Assignment Code: DA-AG-014

Ensemble Learning | **Assignment**

Instructions: Carefully read each question. Use Google Docs, Microsoft Word, or a similar tool to create a document where you type out each question along with its answer. Save the document as a PDF, and then upload it to the LMS. Please do not zip or archive the files before uploading them. Each question carries 20 marks.

Total Marks: 200

Question 1: What is Ensemble Learning in machine learning? Explain the key idea behind it.

Answer:
Ensemble Learning in machine learning is a technique that combines multiple models (called base learners or weak learners) to create a stronger, more accurate model. The key idea is to leverage the collective predictions of these models to improve overall performance, reduce errors, and enhance robustness compared to any single model.

Key Idea:
The core principle of ensemble learning is that by combining diverse models, each with its own strengths and weaknesses, the ensemble can capture a broader range of patterns in the data and mitigate individual model biases or errors. This is often summarized as the "wisdom of the crowd" effect—multiple models working together tend to outperform a single model by averaging out mistakes or emphasizing correct predictions.

How It Works
Ensemble methods typically combine models in one of the following ways:

1. **Voting (Classification)**: For classification tasks, predictions from multiple models are combined, often by majority voting (e.g., the class predicted by most models is chosen).
2. **Averaging (Regression)**: For regression tasks, predictions are averaged (e.g., taking the mean or weighted mean of outputs).
3. **Weighted Combination**: Some models may contribute more to the final prediction based on their performance or reliability.

Common Ensemble Techniques

1. **Bagging (Bootstrap Aggregating)**:
    - Trains multiple models (e.g., decision trees) on different random subsets of the training data (using bootstrap sampling).

- Combines predictions by averaging or voting.
- Example: Random Forest, which uses multiple decision trees to improve accuracy and reduce overfitting.

2. **Boosting**:
   - Builds models sequentially, where each model focuses on correcting the errors of the previous ones by assigning higher weights to misclassified or poorly predicted instances.
   - Example: Gradient Boosting (e.g., XGBoost, LightGBM) and AdaBoost.
3. **Stacking**:
   - Trains multiple models (base learners) and then uses another model (meta-learner) to combine their predictions.
   - The meta-learner learns how to best weigh or combine the base learners' outputs.

Why It Works

- **Diversity**: Different models may excel at capturing different aspects of the data, so combining them leverages their complementary strengths.
- **Error Reduction**: Errors from individual models (e.g., overfitting or underfitting) are often cancelled out or reduced in the ensemble.
- **Robustness**: Ensembles are less sensitive to noise or outliers in the data, making them more stable.

Question 2: What is the difference between Bagging and Boosting?

Answer:

Bagging and Boosting are two popular ensemble learning techniques in machine learning that combine multiple models to improve predictive performance, but they differ in their approach, purpose, and how they handle the models. Below is a clear comparison of the two:

1. Definition:

- Bagging (Bootstrap Aggregating):
  - Bagging involves training multiple independent models (base learners) on different random subsets of the training data, created through bootstrap sampling (sampling with replacement).
  - Predictions are combined by averaging (for regression) or majority voting (for classification).
  - Example: Random Forest.
- Boosting:
  - Boosting builds models sequentially, where each model learns from the mistakes of the previous ones by focusing on misclassified or poorly predicted instances.
  - Predictions are combined through weighted averaging or voting, with weights

often based on model performance.
  - Examples: AdaBoost, Gradient Boosting, XGBoost.

2. Key Differences:

Aspect	Bagging	Boosting
Model Training	Models are trained independently in parallel on random data subsets.	Models are trained sequentially, with each model correcting prior errors.
Data Sampling	Uses bootstrap sampling (random subsets with replacement) to create diversity.	Adjusts weights of data points to focus on misclassified or difficult cases.
Model Dependency	Base learners are independent of each other.	Base learners are dependent, as each learns from the previous model's errors.
Objective	Reduces variance (overfitting) by averaging predictions from diverse models.	Reduces bias and variance by iteratively improving weak learners.
Combining Predictions	Simple averaging (regression) or majority voting (classification).	Weighted averaging or voting, with weights based on model performance.
Robustness to Noise	More robust to noisy data and outliers due to random sampling.	Less robust, as it focuses heavily on hard-to-predict instances, which may include noise.
Overfitting Risk	Less prone to overfitting, especially with diverse models.	More prone to overfitting if not properly tuned, especially with noisy data.
Computational Cost	Can be parallelized, so faster in some cases.	Sequential training, so generally slower unless optimized.
Examples	Random Forest, Bagged Decision Trees.	AdaBoost, Gradient Boosting, XGBoost, LightGBM.

- **Bagging** reduces variance by training independent models on random data subsets and combining their predictions, making it robust and stable (e.g., Random Forest).
- **Boosting** reduces bias and variance by sequentially training models to correct errors, leading to higher accuracy but requiring careful tuning (e.g., XGBoost). Both improve performance over single models, but Bagging emphasizes diversity and stability, while Boosting focuses on iterative improvement and accuracy.

Question 3: What is bootstrap sampling and what role does it play in Bagging methods like Random Forest?

Answer:

Bootstrap sampling is a statistical technique used to create multiple subsets of data by randomly sampling with replacement from the original dataset. In this process:

- Each subset (called a bootstrap sample) is the same size as the original dataset, but some data points may appear multiple times, while others may be omitted.
- For example, if you have a dataset with 100 instances, a bootstrap sample might include some instances twice or thrice and exclude others, still maintaining a size of 100.
- This creates diverse subsets that approximate the variability of the original data, even though they are drawn from the same dataset.

Role of Bootstrap Sampling in Bagging Methods like Random Forest
Bootstrap sampling is a foundational component of **Bagging** (Bootstrap Aggregating), which is used in methods like Random Forest. Its role is to introduce diversity among the base learners (e.g., decision trees) to improve the ensemble's overall performance. Here's how it works and why it's important:

1. **Creating Diverse Training Sets**:
    - In Bagging, multiple models (e.g., decision trees in Random Forest) are trained independently on different bootstrap samples of the training data.
    - Because each bootstrap sample is slightly different (due to sampling with replacement), each model learns from a unique perspective of the data. This diversity reduces the correlation between the models, which is key to improving the ensemble's robustness.
2. **Reducing Variance**:
    - Bagging aims to reduce variance (overfitting) in high-variance models like decision trees, which are sensitive to small changes in the training data.
    - By training each model on a different bootstrap sample, the ensemble averages out the errors or noise specific to individual models. This makes the final prediction more stable and less prone to overfitting compared to a single model.
3. **How It Works in Random Forest**:
    - In Random Forest, a collection of decision trees is built, with each tree trained on a unique bootstrap sample of the training data.
    - Additionally, Random Forest introduces further randomness by selecting a random subset of features at each split in the tree-building process.
    - The bootstrap samples ensure that each tree sees a slightly different version of the data, while the random feature selection ensures diversity in the trees' structures.
    - Final predictions are made by averaging (for regression) or majority voting

(for classification) across all trees, leveraging the diversity created by bootstrap sampling.

4. **Out-of-Bag (OOB) Error Estimation**:
    - A unique advantage of bootstrap sampling is that, on average, about one-third of the data points are not included in any given bootstrap sample (these are called out-of-bag samples).
    - These out-of-bag samples can be used to evaluate the performance of each tree without needing a separate validation set, providing a built-in estimate of the model's generalization error (OOB error).
    - In Random Forest, the OOB error is often used to assess the model's accuracy and tune hyperparameters.

Bootstrap sampling is the process of creating random subsets of the training data with replacement, which is critical to Bagging methods like Random Forest. It plays a central role by:

- Generating diverse training sets for each base learner.
- Reducing variance and overfitting by averaging predictions.
- Enabling out-of-bag error estimation for model evaluation. In Random Forest, bootstrap sampling, combined with random feature selection, creates a powerful ensemble of diverse decision trees that collectively produce more accurate and stable predictions than a single tree.

Question 4: What are Out-of-Bag (OOB) samples and how is OOB score used to evaluate ensemble models?

Answer:

Out-of-Bag (OOB) samples are data points from the original training dataset that are not included in a particular bootstrap sample used to train a model in an ensemble method like Bagging (e.g., Random Forest).

- Context: In Bagging, bootstrap sampling creates subsets of the training data by sampling with replacement. On average, each bootstrap sample includes about 63% of the original data points (due to the nature of sampling with replacement), leaving approximately 37% of the data points unused in that sample. These unused data points are the OOB samples for that specific model (e.g., decision tree in a Random Forest).
- Key Property: Each model in the ensemble (e.g., each tree in a Random Forest) has its own unique set of OOB samples, which can be used as a validation set since they were not used during that model's training.

The OOB score is a performance metric derived from evaluating each model in the ensemble on its respective OOB samples. It provides an estimate of the model's generalization performance without requiring a separate validation set. Here's how it works and its role in evaluating ensemble models:

1. OOB Prediction:

- For each data point in the training set, identify the subset of models (e.g., trees in a Random Forest) where that data point was not included in the bootstrap sample (i.e., it's an OOB sample for those models).
- Use only those models to make predictions for that data point. For example:
  - In classification, the prediction is typically the majority vote across the OOB trees.
  - In regression, the prediction is the average of the OOB trees' outputs.
- This process is repeated for all data points, resulting in an OOB prediction for each instance in the training set.

2. Calculating the OOB Score:
   - The OOB score measures how well the ensemble's OOB predictions match the true labels/values of the training data.
   - For classification, the OOB score is often the accuracy (proportion of correct OOB predictions) or another metric like precision, recall, or F1-score.
   - For regression, the OOB score is typically based on metrics like mean squared error (MSE) or mean absolute error (MAE) between the OOB predictions and actual values.
   - The OOB score is aggregated across all data points to provide an overall estimate of the ensemble's performance.
3. Role in Model Evaluation:
   - Validation Without a Separate Set: The OOB score is a convenient way to estimate the model's generalization performance without splitting the data into training and validation sets. This is especially useful when the dataset is small, as it maximizes the use of available data for training.
   - Unbiased Estimate: Since OOB samples are not used in training the models that predict them, the OOB score provides an unbiased estimate of how the model will perform on unseen data, similar to cross-validation.
   - Hyperparameter Tuning: The OOB score can guide hyperparameter tuning (e.g., number of trees, maximum depth) in models like Random Forest. For example, you can compare OOB scores across different configurations to select the best parameters.
   - Feature Importance: In Random Forest, OOB samples can be used to estimate feature importance by measuring how much the OOB error increases when a feature's values are randomly permuted, indicating the feature's contribution to predictive accuracy.
4. Example in Random Forest:
   - Suppose a Random Forest has 100 trees, and a specific data point is an OOB sample for 30 of those trees (i.e., it wasn't included in the bootstrap samples for those 30 trees).
   - The predictions from those 30 trees are used to compute the OOB prediction for that data point (e.g., majority vote for classification or average for regression).
   - The OOB error is calculated by comparing these predictions to the true labels/values across all data points.
   - If the OOB accuracy is high (for classification) or the OOB error is low (for regression), it indicates the model generalizes well.

Question 5: Compare feature importance analysis in a single Decision Tree vs. a Random Forest.

Answer:

Feature importance analysis is a technique used to evaluate the contribution of each feature to the predictive power of a model. Both single Decision Trees and Random Forests (an ensemble of decision trees) can provide feature importance scores, but the methods and reliability of these scores differ significantly due to their structural differences. Below is a detailed comparison of feature importance analysis in a single Decision Tree versus a Random Forest.

1. Overview of Feature Importance

- **Single Decision Tree**: A decision tree splits the data based on features that best reduce a criterion (e.g., Gini impurity for classification or variance for regression). Feature importance is typically calculated based on how much each feature contributes to reducing this criterion across all splits.
- **Random Forest**: As an ensemble of decision trees, Random Forest aggregates feature importance scores across all trees, leveraging bootstrap sampling and random feature selection to provide a more robust and generalized estimate.

2. Key Differences in Feature Importance Analysis:

Aspect	Single Decision Tree	Random Forest
Calculation Method	Importance is based on the total reduction in the criterion (e.g., Gini impurity, variance) caused by splits on a feature across the tree. Common metrics include: • Gini Importance: Sum of Gini impurity reductions for all splits using the feature. • Mean Decrease in Impurity (MDI): Total decrease in impurity weighted by the number of samples at each split.	Importance is the average of the importance scores from individual trees. For each feature: • Compute the reduction in impurity (e.g., Gini or variance) for each tree where the feature is used. • Average this across all trees. • Optionally, use permutation importance (see below).
Permutation Importance	Rarely used, as it's computationally expensive and less common for a single tree.	Commonly used in Random Forests. It measures the increase in Out-of-Bag (OOB) error when a feature's values are

Aspect	Single Decision Tree	Random Forest
	Permutation importance involves shuffling a feature's values and measuring the increase in prediction error.	randomly permuted, averaged across all trees. This provides a robust estimate of feature importance.
Stability	Highly unstable. A single tree is sensitive to small changes in the training data, leading to different splits and importance scores if the data changes slightly.	More stable due to averaging across many trees trained on different bootstrap samples. The ensemble reduces the impact of noise and variability in individual trees.
Bias	Biased toward features with many unique values (e.g., continuous features) because they offer more potential split points, often inflating their importance.	Less biased. Random feature selection at each split (a hallmark of Random Forest) reduces the preference for features with many unique values, and averaging across trees further mitigates bias.
Robustness to Noise	Less robust. A single tree may overfit to noise or outliers, leading to misleading importance scores for irrelevant features.	More robust. Bootstrap sampling and random feature selection reduce the impact of noise, making importance scores more reliable for identifying truly predictive features.
Interpretability	Easier to interpret, as importance is based on a single tree's structure. You can directly trace splits to understand why a feature is deemed important.	Less interpretable at the individual tree level, as importance is aggregated across many trees. However, the overall importance score is more reliable for understanding feature contributions.
Out-of-Bag (OOB) Samples	Not applicable, as a single tree doesn't use bootstrap sampling, so there are no OOB samples for evaluation.	OOB samples are used to compute permutation importance, providing an unbiased estimate of feature importance without needing a separate validation set.
Computational Cost	Lower, as it involves analyzing splits in only one tree.	Higher, as it requires computing and averaging importance across many trees, plus optional permutation tests on OOB samples.
Feature Correlation	Struggles with correlated features. If two features are highly correlated, the tree may arbitrarily select one, inflating its importance while ignoring the other.	Better handles correlated features. Random feature selection and averaging across trees distribute importance more evenly among correlated features, reducing bias toward a single feature.

Question 6: Write a Python program to:

- Load the Breast Cancer dataset using `sklearn.datasets.load_breast_cancer()`
- Train a Random Forest Classifier
- Print the top 5 most important features based on feature importance scores.

(*Include your Python code and output in the code box below.*)

Answer:

Here's the Python program that fulfills the requirements

```
from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import RandomForestClassifier
import pandas as pd
import numpy as np

# Load the dataset
data = load_breast_cancer()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = data.target

# Train Random Forest Classifier
clf = RandomForestClassifier(n_estimators=100, random_state=42)
clf.fit(X, y)

# Get feature importances
importances = clf.feature_importances_

# Create a DataFrame for features and their importance
feature_importance_df = pd.DataFrame({
    'Feature': data.feature_names,
    'Importance': importances})

# Sort by importance and get top 5
top5 = feature_importance_df.sort_values(by='Importance', ascending=False).head(5)

# Print top 5
print("Top 5 Most Important Features:")
print(top5)
```

Explanation of Steps

1. **Load dataset** → load_breast_cancer() gives us both features (data.data) and target labels (data.target).

2. **Train model** → A RandomForestClassifier is trained on the full dataset.
3. **Get feature importance** → clf.feature_importances_ provides the importance score for each feature.
4. **Sort & display** → Use Pandas to sort and print the top 5.

Question 7: Write a Python program to:

- Train a Bagging Classifier using Decision Trees on the Iris dataset
- Evaluate its accuracy and compare with a single Decision Tree

(*Include your Python code and output in the code box below.*)

Answer:

Here's the Python program for the question

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.metrics import accuracy_score

# Load Iris dataset
data = load_iris()
X = data.data
y = data.target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Single Decision Tree
dt = DecisionTreeClassifier(random_state=42)
dt.fit(X_train, y_train)
y_pred_dt = dt.predict(X_test)
accuracy_dt = accuracy_score(y_test, y_pred_dt)

# Bagging Classifier with Decision Trees
bagging = BaggingClassifier(
    base_estimator=DecisionTreeClassifier(),
    n_estimators=50,
    random_state=42
)
bagging.fit(X_train, y_train)
y_pred_bag = bagging.predict(X_test)
accuracy_bag = accuracy_score(y_test, y_pred_bag)
```

```
# Results
print("Single Decision Tree Accuracy:", accuracy_dt)
print("Bagging Classifier Accuracy:", accuracy_bag)
```

How it works

1. Load dataset → Uses load_iris() from sklearn.datasets.
2. Single Decision Tree → Trains and evaluates accuracy on the test set.
3. Bagging Classifier → Trains an ensemble of decision trees using bootstrap sampling.
4. Comparison → Prints both accuracies side by side.

Question 8: Write a Python program to:

- Train a Random Forest Classifier
- Tune hyperparameters `max_depth` and `n_estimators` using GridSearchCV
- Print the best parameters and final accuracy

(*Include your Python code and output in the code box below)*

Answer:

Here's the Python program:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Load dataset
data = load_iris()
X = data.data
y = data.target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Define Random Forest model
rf = RandomForestClassifier(random_state=42)

# Define hyperparameter grid
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [None, 5, 10]
}
```

```
# GridSearchCV for tuning
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=5, n_jobs=-1)
grid_search.fit(X_train, y_train)

# Best model
best_rf = grid_search.best_estimator_

# Predictions
y_pred = best_rf.predict(X_test)

# Accuracy
accuracy = accuracy_score(y_test, y_pred)

# Results
print("Best Parameters:", grid_search.best_params_)
print("Final Accuracy:", accuracy)
```

How it works

1. Loads the Iris dataset for demonstration.
2. Splits data into training and test sets.
3. Random Forest Classifier is defined.
4. GridSearchCV searches over combinations of n_estimators and max_depth.
5. Best parameters & accuracy are printed.

Question 9: Write a Python program to:

- Train a Bagging Regressor and a Random Forest Regressor on the California Housing dataset
- Compare their Mean Squared Errors (MSE)

(*Include your Python code and output in the code box below.*)

Answer:

Here's the Python program

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import BaggingRegressor, RandomForestRegressor
from sklearn.metrics import mean_squared_error

# Load California Housing dataset
data = fetch_california_housing()
X = data.data
y = data.target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Bagging Regressor (with Decision Tree as base estimator)
bagging = BaggingRegressor(
    base_estimator=DecisionTreeRegressor(),
    n_estimators=50,
    random_state=42
)
bagging.fit(X_train, y_train)
y_pred_bag = bagging.predict(X_test)
mse_bag = mean_squared_error(y_test, y_pred_bag)

# Random Forest Regressor
rf = RandomForestRegressor(n_estimators=50, random_state=42)
rf.fit(X_train, y_train)
y_pred_rf = rf.predict(X_test)
mse_rf = mean_squared_error(y_test, y_pred_rf)

# Results
```

print("Bagging Regressor MSE:", mse_bag)
print("Random Forest Regressor MSE:", mse_rf)

Explanation:

- Dataset → fetch_california_housing() gives features & target values.
- Bagging Regressor → Uses multiple decision trees trained on random subsets of the data.
- Random Forest Regressor → Similar to bagging but also introduces random feature selection at each split.
- Comparison → Mean Squared Error (MSE) is calculated for both models.

Question 10: You are working as a data scientist at a financial institution to predict loan default. You have access to customer demographic and transaction history data.

You decide to use ensemble techniques to increase model performance.

Explain your step-by-step approach to:

- Choose between Bagging or Boosting
- Handle overfitting
- Select base models
- Evaluate performance using cross-validation
- Justify how ensemble learning improves decision-making in this real-world context.

(*Include your Python code and output in the code box below.*)

Answer:

Here's a structured step-by-step solution for Question, with both the conceptual explanation and an example Python implementation so it fits your "code + output" requirement.

Step-by-Step Approach

1. Choose Between Bagging or Boosting

- **Bagging**: Reduces variance by training multiple models on bootstrapped samples (e.g., Random Forest).
- **Boosting**: Reduces bias by sequentially training models, focusing on correcting previous mistakes (e.g., XGBoost, AdaBoost).
- **Choice for Loan Default Prediction**:
  Since financial datasets often have **imbalanced data** and **complex patterns**, **Boosting** is generally preferred because it can improve predictive power by focusing on hard-to-predict defaulters.

2. Handle Overfitting

- Use **max_depth**, **min_samples_leaf**, and **learning_rate** tuning for Boosting models.
- Apply **cross-validation** to ensure the model generalizes well.
- Use **early stopping** in gradient boosting to prevent overfitting.

---

3. Select Base Models

- For Bagging → Decision Trees (strong variance reduction).
- For Boosting → Shallow Decision Trees (weak learners to avoid overfitting).

---

4. Evaluate Performance using Cross-Validation

- Use **Stratified K-Fold Cross-Validation** (since default prediction is a classification problem).
- Metrics: **AUC-ROC** and **F1-score** are better than accuracy for imbalanced datasets.

---

5. Justify Ensemble Learning in this Context

- **Financial decisions are high-stakes** — ensemble models reduce the risk of relying on a single model.
- By combining multiple learners:
  - Bagging ensures stability (less variance in predictions).
  - Boosting increases accuracy (reduces bias).
- This leads to **better identification of risky customers**, minimizing default losses.

Python Example: Boosting for Loan Default Prediction:

```
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split, StratifiedKFold, cross_val_score
from sklearn.metrics import roc_auc_score
from xgboost import XGBClassifier

# Step 1: Simulate loan default dataset
X, y = make_classification(n_samples=5000, n_features=20, n_informative=10,
                           n_redundant=5, weights=[0.8, 0.2], random_state=42)

# Step 2: Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y,
```

```
random_state=42)

# Step 3: Define Boosting model (XGBoost)
model = XGBClassifier(
    n_estimators=200,
    learning_rate=0.1,
    max_depth=4,
    eval_metric='logloss',
    subsample=0.8,
    colsample_bytree=0.8,
    random_state=4)

# Step 4: Cross-validation (Stratified K-Fold)
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
auc_scores = cross_val_score(model, X_train, y_train, cv=cv, scoring='roc_auc')

# Step 5: Fit on full training data
model.fit(X_train, y_train)

# Step 6: Evaluate on test set
y_pred_proba = model.predict_proba(X_test)[:, 1]
test_auc = roc_auc_score(y_test, y_pred_proba)

# Output results
print("Cross-Validation AUC Scores:", auc_scores)
print("Mean CV AUC:", np.mean(auc_scores))
print("Test Set AUC:", test_auc)
```

Sample Output (Example)

Cross-Validation AUC Scores: [0.943, 0.946, 0.950, 0.941, 0.948]
Mean CV AUC: 0.9456
Test Set AUC: 0.9492

Why this works in real-world loan default prediction:

- Boosting focuses on customers likely to be misclassified by simpler models.
- Cross-validation ensures that the model works well across different subsets of customers.
- High AUC indicates strong discrimination between defaulters and non-defaulters, improving credit risk assessment.