USN | | | | | | | | | | |

Internal Assessment Test 3 – July 2024

Sub:	**Machine Learning**					Sub Code:	**21AI63**	Branch:	**AIML**	
Date:	**31/7/2024**	Duration:	**90 minutes**	Max Marks:	**50**	Sem/Sec:	**VI**		**OBE**	

		Answer any FIVE FULL Questions	**MARKS**	**CO**	**RBT**
1	a	Explain the following in brief: a) Training and visualizing of decision tree b) Making predictions ## a) Training and Visualizing of Decision Tree: **Training a Decision Tree**: Training a decision tree involves using a dataset where each data point has known inputs (features) and an output (target). The tree is built by recursively splitting the data based on feature values that result in the best separation of the target variable (using criteria like Gini Impurity or Entropy). The algorithm continues splitting until it either reaches a stopping condition (like maximum depth or minimum sample size) or cannot improve the splits further. **Visualizing a Decision Tree**: After training, a decision tree can be visualized as a flowchart-like structure. Each node represents a decision based on a feature, and each branch represents a possible outcome or condition. The leaves (end nodes) represent the final decision or prediction. Visualization helps in understanding how the tree splits data and makes decisions at each step. ## b) Making Predictions: Once a decision tree is trained, making predictions involves traversing the tree from the root to the appropriate leaf node. For a given input (a set of feature values), the tree applies the splits defined during training, choosing branches that match the feature values, until it reaches a leaf node. The output at the leaf node represents the predicted value (classification or regression) for that input. In classification, this output is typically the most common class in the leaf node, while in regression, it could be the average value.	[10]	4	L2
2	a	Demonstrate working of Voting classifier with code that creates and trains it. A **Voting Classifier** combines the predictions of multiple models to make a final prediction. It can use two types of voting: 1. **Hard Voting**: The class with the majority vote is selected as the prediction. 2. **Soft Voting**: The class probabilities are averaged, and the class with the highest average probability is selected. Below is an example demonstrating the use of a **Voting Classifier** using **hard voting**. We'll use scikit-learn to create and train the classifier. **Steps:** 1. Load a dataset (e.g., the **Iris dataset**). 2. Create individual classifiers (e.g., Logistic Regression, KNN, and Decision Tree). 3. Combine them into a **VotingClassifier**. 4. Train the classifier on the training data. 5. Make predictions on the test data. **Code Implementation:** python	[10]	4	L3

Copy code

```
# Import necessary libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.metrics import accuracy_score

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

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize individual classifiers
clf1 = LogisticRegression(max_iter=1000, random_state=42)
clf2 = KNeighborsClassifier()
clf3 = DecisionTreeClassifier(random_state=42)

# Create a Voting Classifier (using hard voting)
voting_clf = VotingClassifier(estimators=[('lr', clf1), ('knn', clf2), ('dt', clf3)], voting='hard')

# Train the Voting Classifier
voting_clf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = voting_clf.predict(X_test)

# Evaluate the model's accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy of the Voting Classifier: {accuracy * 100:.2f}%')

# Optionally, compare with individual classifiers
for clf in (clf1, clf2, clf3, voting_clf):
    clf_name = clf.__class__.__name__
    clf.fit(X_train, y_train)
    y_pred_individual = clf.predict(X_test)
    accuracy_individual = accuracy_score(y_test, y_pred_individual)
    print(f'Accuracy of {clf_name}: {accuracy_individual * 100:.2f}%')
```

Explanation:

1. **Dataset**: We use the **Iris dataset**, a popular dataset for classification tasks. It has 4 features and 3 classes.
2. **Classifiers**:
    - **Logistic Regression** (clf1)
    - **K-Nearest Neighbors** (clf2)
    - **Decision Tree Classifier** (clf3)
3. **Voting Classifier**: The VotingClassifier is created with a list of classifiers (clf1, clf2, clf3) and set to use **hard voting** (voting='hard').
4. **Training**: The model is trained using the training data (X_train, y_train).
5. **Prediction and Evaluation**: We make predictions on the test data (X_test) and compute the accuracy.

Output (Example):

text

Copy code

```
Accuracy of the Voting Classifier: 97.78%
Accuracy of LogisticRegression: 97.78%
```

		Accuracy of KNeighborsClassifier: 95.56% Accuracy of DecisionTreeClassifier: 91.11% Accuracy of VotingClassifier: 97.78% **Key Points:** • The **Voting Classifier** aggregates the predictions of the individual models, generally resulting in better performance than a single model. • In **hard voting**, each classifier makes a prediction, and the majority vote is taken as the final output. • **Soft voting** (not demonstrated here) would involve averaging the predicted probabilities for each class and selecting the class with the highest average probability.			
3	a	Demonstrate how new predictors can correct its predecessor by using training instances of underfitted predecessor. To demonstrate how new predictors can correct the mistakes of an underfitted predecessor, we can use **Boosting**. Boosting is an ensemble technique where each new model (predictor) is trained to correct the errors (residuals) made by its predecessor. This technique combines weak learners (models that perform slightly better than random guessing) to form a strong learner, by focusing more on the misclassified instances during each iteration. A common example of boosting is **AdaBoost** (Adaptive Boosting). In AdaBoost, the algorithm gives higher weight to the training instances that were misclassified by previous models, thereby encouraging the new model to focus on those instances and correct the errors. Let's demonstrate how AdaBoost works and how new predictors can correct the mistakes of underfitted predecessors using Python code: Steps: 1. **Train an underfitted model** (weak learner), such as a shallow decision tree (a "stump"). 2. **Boost the model** by training additional weak learners to focus on the errors of the previous learners. 3. **Combine the models** to make the final prediction. Code Implementation with AdaBoost: ```python # Import necessary libraries from sklearn.datasets import load_iris from sklearn.model_selection import train_test_split from sklearn.tree import DecisionTreeClassifier from sklearn.ensemble import AdaBoostClassifier from sklearn.metrics import accuracy_score # Load the Iris dataset data = load_iris() X = data.data y = data.target # Split the data into training and testing sets X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42) # Initialize a weak learner (shallow decision tree) weak_learner = DecisionTreeClassifier(max_depth=1, random_state=42) # Train the weak learner weak_learner.fit(X_train, y_train) # Make predictions with the weak learner	[5]	4	L3

		y_pred_weak = weak_learner.predict(X_test) # Evaluate the performance of the weak learner accuracy_weak = accuracy_score(y_test, y_pred_weak) print(f'Accuracy of the weak learner (underfitted): {accuracy_weak * 100:.2f}%') # Now, apply AdaBoost to improve performance by adding additional weak learners adaboost = AdaBoostClassifier(base_estimator=weak_learner, n_estimators=50, random_state=42) # Train AdaBoost model adaboost.fit(X_train, y_train) # Make predictions with the AdaBoost model y_pred_adaboost = adaboost.predict(X_test) # Evaluate the performance of the AdaBoost model accuracy_adaboost = accuracy_score(y_test, y_pred_adaboost) print(f'Accuracy of the AdaBoost model: {accuracy_adaboost * 100:.2f}%') ``` Explanation: 1. **Weak Learner (Shallow Decision Tree)**: We first train a weak learner (a decision tree with depth 1, also known as a "decision stump"). This model is underfitted and is expected to perform poorly on the test data. 2. **AdaBoost Classifier**: AdaBoost is then used to create an ensemble of 50 weak learners. AdaBoost adjusts the weights of the training instances during each iteration, emphasizing those that were misclassified by previous models. This helps subsequent models focus on correcting the errors of the predecessor. 3. **Training and Predictions**: The `AdaBoostClassifier` is trained using the weak learner, and predictions are made on the test data. The accuracy of both the underfitted weak learner and the AdaBoost ensemble model is evaluated. Output (Example): ```text Accuracy of the weak learner (underfitted): 55.56% Accuracy of the AdaBoost model: 97.78% ``` Key Insights: - The **weak learner** (decision stump) has poor accuracy because it is underfitted. For example, it might only learn basic patterns like the class distribution but not the intricate relationships in the data. - **AdaBoost** significantly improves accuracy by adding successive learners that focus on the misclassified instances. As each new model is trained, it tries to correct the mistakes of the previous ones by assigning more weight to the misclassified instances. This leads to a much better performance in the final ensemble model. - The improvement shows how new predictors (in the form of AdaBoost's weak learners) correct the errors of underfitted predecessors.			
	b	Explain Gradient Boosting in brief. Gradient Boosting in Brief **Gradient Boosting** is an advanced ensemble technique used for both regression and classification tasks. It builds a strong model by combining multiple weak learners (usually decision trees) sequentially, where each new learner corrects the errors made by the	[5]	4	L2

previous ones. The key idea is that instead of fitting new models to the original data, each new model is trained to predict the residual errors (the difference between the actual and predicted values) of the previous model.

Here's how **Gradient Boosting** works, step-by-step:

1. **Initialization**:
   - The first model in the sequence is trained on the data, typically a simple model (e.g., decision tree).
   - For regression, the initial prediction can be the mean of the target values. For classification, it could be the log odds of the classes.

2. **Compute Residuals**:
   - After the first model makes its predictions, the residuals (errors) are computed as the difference between the true target values and the predictions.
   - These residuals represent the areas where the model is making mistakes and where future models will focus to improve the overall performance.

3. **Train the Next Model**:
   - A new model is trained on the residuals from the previous model (instead of the original target values). This new model tries to predict the errors made by the previous model.
   - The new model is usually a small decision tree (often referred to as a "decision stump" for simplicity).

4. **Update the Model**:
   - The predictions of the new model are combined with the previous model's predictions. Typically, a learning rate (shrinkage factor) is applied to control how much influence the new model has on the final prediction.
   - The combined model will now make better predictions, as the second model has corrected some of the errors of the first one.

5. **Repeat the Process**:
   - Steps 2 through 4 are repeated iteratively. With each iteration, a new model is added to reduce the residuals of the previous ensemble model.
   - The process continues for a specified number of iterations or until a stopping criterion is met (such as no improvement in performance).

6. **Final Prediction**:
   - The final prediction is a weighted sum of the predictions from all the individual models, where each model contributes according to its accuracy and the learning rate.

Key Characteristics of Gradient Boosting:
- **Boosting**: Gradient Boosting is a boosting technique, meaning it builds models sequentially and corrects the errors of the previous models.
- **Gradient Descent**: The "gradient" part comes from using gradient descent to minimize the residual errors. This is what differentiates Gradient Boosting from other boosting algorithms (like AdaBoost). Gradient descent is used to find the optimal model by iteratively adjusting the model parameters to reduce errors.
- **Additive Model**: The method adds new trees (or models) to the ensemble one at a time, with each new tree focusing on the mistakes made by the current ensemble of trees.
- **Learning Rate**: A key hyperparameter in Gradient Boosting that controls the contribution of each new model. A lower learning rate requires more trees but may result in better generalization.

Advantages of Gradient Boosting:
- **High Accuracy**: Gradient Boosting often provides state-of-the-art performance in many machine learning tasks.
- **Flexibility**: It can be used for both regression and classification tasks and works well on a variety of data types.
- **Handles Complex Data**: Gradient Boosting can model complex data relationships and interactions due to the iterative process of correcting residuals.

Disadvantages:
- **Prone to Overfitting**: If not properly tuned (especially the number of trees or the learning rate), Gradient Boosting can overfit the training data.
- **Computationally Intensive**: It can be slower to train compared to other algorithms, especially when the number of iterations is large.
- **Sensitivity to Hyperparameters**: Proper tuning of parameters such as the learning rate, number of trees, and tree depth is critical for good performance.

Example of Gradient Boosting in Scikit-learn:

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score

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

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create a Gradient Boosting model
gb_model = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, max_depth=3)

# Train the model
gb_model.fit(X_train, y_train)

# Make predictions
y_pred = gb_model.predict(X_test)

# Evaluate performance
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Gradient Boosting Model: {accuracy * 100:.2f}%")
```

4	a	Construct a regression using the following data which consists of 10 data instances and three attributes “Assessment’, ‘Assignment’ and Project.(Refer the notes) Sl. no / Assessments / Assignment / Project / Result(%) 1 / Good / Yes / Yes / 95 2 / Average / Yes / No / 70 3 / Good / No / Yes / 75 4 / Poor / No / No / 45 5 / Good / Yes / Yes / 98 6 / Average / No / Yes / 80 7 / Good / No / No / 75 8 / Poor / Yes / Yes / 65 9 / Average / No / No / 58 10 / Good / Yes / Yes / 89	[10]	4	L3
5	a	Explain Bagging and Pasting in brief. **Bagging and Pasting: Brief Overview** **Bagging** (Bootstrap Aggregating) and **Pasting** are both ensemble methods designed to improve the performance of machine learning models by combining the predictions of multiple base models. Both techniques reduce variance and prevent overfitting, but they differ in how they generate the training data for each base model. **1. Bagging (Bootstrap Aggregating):**	[10]	4	L2

Bagging is an ensemble technique that trains multiple models on **random subsets** of the training data, where each subset is drawn with **replacement**. The idea is to generate different training sets by sampling with replacement (i.e., bootstrap sampling) and then averaging the predictions (for regression) or using a majority vote (for classification) to produce the final result.

Key Characteristics of Bagging:

- **Data Sampling**: Each model is trained on a random subset of the training data sampled **with replacement**. This means that some data points may appear multiple times in a subset, while others may not appear at all.
- **Model Independence**: The base models (usually weak learners like decision trees) are trained independently of each other.
- **Final Prediction**: For regression tasks, the predictions of the models are averaged. For classification, the majority vote from all the base models is taken.

Benefits of Bagging:

- Reduces variance by averaging out the predictions of different models.
- Helps prevent overfitting by reducing the sensitivity to the noise in the data.
- Can improve the performance of weak learners, such as decision trees, by using multiple models.

Example:

- A common example of **Bagging** is the **Random Forest** algorithm, where multiple decision trees are trained on different subsets of the data (with replacement) and then averaged (for regression) or voted on (for classification).

2. Pasting:

Pasting is very similar to Bagging, but with one key difference: in **Pasting**, the subsets of the training data are sampled **without replacement**. This means that each subset used to train the base models contains unique data points, without duplication.

Key Characteristics of Pasting:

- **Data Sampling**: Each model is trained on a random subset of the training data, but the subset is drawn **without replacement**. This results in training sets that are smaller than the original dataset, and each data point can appear only once in each subset.
- **Model Independence**: Like Bagging, the models are trained independently on these subsets.
- **Final Prediction**: The final prediction is made by averaging the predictions (for regression) or using a majority vote (for classification).

Benefits of Pasting:

- Reduces variance by using multiple models, but with no repeated data points in any single training set.
- Helps avoid overfitting, similar to Bagging, but may be more robust in scenarios where the dataset is not large.

Example:

- **Pasting** can be seen as a variant of **Bagging** in algorithms like **Bootstrap Aggregating**, but it is less commonly used compared to Bagging.

6	a	Explain Stack Generalization in brief. **Stacking**, also known as **Stacked Generalization**, is an ensemble learning technique that combines multiple machine learning models to improve predictive performance. Unlike traditional methods like Bagging or Boosting, which rely on combining predictions from models of the same type, **Stacking** involves using different types of models and combining their outputs using another model (called the **meta-model** or **stacker**). The main idea behind stacking is that different base models might have different strengths and weaknesses, and by combining them, we can improve generalization and make more accurate predictions. **How Stacking Works:** 1. **Train Base Learners**: o First, multiple base models (also called **level-0 models**) are trained on the original dataset. These base models could be different algorithms (e.g., decision trees, SVM, logistic regression) or even different hyperparameter configurations of the same algorithm. 2. **Generate Predictions for Meta-Model**:	[10]	4	L2

- Once the base learners are trained, they make predictions on the training data (or a separate validation set). These predictions are collected and treated as new features for the next model.

3. **Train Meta-Model**:
   - A **meta-model** (also called a **level-1 model**) is trained on the predictions made by the base learners. The meta-model learns how to combine the outputs of the base models to make the final prediction. Commonly, this meta-model is a simple algorithm like logistic regression, but it can be any model that can handle the predictions from the base models as input.
4. **Final Prediction**:
   - To make predictions on new (test) data, the base learners first generate predictions. These predictions are then passed to the meta-model, which combines them to make the final prediction.

Example of Stacking Workflow:

1. **Step 1**: Train multiple base models (e.g., Random Forest, SVM, and KNN) on the training data.
2. **Step 2**: Use the trained base models to make predictions on the training data (or a holdout set). These predictions form a new feature matrix.
3. **Step 3**: Train a meta-model (e.g., Logistic Regression or another model) on the predictions of the base models.
4. **Step 4**: When making predictions on new data, the base models generate predictions, which are fed into the meta-model to produce the final output.

Advantages of Stacking:

- **Improved Generalization**: By combining models that may perform differently on different parts of the data, stacking often leads to better generalization compared to individual models.
- **Flexibility**: Stacking can use different types of models, making it flexible and adaptable to various problems.
- **Reduction of Bias**: The meta-model can learn to correct the biases or weaknesses of the base models, often leading to improved predictive accuracy.

Disadvantages of Stacking:

- **Complexity**: Stacking involves training multiple models and requires additional computational resources compared to simpler ensemble methods like Bagging or Boosting.
- **Overfitting Risk**: If not properly tuned (e.g., when the base models are too complex), stacking can lead to overfitting, especially if the meta-model overfits to the predictions of the base models.
- **Data Requirements**: Stacking often requires more data, as the meta-model needs a sufficiently large dataset of base model predictions to make accurate predictions.

CI **CCI** **HoD**