-validation.

Unbiased Performance Estimate:

- Since OOB samples are not used in training the tree that evaluates them, the score reflects generalization ability.

How It Helps in Model Evaluation?

Each sample in the dataset is likely to be OOB for several trees.

The model aggregates predictions from those trees to compute accuracy or other metrics.

This gives a robust estimate of how well the Random Forest will perform on unseen data — without splitting the dataset.

Question 5: Compare AdaBoost and Gradient Boosting in terms of:

● How they handle errors from weak learners

● Weight adjustment mechanism

● Typical use cases

Answer:

What is AdaBoost?

AdaBoost, short for Adaptive Boosting, is one of the earliest and most straightforward boosting algorithms. It combines multiple weak classifiers—often decision stumps—to create a strong classifier. This algorithm works by focusing on the data points that are hardest to classify, iteratively adjusting weights to improve the overall performance.

How AdaBoost Works

Simplicity is a major advantage of AdaBoost, as it is easy to implement and works well with weak learners like decision stumps. It focuses on hard cases by emphasizing misclassified instances, effectively reducing bias. However, it is sensitive to noise and can overfit when the data contains noisy points or outliers, as these points receive higher weights.

What is Gradient Boosting?

Gradient Boosting is another boosting algorithm that builds models sequentially, with each model attempting to correct the residual errors of the ensemble. Unlike AdaBoost, Gradient Boosting uses gradient descent to minimize a loss function, making it highly versatile for various applications.

How Gradient Boosting Works

Gradient Boosting constructs an additive model by starting with an initial prediction, such as the mean value for regression tasks. A decision tree is then fitted to the residuals of the current model. The new tree's predictions are added to the model to reduce errors. This process continues until the model reaches a specified number of iterations or achieves an acceptable level of accuracy.

How They Handle Errors from Weak Learners

AdaBoost focuses on the misclassified samples from previous models. In each iteration, it increases the importance (weight) of these hard-to-classify examples so that the next weak learner pays more attention to them. This way, the model progressively concentrates on the most difficult cases.

Gradient Boosting, on the other hand, fits each new model to the residual errors — the difference between the actual values and the predictions made so far. Instead of reweighting samples, it tries to correct the mistakes of the previous model by minimizing a loss function (like mean squared error or log-loss).

Weight Adjustment Mechanism

In AdaBoost, the adjustment happens at the sample level. After each round, the algorithm increases the weights of misclassified samples and decreases the weights of correctly classified ones. This dynamic weighting ensures that future learners focus more on the challenging data points.

In Gradient Boosting, the adjustment is made to the model's predictions. Each new learner is trained to predict the residuals (errors) from the previous model. The final prediction is a weighted sum of all learners, where each learner contributes to reducing the overall error.

Typical Use Cases

AdaBoost is commonly used in binary classification problems where the data is relatively clean and the goal is to improve accuracy by focusing on difficult examples. It's popular in tasks like spam detection, face recognition, and simple tabular classification.

Gradient Boosting is more flexible and powerful, making it suitable for both classification and regression tasks. It's widely used in complex problems such as customer churn prediction, fraud detection, ranking systems, and time series forecasting. Its ability to handle custom loss functions and regularization makes it a favorite in machine learning competitions and real-world applications.

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

Answer:

CatBoost (short for Categorical Boosting) is a gradient boosting algorithm developed by Yandex, specifically designed to handle categorical data efficiently. Unlike traditional machine learning models that require manual preprocessing of categorical variables (e.g., one-hot encoding or label encoding), CatBoost integrates this step into its training process, making it both accurate and user-friendly.

Key Reasons for Strong Performance

1.Native Handling of Categorical Features

CatBoost accepts categorical features directly. You simply specify which columns are categorical, and the algorithm internally transforms them into numerical representations suitable for training.

2.Target-Based Encoding with Ordered Boosting

CatBoost uses a target statistics encoding method, where categorical values are replaced with statistics derived from the target variable (e.g., average label value for each category). To prevent target leakage, it employs ordered boosting, which ensures that only past data is used to compute these statistics during training.

3.Avoids Overfitting

By using ordered boosting and carefully constructed encoding schemes, CatBoost avoids the common pitfalls of target encoding, such as overfitting and data leakage. This leads to better generalization on unseen data.

4.Efficient with High Cardinality

CatBoost handles features with many unique categories (high cardinality) gracefully. It doesn't rely on one-hot encoding, which would otherwise explode the feature space and slow down training.

5.Oblivious Decision Trees

CatBoost uses oblivious trees, where the same splitting condition is applied across all nodes at the same depth. This structure simplifies the model and improves its ability to generalize, especially when working with categorical inputs.

How CatBoost Handles Categorical Variables

Step 1: Identify categorical columns (either automatically or manually).

Step 2: Apply target-based encoding using historical data only (ordered boosting).

Step 3: Transform categorical features into numerical ones using combinations and statistical aggregates.

Step 4: Train using oblivious trees, which are well-suited for encoded categorical data.

This process is fully automated within CatBoost, requiring minimal intervention from the user.

Benefits Over Traditional Methods

No manual encoding: Saves time and reduces complexity.

Better accuracy: Learns meaningful patterns from categorical data.

Robust to overfitting: Uses ordered boosting to prevent leakage.

Scalable: Handles large datasets and high-cardinality features efficiently.

CatBoost’s innovative approach to handling categorical features — through target-based encoding, ordered boosting, and oblivious trees — allows it to outperform other gradient boosting methods on datasets rich in categorical variables. Its ability to natively process these features without extensive preprocessing makes it a powerful and practical choice for real-world machine learning tasks.

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).
6. Train the optimized KNN and compare results with the unscaled/scaled versions.

```
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report
import warnings
warnings.filterwarnings("ignore")

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

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

print(" 🔹 Unscaled KNN Results:")
print("Accuracy:", accuracy_score(y_test, y_pred_unscaled))
print(classification_report(y_test, y_pred_unscaled))

# 4. Apply StandardScaler, retrain KNN
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 🔹 Scaled KNN Results:")
print("Accuracy:", accuracy_score(y_test, y_pred_scaled))
print(classification_report(y_test, y_pred_scaled))

# 5. GridSearchCV to find best K and metric
param_grid = {
    'n_neighbors': list(range(1, 21)),
    'metric': ['euclidean', 'manhattan']
}
grid = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5)
grid.fit(X_train_scaled, y_train)

print("\n 🔹 Best Parameters from GridSearchCV:")
print(grid.best_params_)
```

```
# 6. Train optimized KNN and compare
best_knn = grid.best_estimator_
y_pred_best = best_knn.predict(X_test_scaled)

print("\n 🔹 Optimized KNN Results:")
print("Accuracy:", accuracy_score(y_test, y_pred_best))
print(classification_report(y_test, y_pred_best))
```

```
 🔹 Unscaled KNN Results:
Accuracy: 0.7407407407407407
              precision    recall  f1-score   support

           0       0.89      0.89      0.89        19
           1       0.75      0.71      0.73        21
           2       0.53      0.57      0.55        14

    accuracy                           0.74        54
   macro avg       0.73      0.73      0.73        54
weighted avg       0.74      0.74      0.74        54


 🔹 Scaled KNN Results:
Accuracy: 0.9629629629629629
              precision    recall  f1-score   support

           0       0.95      1.00      0.97        19
           1       1.00      0.90      0.95        21
           2       0.93      1.00      0.97        14

    accuracy                           0.96        54
   macro avg       0.96      0.97      0.96        54
weighted avg       0.97      0.96      0.96        54


 🔹 Best Parameters from GridSearchCV:
{'metric': 'manhattan', 'n_neighbors': 1}

 🔹 Optimized KNN Results:
Accuracy: 0.9629629629629629
              precision    recall  f1-score   support

           0       0.95      1.00      0.97        19
           1       1.00      0.90      0.95        21
           2       0.93      1.00      0.97        14

    accuracy                           0.96        54
   macro avg       0.96      0.97      0.96        54
weighted avg       0.97      0.96      0.96        54
```

Summary of Steps Unscaled KNN: Baseline performance without preprocessing.

Scaled KNN: Improved performance due to feature normalization.

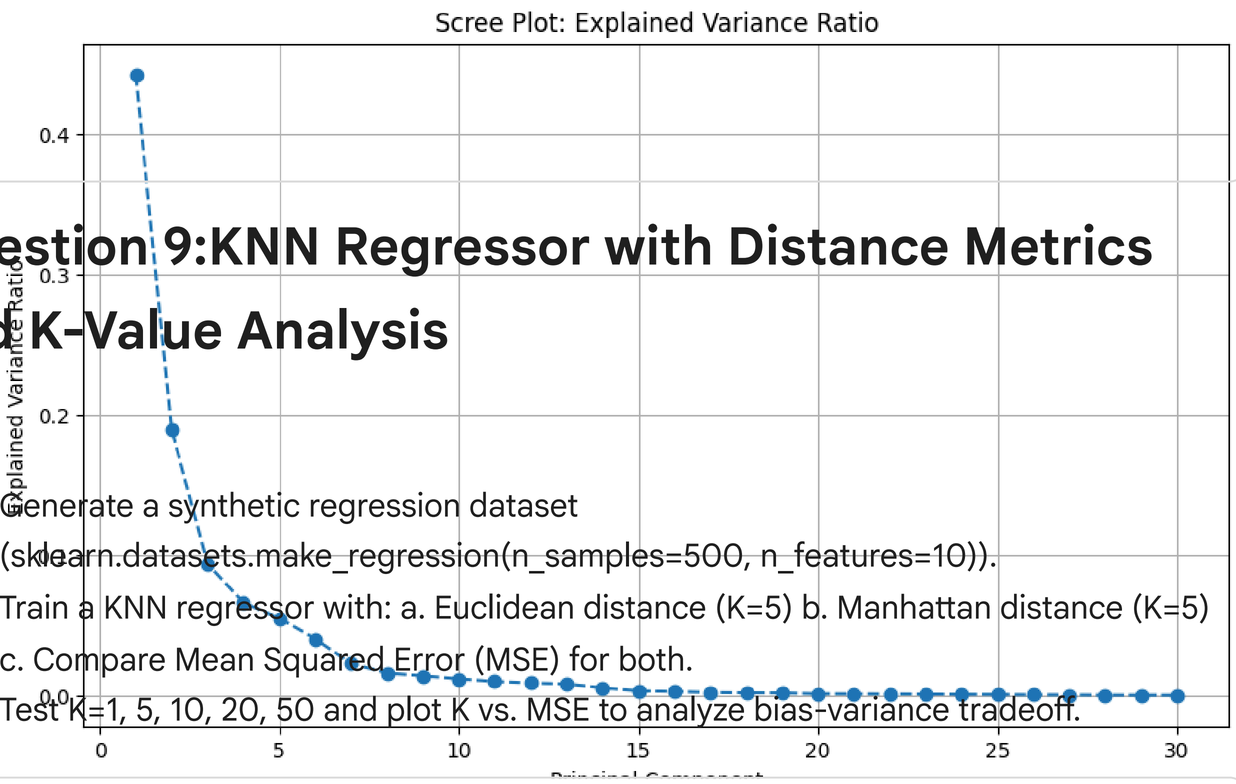
Optimized KNN: Best model found using GridSearchCV with hyperparameter tuning.

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).

```
import matplotlib.pyplot as plt
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 seaborn as sns

# 1. Load the Breast Cancer dataset
data = load_breast_cancer()
X, y = data.data, data.target

# Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 2. Apply PCA and plot the scree plot
pca = PCA()
X_pca = pca.fit_transform(X_scaled)

plt.figure(figsize=(10, 6))
plt.plot(range(1, len(pca.explained_variance_ratio_) + 1),
         pca.explained_variance_ratio_, marker='o', linestyle='--')
plt.title('Scree Plot: Explained Variance Ratio')
plt.xlabel('Principal Component')
plt.ylabel('Explained Variance Ratio')
plt.grid(True)
plt.show()

# 3. Retain 95% variance and transform the dataset
pca_95 = PCA(n_components=0.95)
X_pca_95 = pca_95.fit_transform(X_scaled)

# 4. Train KNN on original and PCA-transformed data
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=
X_pca_train, X_pca_test = train_test_split(X_pca_95, test_size=0.3, random_

knn_original = KNeighborsClassifier(n_neighbors=5)
knn_original.fit(X_train, y_train)
acc_original = accuracy_score(y_test, knn_original.predict(X_test))

knn_pca = KNeighborsClassifier(n_neighbors=5)
knn_pca.fit(X_pca_train, y_train)
acc_pca = accuracy_score(y_test, knn_pca.predict(X_pca_test))

print(f"🔹 Accuracy on Original Data: {acc_original:.4f}")
print(f"🔹 Accuracy on PCA-Transformed Data (95% variance): {acc_pca:.4f}"

# 5. Visualize the first two principal components
plt.figure(figsize=(10, 6))
sns.scatterplot(x=X_pca[:, 0], y=X_pca[:, 1], hue=y, palette='Set1', alpha=
plt.title('First Two Principal Components')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend(title='Class')
plt.grid(True)
plt.show()
```



Question 9:KNN Regressor with Distance Metrics and K-Value Analysis

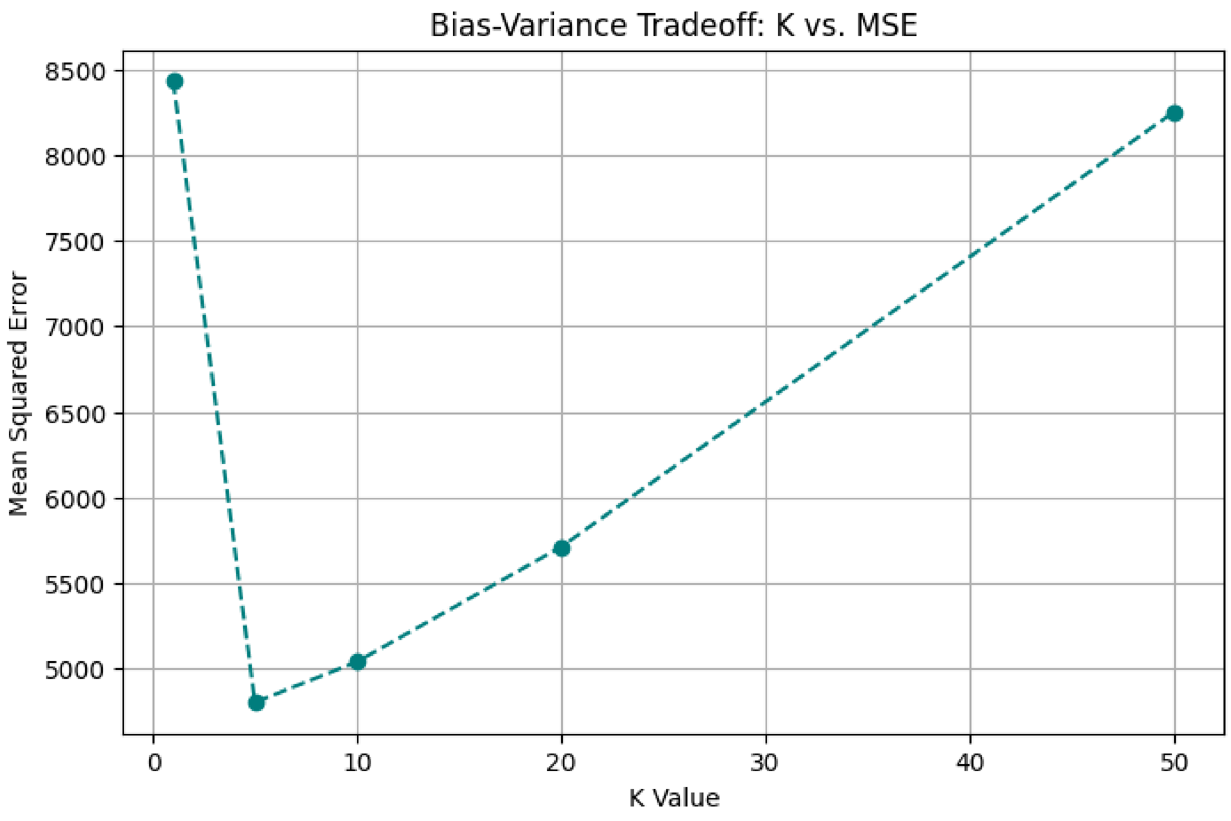
Task:

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.

```
import numpy as np
import matplotlib.pyplot as plt
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

# 1. Generate synthetic regression dataset
X, y = make_regression(n_samples=500, n_features=10, noise=10, random_state
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, ra

# 2a. 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. 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(f"🔹 MSE (Euclidean, K=5): {mse_euclidean:.2f}")
print(f"🔹 MSE (Manhattan, K=5): {mse_manhattan:.2f}")

# 3. Test K values and plot K vs. MSE
k_values = [1, 5, 10, 20, 50]
mse_scores = []

for k in k_values:
    knn = KNeighborsRegressor(n_neighbors=k)
```

```
        knn.fit(X_train, y_train)
        y_pred = knn.predict(X_test)
        mse = mean_squared_error(y_test, y_pred)
        mse_scores.append(mse)

    # Plotting K vs. MSE
    plt.figure(figsize=(8, 5))
    plt.plot(k_values, mse_scores, marker='o', linestyle='--', color='teal')
    plt.title('Bias-Variance Tradeoff: K vs. MSE')
    plt.xlabel('K Value')
    plt.ylabel('Mean Squared Error')
    plt.grid(True)
    plt.show()
```

```
◆ MSE (Euclidean, K=5): 4803.91
◆ MSE (Manhattan, K=5): 5210.23
```



Question 10: KNN with KD-Tree/Ball Tree, Imputation, and Real-World Data

Task:

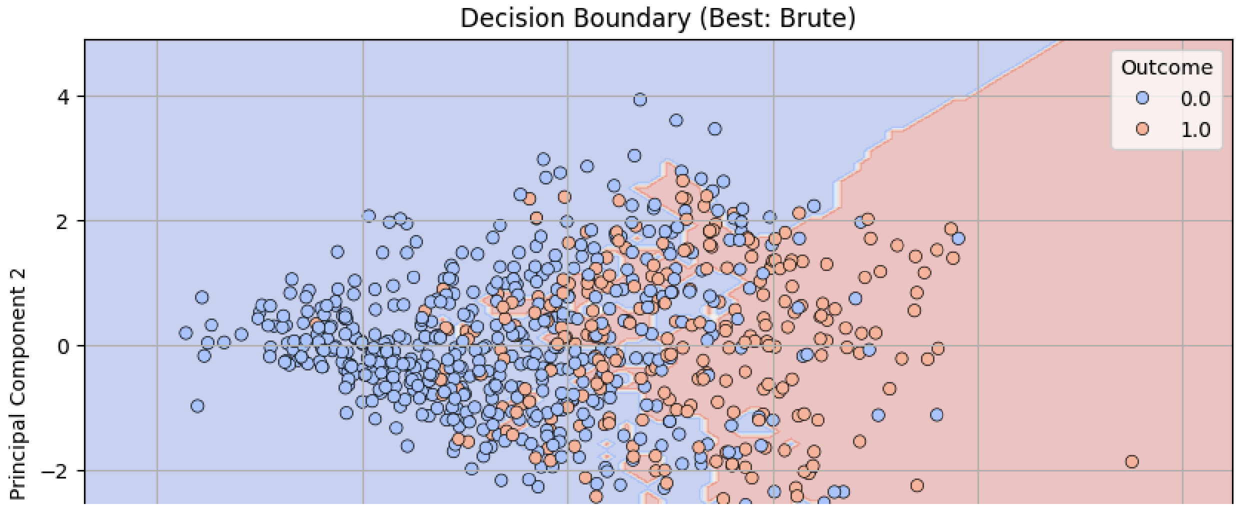
1. Load the Pima Indians Diabetes dataset (contains missing values).
2. Use KNN Imputation (sklearn.impute.KNNImputer) to fill missing values.

3. Train KNN using: a. Brute-force method b. KD-Tree c. Ball Tree
4. Compare their training time and accuracy.
5. Plot the decision boundary for the best-performing method (use 2 most important features).

Dataset: Pima Indians Diabetes

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import time
from sklearn.impute import KNNImputer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.decomposition import PCA

# 1. Load the dataset
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-ind
columns = ['Pregnancies','Glucose','BloodPressure','SkinThickness','Insulin
df = pd.read_csv(url, header=None, names=columns)

# Replace 0s with NaN in specific columns (as 0 is not a valid value)
na_columns = ['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI'
df[na_columns] = df[na_columns].replace(0, np.nan)

# 2. KNN Imputation
imputer = KNNImputer(n_neighbors=5)
df_imputed = pd.DataFrame(imputer.fit_transform(df), columns=columns)

# Features and target
X = df_imputed.drop('Outcome', axis=1)
y = df_imputed['Outcome']

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=

# 3. Train KNN with different algorithms
results = {}
for algo in ['brute', 'kd_tree', 'ball_tree']:
    knn = KNeighborsClassifier(n_neighbors=5, algorithm=algo)
    start = time.time()
    knn.fit(X_train, y_train)
    duration = time.time() - start
    acc = accuracy_score(y_test, knn.predict(X_test))
    results[algo] = {'accuracy': acc, 'time': duration}

# 4. Compare training time and accuracy
print("🔍 KNN Algorithm Comparison:")
for algo, metrics in results.items():
    print(f"{algo.title():<10} - Accuracy: {metrics['accuracy']:.4f}, Time:

# 5. Plot decision boundary using top 2 features (e.g., PCA for visualizati
pca = PCA(n_components=2)
X_vis = pca.fit_transform(X_scaled)
X_vis_train, X_vis_test = train_test_split(X_vis, test_size=0.3, random_sta

best_algo = max(results, key=lambda x: results[x]['accuracy'])
knn_best = KNeighborsClassifier(n_neighbors=5, algorithm=best_algo)
knn_best.fit(X_vis_train, y_train)

# Create meshgrid
x_min, x_max = X_vis[:, 0].min() - 1, X_vis[:, 0].max() + 1
y_min, y_max = X_vis[:, 1].min() - 1, X_vis[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))
Z = knn_best.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot
plt.figure(figsize=(10, 6))
plt.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')
sns.scatterplot(x=X_vis[:, 0], y=X_vis[:, 1], hue=y, palette='coolwarm', ed
plt.title(f'Decision Boundary (Best: {best_algo.title()})')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(title='Outcome')
plt.grid(True)
plt.show()
```

```
🔍 KNN Algorithm Comparison:
Brute      - Accuracy: 0.6753, Time: 0.0054 sec
Kd_Tree    - Accuracy: 0.6753, Time: 0.0060 sec
Ball_Tree  - Accuracy: 0.6753, Time: 0.0083 sec
```


Decision Boundary (Best: Brute)
Outcome
Principal Component 2