Bagging & Boosting, KNN & Stacking — Answers

Question 1: Fundamental idea behind ensemble techniques; bagging vs boosting.

Fundamental idea:
Ensemble techniques combine multiple base learners (models) to produce a single, usually stronger, predictive model. The core intuition is that different models make different errors; by aggregating them (averaging, voting, or stacking), we can reduce variance, bias, or both, and improve generalization compared to any single model.

Bagging vs Boosting (approach & objective):
Bagging (Bootstrap Aggregating):
- Approach: Train multiple base learners (usually identical algorithms, e.g., decision trees) in parallel on different bootstrap samples (random samples with replacement) drawn from the training data. Final prediction is by averaging (regression) or majority vote (classification).
- Objective: Reduce variance and avoid overfitting of high-variance models. Each model is independent and contributes equally.

Boosting:
- Approach: Train base learners sequentially. Each learner tries to correct mistakes made by the previous learners by giving more weight to misclassified or high-error instances. Predictions are combined using a weighted sum (or vote).
- Objective: Reduce bias (and sometimes variance) by focusing on hard examples and building a strong learner from weak learners. Models are dependent and weighted by their performance.

Key practical differences:
- Bagging is parallel and helpful with high-variance models (e.g., deep trees). Boosting is sequential and can reduce bias and improve weak learners (e.g., shallow trees).
- Bagging is more robust to noise; boosting can overfit noisy labels if not regularized.
- Popular algorithms: Bagging -> Random Forest; Boosting -> AdaBoost, Gradient Boosting, XGBoost, LightGBM, CatBoost.

Question 2: How Random Forest reduces overfitting vs single tree; role of two hyperparameters.

Explanation:
A single decision tree typically overfits because it can grow deep and learn noise. Random Forest reduces overfitting by creating an ensemble of many decision trees trained on different random samples and feature subsets; averaging their predictions smooths out individual tree errors and reduces variance.

Two key hyperparameters and their roles:
1) n_estimators (number of trees): Increasing the number of trees reduces variance (the ensemble becomes more stable) and usually improves generalization up to a point.
2) max_features (number of features to consider when splitting): By selecting a random subset of features at each split, trees become decorrelated — each tree sees different information, reducing the chance that all trees make the same mistake. Typical settings: sqrt(n_features) for classification, n_features/3 for regression.

Other useful hyperparameters: max_depth (limits tree depth to reduce overfitting) and min_samples_leaf/min_samples_split (prevent overly specific splits).

Question 3: What is Stacking? How differs from bagging/boosting? Example use case.

Definition and mechanism:
Stacking (stacked generalization) is an ensemble technique that trains multiple base (level-0) models and then trains a meta-model (level-1) on the base models' out-of-fold predictions. The meta-model learns how to best combine the base models' strengths.

How it differs from bagging/boosting:
- Bagging combines base learners by simple averaging or majority voting and trains them in parallel on bootstrapped data. Boosting trains learners sequentially with reweighting to correct errors.
- Stacking trains base learners (can be heterogeneous: trees, linear models, SVMs, etc.) and uses their

predictions as input features to a higher-level learner. The meta-learner can learn complex combinations rather than simple averages.

Simple example use case:
- Level-0: Train logistic regression, random forest, and an SVM on training data using cross-validated predictions.
- Level-1: Train a lightweight model (e.g., logistic regression) on the cross-validated predictions of these models to output final predictions. This often improves performance when base learners have complementary strengths.

Question 4: OOB Score in Random Forest and usefulness.

Out-of-Bag (OOB) score:
When training each tree in a Random Forest, a bootstrap sample (sampling with replacement) is drawn from the training set. On average, about 63% of samples are included in a bootstrap sample, leaving ~37% not included — these are called out-of-bag samples for that tree.

Usefulness and model evaluation without a separate validation set:
- For each training instance, predictions from trees for which that instance was OOB can be aggregated to form an OOB prediction for that instance.
- Comparing OOB predictions to true labels yields an OOB error (or OOB score) that estimates generalization performance without needing a separate validation set or cross-validation.
- OOB is computationally efficient and particularly handy when data is limited; however, for fine-grained model selection, cross-validation may still be preferred.

Question 5: Compare AdaBoost and Gradient Boosting (errors handling, weight adjustment, use cases).

Comparison:
1) How they handle errors from weak learners:
- AdaBoost: Adjusts the weights of training samples — after each weak learner, misclassified samples get higher weights so subsequent learners focus on them. The final model is a weighted sum of weak learners where weights reflect learner accuracy.
- Gradient Boosting: Views boosting as gradient descent in function space. Each new learner is fit to the negative gradient (residuals) of a loss function of the ensemble so far (e.g., residuals for least squares). It corrects errors by directly modeling residuals rather than reweighting instances.

2) Weight adjustment mechanism:
- AdaBoost: Explicit sample reweighting; sample weights are updated multiplicatively based on whether they were correctly classified and the learner's error.
- Gradient Boosting: No explicit sample weights (though sample weights can be used); it fits learners to residuals or pseudo-residuals computed from gradients of the loss.

3) Typical use cases:
- AdaBoost: Historically used with simple weak learners (stumps). Works well on clean data and problems where misclassification focus helps. Less commonly used than gradient methods today.
- Gradient Boosting (and its implementations: XGBoost, LightGBM, CatBoost): Very popular for structured/tabular data, flexible loss functions, handles regression/classification/ranking, often top performers in ML competitions. Use when you need high predictive accuracy and are willing to tune hyperparameters.

Question 6: Why CatBoost performs well on categorical features; brief handling explanation.

Why CatBoost handles categorical features well:
- CatBoost uses ordered target statistics and combinations of categorical features with careful permutation-driven schemes to avoid target leakage and overfitting.

Brief explanation:
- Instead of naive one-hot encoding, CatBoost computes category encodings based on target statistics (e.g., mean target value for each category) but uses permutations and 'ordered' boosting to ensure the statistic for a given instance is computed using only prior data in the permutation — preventing leakage.
- It can automatically handle high-cardinality categorical features, build combinations of categorical variables, and

use specialized hashing/encoding internally. This reduces the need for manual preprocessing and often improves performance on datasets with many categorical variables.

Question 7: KNN Classifier Assignment — Wine dataset (procedural answer and example code).

Solution outline and code template (Python / scikit-learn):

1) Load dataset and split:
- from sklearn.datasets import load_wine
- X, y = load_wine(return_X_y=True)
- use train_test_split with test_size=0.3, random_state=42

2) Train KNN (default K=5) without scaling and evaluate:
- from sklearn.neighbors import KNeighborsClassifier
- model = KNeighborsClassifier() ; model.fit(X_train, y_train)
- get accuracy, classification_report from sklearn.metrics

3) Apply StandardScaler, retrain and compare metrics.

4) GridSearchCV for best K and metric:
- param_grid = {'n_neighbors': range(1,21), 'metric': ['euclidean','manhattan']}
- GridSearchCV(KNeighborsClassifier(), param_grid, cv=5, scoring='accuracy')

5) Train optimized KNN (best params) on training set and evaluate on test set and compare results.

Code snippet (run locally):

```
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

X, y = load_wine(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)

# Without scaling
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
print('Accuracy (no scaling):', accuracy_score(y_test, knn.predict(X_test)))
print(classification_report(y_test, knn.predict(X_test)))

# With scaling
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)
knn_s = KNeighborsClassifier()
knn_s.fit(X_train_s, y_train)
print('Accuracy (with scaling):', accuracy_score(y_test, knn_s.predict(X_test_s)))
print(classification_report(y_test, knn_s.predict(X_test_s)))

# Grid search
param_grid = {'n_neighbors': list(range(1,21)), 'metric': ['euclidean','manhattan']}
grid = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5, scoring='accuracy')
grid.fit(X_train_s, y_train)
print('Best params:', grid.best_params_)

best_knn = grid.best_estimator_
print('Test acc (best):', accuracy_score(y_test, best_knn.predict(X_test_s)))
```

Question 8: PCA + KNN with variance analysis & visualization (outline and code).

Solution outline and code template:

1) Load breast cancer dataset: from sklearn.datasets import load_breast_cancer
2) Standardize features (recommended before PCA).
3) Apply PCA, plot scree plot (explained variance ratio).
4) Retain components that explain 95% variance (use PCA(n_components=0.95)).
5) Train KNN on original scaled data and on PCA-transformed data and compare accuracies.
6) Scatter plot first two principal components colored by class.

Code snippet (run locally):

```
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

X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)

pca = PCA()
pca.fit(X_train_s)
explained = pca.explained_variance_ratio_
# Scree plot
plt.plot(range(1, len(explained)+1), explained.cumsum())
plt.xlabel('Number of components')
plt.ylabel('Cumulative explained variance')
plt.show()

# Retain 95% variance
pca95 = PCA(n_components=0.95)
X_train_p = pca95.fit_transform(X_train_s)
X_test_p = pca95.transform(X_test_s)

# KNN on original scaled data
knn = KNeighborsClassifier()
knn.fit(X_train_s, y_train)
print('Accuracy (original):', accuracy_score(y_test, knn.predict(X_test_s)))

# KNN on PCA data
knn_p = KNeighborsClassifier()
knn_p.fit(X_train_p, y_train)
print('Accuracy (PCA 95%):', accuracy_score(y_test, knn_p.predict(X_test_p)))

# Scatter plot first two pcs
pca2 = PCA(n_components=2)
X2 = pca2.fit_transform(scaler.fit_transform(X))
plt.scatter(X2[:,0], X2[:,1], c=y, cmap='viridis', s=10)
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.title('First two principal components')
plt.show()
```

Question 9: KNN Regressor distance metrics and K-value analysis (outline and code).

Solution outline and code template:

1) Generate synthetic dataset: from sklearn.datasets import make_regression
2) Train KNN regressor with Euclidean (p=2) and Manhattan (p=1) distance—compare MSE.
3) Evaluate for K in [1,5,10,20,50] and plot K vs MSE to study bias-variance tradeoff.

Code snippet (run locally):

```
from sklearn.datasets import make_regression
```

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

X, y = make_regression(n_samples=500, n_features=10, noise=10, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Compare euclidean and manhattan at K=5
knn_e = KNeighborsRegressor(n_neighbors=5, p=2)
knn_e.fit(X_train, y_train)
knn_m = KNeighborsRegressor(n_neighbors=5, p=1)
knn_m.fit(X_train, y_train)

mse_e = mean_squared_error(y_test, knn_e.predict(X_test))
mse_m = mean_squared_error(y_test, knn_m.predict(X_test))
print('MSE Euclidean (K=5):', mse_e)
print('MSE Manhattan (K=5):', mse_m)

# K vs MSE
Ks = [1,5,10,20,50]
mses = []
for k in Ks:
model = KNeighborsRegressor(n_neighbors=k)
model.fit(X_train, y_train)
mses.append(mean_squared_error(y_test, model.predict(X_test)))

plt.plot(Ks, mses, marker='o')
plt.xlabel('K')
plt.ylabel('MSE')
plt.title('K vs MSE')
plt.show()
```

Question 10: KNN with KD-Tree/Ball Tree, imputation, and real-world data (outline and code).

Solution outline and code template:

1) Load Pima Indians Diabetes dataset (CSV link provided). Use pandas to read.
2) Identify missing values and apply KNNImputer from sklearn.impute to fill missing values.
3) Train KNN classifier with algorithm='brute', algorithm='kd_tree', algorithm='ball_tree' and compare runtime and accuracy.
4) To measure training time use time.time() around fit calls.
5) For decision boundary, pick two most important features (e.g., after feature importance via random forest or univariate selection) and plot 2D decision boundary of best method.

Code snippet (run locally):

```
import pandas as pd
from sklearn.impute import KNNImputer
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import time

url = 'https://raw.githubusercontent.com/MasteriNeuron/datasets/refs/heads/main/diabetes.csv'
df = pd.read_csv(url)
# Identify columns with zeros as missing if dataset uses 0 for missing (common for Pima)
# e.g., columns = ['Glucose','BloodPressure','SkinThickness','Insulin','BMI']
cols_with_zero = ['Glucose','BloodPressure','SkinThickness','Insulin','BMI']
for c in cols_with_zero:
df[c].replace(0, pd.NA, inplace=True)

imputer = KNNImputer(n_neighbors=5)
df_imputed = pd.DataFrame(imputer.fit_transform(df), columns=df.columns)
```

```
X = df_imputed.drop('Outcome', axis=1)
y = df_imputed['Outcome']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)

results = {}
for algo in ['brute', 'kd_tree', 'ball_tree']:
knn = KNeighborsClassifier(algorithm=algo)
t0 = time.time()
knn.fit(X_train, y_train)
t1 = time.time()
preds = knn.predict(X_test)
acc = accuracy_score(y_test, preds)
results[algo] = {'time': t1-t0, 'accuracy': acc}

print(results)

# For decision boundary: select two features (e.g., 'Glucose' and 'BMI'), train best model and plot meshgrid predictions.
```