**Name: Mozeb Ahmed Khan**

**Roll No: 20F-0161**

**Sec: BS(CS)-7A**

**Assignment: 02**

**Course: Applied Machine Learning**

**Question 1: Decision Tree and Random Forest**

1. **What is motivation behind ensemble methods?**

An ensemble method is a technique that aims to improves the performance and accuracy of machine learning algorithms, by combining the predictions from multiple base models. A model comprising of many models is called an ensemble model. The motivation to introduce these ensemble methods is to make more accurate predictions than any individual model and provide more reliability in various real-world scenarios. Some of the factors behind motivation are listed below:

* **Reducing Variance and Error:** Ensemble methods decreases the ratio of error and variance, caused in ensemble models than the variance and error caused in individual model. It happens because, when the individual models are combined, errors and biases are being averaged out, which brings out a more accurate prediction.
* **Handle Complex Data:** Ensemble model enhances the ability to generalize unseen data because it may analyze and understand the data patterns and relationships, in much better way as compared to individual model.
* **Increase Robustness:** Ensemble models provide more robustness to handle outliers and noisy data and they have less chances of overfitting the training data.

1. **How Random Forest training and inference works? Give pseudo code.**

Random Forest is an ensemble learning method that builds multiple decision trees during training and combines their predictions during inference. Random forest is a bagging technique and not a boosting technique. The trees in random forests run in parallel. There is no interaction between these trees while building the trees.

* **Training:** In training, we randomly select a subset of the training data (with replacement) to create a bootstrap sample. This sample helps to train each decision tree.

For each tree in a random forest:

1. We randomly select a subset of features from the whole set of features.
2. Then, we build a decision tree using the created sample and selected features. In the tree, we split data at each node by recursively selecting the best feature.

**Pseudo Code:**

def RandomForestTrain(data, num\_trees):

forest = []

for i in range(num\_trees):

subset = random\_subsample(data)

tree = DecisionTreeTrain(subset)

forest.append(tree)

return forest

* **Inference:** In inference, we collect the predictions from each decision tree either through classification or regression.

For each input sample to be predicted:

1. First, we pass the sample through each decision tree in the random forest.
2. Then, we collect the predictions from each tree (for classification, use class labels; for regression, use numeric values).
3. Lastly, we determine the final predicted class (for classification, use majority voting to determine the class; for regression, use the average of the predictions).

**Pseudo Code:**

def RandomForestPredict(forest, new\_data, task='classification'):

predictions = []

for tree in forest:

prediction = DecisionTreePredict(tree, new\_data)

predictions.append(prediction)

if task == 'classification':

# For classification, use majority voting

final\_prediction = majority\_vote(predictions)

elif task == 'regression':

# For regression, use average

final\_prediction = average(predictions)

return final\_prediction

1. **Iris dataset has 50 samples for each of three different species of Iris flower (total number of samples is 150). For each data sample, you have sepal length, sepal width, petal length and petal width and a species name (class/label). Figure below shows Iris flower and features in dataset:**



**Perform the following tasks:**

* **Load the given Iris dataset**

# Step 1: Load the Iris Dataset

# Using pandas library for data analysis

import pandas as pd

# Load the given Iris dataset from the Iris.csv file

iris\_data = pd.read\_csv('/Iris.csv')

* **Split it into a training set and a test set**

# Step 2: Split the Data into a Training Set and Test Set

# Importing train\_test\_split from Scikit-Learn to split data into training and testing sets

from sklearn.model\_selection import train\_test\_split

# X means features (sepal length, sepal width, petal length, and petal width)

X = iris\_data.iloc[:, 1:5]

# Y means target variable (class or label)

Y = iris\_data['Species']

# As 'test\_size=0.2', thus, 20% of the data will be used for testing, and the remaining 80% will be used for training

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.2, random\_state=1)

* **Preprocess the data**

# Step 3: Preprocess the Data (if needed)

# Encoding 'Species' column into numerical values

iris\_data = pd.get\_dummies(iris\_data, columns=['Species'], prefix=['Species'], drop\_first=True)

# Importing the StandardScaler from Scikit-Learn to standardize the specific colums of the dataset.

from sklearn.preprocessing import StandardScaler

# Creating a StandardScaler instance

scaler = StandardScaler()

# # Apply the StandardScaler to these selected columns

iris\_data[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']] = scaler.fit\_transform(iris\_data[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']])

* **Build a Decision Tree classifier**

# Step 4: Build a Decision Tree Classifier

# Importing the DecisionTreeClassifier from sklearn.tree to build decision tree classifier

from sklearn.tree import DecisionTreeClassifier

# Create a decision tree classifier

decision\_tree\_classifier = DecisionTreeClassifier()

# Train the model on the training data

decision\_tree\_classifier.fit(X\_train, Y\_train)

* **Then train a Random Forest classifier**

# Step 5: Train a Random Forest Classifier

# Importing the RandomForestClassifier from sklearn.ensemble to train a random forest classifier

from sklearn.ensemble import RandomForestClassifier

# Create a random forest classifier

random\_forest\_classifier = RandomForestClassifier(n\_estimators=100)

# Train the model on the training data

random\_forest\_classifier.fit(X\_train, Y\_train)

* **How much better does it perform compared to the Decision Tree classifier?**

# Step 6: Evaluate the Classifiers

# Importing accuracy\_score and classification\_report from sklearn.metrics to evaluate the accuracy between decision tree and random forest

from sklearn.metrics import accuracy\_score, classification\_report

# Evaluate the Decision Tree classifier

decision\_tree\_predictions = decision\_tree\_classifier.predict(X\_test)

# Using accuracy\_score to calculate the accuracy of a model's predictions.

decision\_tree\_accuracy = accuracy\_score(Y\_test, decision\_tree\_predictions)

print("Decision Tree Classifier:")

print(f"Accuracy: {decision\_tree\_accuracy}\n")

# Using classification\_report for evaluating the performance of a classification model.

print(classification\_report(Y\_test, decision\_tree\_predictions))

# Evaluate the Random Forest classifier

random\_forest\_predictions = random\_forest\_classifier.predict(X\_test)

# Using accuracy\_score to calculate the accuracy of a model's predictions.

random\_forest\_accuracy = accuracy\_score(Y\_test, random\_forest\_predictions)

print("\nRandom Forest Classifier:")

print(f"Accuracy: {random\_forest\_accuracy}\n")

# Using classification\_report for evaluating the performance of a classification model.

print(classification\_report(Y\_test, random\_forest\_predictions))

* **Code:**

# Step 1: Load the Iris Dataset

# Using pandas library for data analysis

import pandas as pd

# Load the given Iris dataset from the Iris.csv file

iris\_data = pd.read\_csv('/Iris.csv')

# Step 2: Split the Data into a Training Set and Test Set

# Importing train\_test\_split from Scikit-Learn to split data into training and testing sets

from sklearn.model\_selection import train\_test\_split

# X means features (sepal length, sepal width, petal length, and petal width)

X = iris\_data.iloc[:, 1:5]

# Y means target variable (class or label)

Y = iris\_data['Species']

# As 'test\_size=0.2', thus, 20% of the data will be used for testing, and the remaining 80% will be used for training

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.2, random\_state=1)

# Step 3: Preprocess the Data (if needed)

# Encoding 'Species' column into numerical values

iris\_data = pd.get\_dummies(iris\_data, columns=['Species'], prefix=['Species'], drop\_first=True)

# Importing the StandardScaler from Scikit-Learn to standardize the specific colums of the dataset.

from sklearn.preprocessing import StandardScaler

# Creating a StandardScaler instance

scaler = StandardScaler()

# # Apply the StandardScaler to these selected columns

iris\_data[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']] = scaler.fit\_transform(iris\_data[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']])

# Step 4: Build a Decision Tree Classifier

# Importing the DecisionTreeClassifier from sklearn.tree to build decision tree classifier

from sklearn.tree import DecisionTreeClassifier

# Create a decision tree classifier

decision\_tree\_classifier = DecisionTreeClassifier()

# Train the model on the training data

decision\_tree\_classifier.fit(X\_train, Y\_train)

# Step 5: Train a Random Forest Classifier

# Importing the RandomForestClassifier from sklearn.ensemble to train a random forest classifier

from sklearn.ensemble import RandomForestClassifier

# Create a random forest classifier

random\_forest\_classifier = RandomForestClassifier(n\_estimators=100)

# Train the model on the training data

random\_forest\_classifier.fit(X\_train, Y\_train)

# Step 6: Evaluate the Classifiers

# Importing accuracy\_score and classification\_report from sklearn.metrics to evaluate the accuracy between decision tree and random forest

from sklearn.metrics import accuracy\_score, classification\_report

# Evaluate the Decision Tree classifier

decision\_tree\_predictions = decision\_tree\_classifier.predict(X\_test)

# Using accuracy\_score to calculate the accuracy of a model's predictions.

decision\_tree\_accuracy = accuracy\_score(Y\_test, decision\_tree\_predictions)

print("Decision Tree Classifier:")

print(f"Accuracy: {decision\_tree\_accuracy}\n")

# Using classification\_report for evaluating the performance of a classification model.

print(classification\_report(Y\_test, decision\_tree\_predictions))

# Evaluate the Random Forest classifier

random\_forest\_predictions = random\_forest\_classifier.predict(X\_test)

# Using accuracy\_score to calculate the accuracy of a model's predictions.

random\_forest\_accuracy = accuracy\_score(Y\_test, random\_forest\_predictions)

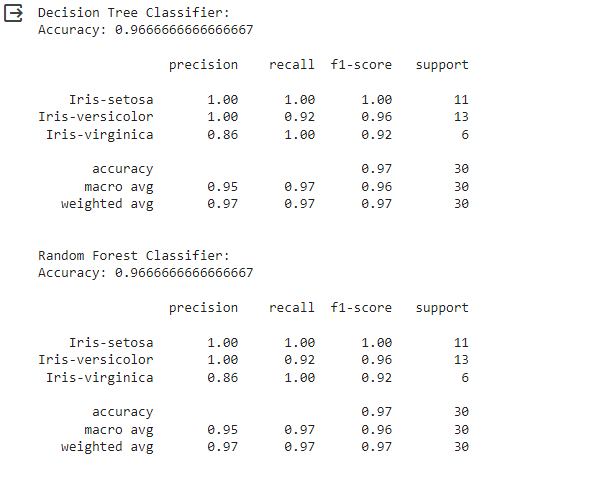
print("\nRandom Forest Classifier:")

print(f"Accuracy: {random\_forest\_accuracy}\n")

# Using classification\_report for evaluating the performance of a classification model.

print(classification\_report(Y\_test, random\_forest\_predictions))

* **Output:**



**Question 2: Support Vector Machine (SVM)**

1. **What is a support vector? Derive the objective function of support vector machines (SVM) for linearly separable data?**

**Support Vector:**

A support vector is used to support the hyperplane. A hyperplane is a decision boundary, drawn between two classes, that maximizes the margin or space between two classes. A support vector is actually a datapoint that is closest to the margin or linear separator. Thus, it is very crucial for defining the decision boundary (hyperplane) between the two classes.

**Objective Function For SVM For Linearly Separable Data:**

For linearly separable data, we have to derive an objective function of support vector machines such that the hyperplane gives maximum margin between the two classes and the minimum number of classification errors.

**Margin:**

The margin will be defined as the perpendicular distance between any point closest and the hyperplane.

**Support Vectors:**

The support vectors will be the data points that are on the margin or within the boundary.

**Objective Function:**

This formulation requires that all training examples are correctly classified, and there is no allowance for misclassifications. The objective function for a linearly separable SVM will be derived in this manner:

1. Given a dataset with training examples:

Input features: **xᵢ (i = 1, 2, ..., n)**

Class labels: **yᵢ (yᵢ = +1 or yᵢ = -1 for binary classification)**

1. The objective is to find the optimal hyperplane, defined by a weight vector **(w)** and a bias term **(b)** such that it maximizes the margin while ensuring all data points are correctly classified. The margin **(2/‖w‖)** is inversely proportional to the norm **(‖w‖)** of the weight vector.
2. The decision function for this hyperplane is given by:

**f(x) = w·x + b**

1. The margin can be defined as follows:

**Margin = 2 / ‖w‖**

1. The SVM objective is to maximize the margin while minimizing the classification error. This can be formulated as an optimization problem:

Maximize: **2 / ‖w‖**

Subject to: **yᵢ (w·xᵢ + b) ≥ 1** for all training examples **(i = 1, 2, ..., n)**

The constraint **yᵢ (w·xᵢ + b) ≥ 1** ensures that all data points are correctly classified and lie outside a certain margin.

1. This optimization problem can be converted into a minimization problem by directly optimizing the inverse of the margin, which is equivalent to minimizing ‖w‖². So, we aim to minimize ‖w‖²:

Minimize: **1/2 ‖w‖²**

Subject to: **yᵢ (w·xᵢ + b) ≥ 1** for all training examples **(i = 1, 2, ..., n)**

The objective function now is to minimize 1/2 ‖w‖² while ensuring that all data points are correctly classified and have a margin of at least 1. We can determine the decision boundary and make predictions when we get optimal values of w and b.

It is suitable when the data is perfectly linearly separable and we are not allowing any classification errors.

1. **Differentiate between soft margin and hard margin classifier.**

Hard margin and soft margin classifiers are the two variations of SVM. However, they vary in their approach to handle margin and data points.

**Key Differences:**

Some of the key differences on the basis of few factors are listed below:

|  |  |  |
| --- | --- | --- |
| **Factors** | **Soft Margin Classifier** | **Hard Margin Classifier** |
| **Robustness** | It is more robust to outliers and noisy data as it can handle misclassification to some extent. | It is less robust to outliers and noisy data as it cannot handle misclassification. |
| **Margin** | The margin is smaller and flexible as it has to accommodate some misclassifications, occurred due to outliers. | The margin is larger as it is directly associated with the separation of classes (perfectly linearly separable data allows no misclassification). |
| **Misclassification** | It is flexible for data which is not perfectly linearly separable (noisy data and outlier) and allows certain misclassifications within the margin. | It can provide solution only for perfectly linearly separable data points and allows no misclassification. |
| **Hyperparameters** | It has hyperparameter to balance between maximizing the margin and allowing misclassifications. | It has no hyperparameter because it does not allow any misclassification. |
| **Aim** | It aims to find an optimal solution which maximizes the margin and minimizes the classification error. | It aims to find the largest possible margin that correctly separates the data and data must be perfectly linearly separable. |

1. **In this question we will be using the popular MNIST dataset, which is a set of 70,000 small images of digits handwritten digits (https://www.kaggle.com/datasets/hojjatk/mnist-dataset). Each image is labelled with the digit it represents. There are 70,000 images, and each image has 784 features. This is because each image is 28 × 28 pixels, and each feature simply represents one pixel’s intensity, from 0 (white) to 255 (black). Figure below shows some digits from the MNIST dataset:**



**Perform the following tasks:**

* **Download MNIST dataset.**

# Step 1: Load the MNIST Dataset

import pandas as pd

# Load the training and test data from CSV files

train\_data = pd.read\_csv('mnist\_train.csv')

test\_data = pd.read\_csv('mnist\_test.csv')

# Extract labels and features

x\_train = train\_data.iloc[:, 1:].values

y\_train = train\_data.iloc[:, 0].values

x\_test = test\_data.iloc[:, 1:].values

y\_test = test\_data.iloc[:, 0].values

* **Train a linear SVM classifier on the MNIST. Since SVM classifiers are binary classifiers (you will need to use one-versus-all to classify all 10 digits), and also report its training accuracy.**

# Step2: Train linear SVM classifier

from sklearn import svm

from sklearn.impute import SimpleImputer  # Import SimpleImputer

# Create a simple imputer to replace missing values with the mean of the column

imputer = SimpleImputer(strategy='mean')

# Fit and transform the imputer on x\_train

x\_train = imputer.fit\_transform(x\_train)

x\_test = imputer.transform(x\_test)

# Train a linear SVM classifier

clf\_linear = svm.SVC(kernel='linear', decision\_function\_shape='ovr')

clf\_linear.fit(x\_train, y\_train)

# Report training accuracy

train\_accuracy = clf\_linear.score(x\_train, y\_train)

print("Linear SVM Training Accuracy:", train\_accuracy)

* **Scale (standardize) the data first, retrain an linear SVM classifier and also report its training accuracy.**

# Step3: Standard Scaling

from sklearn.preprocessing import StandardScaler

# Create a standard scaler

scaler = StandardScaler()

# Fit and transform the training data

x\_train\_scaled = scaler.fit\_transform(x\_train)

# Retrain the linear SVM classifier on the scaled data

clf\_linear\_scaled = svm.SVC(kernel='linear', decision\_function\_shape='ovr')

clf\_linear\_scaled.fit(x\_train\_scaled, y\_train)

# Report scaled training accuracy

train\_accuracy\_scaled = clf\_linear\_scaled.score(x\_train\_scaled, y\_train)

print("Linear SVM (Scaled) Training Accuracy:", train\_accuracy\_scaled)

* **Now retrain the classifier with a non-linear SVM using Radial Basis Function (RBF) Kernel (aka Gaussian Kernel), and also report its training accuracy.**

# Step4: Train a non-linear SVM with the RBF kernel

clf\_rbf = svm.SVC(kernel='rbf', decision\_function\_shape='ovr')

clf\_rbf.fit(x\_train\_scaled, y\_train)

# Report RBF SVM training accuracy

train\_accuracy\_rbf = clf\_rbf.score(x\_train\_scaled, y\_train)

print("RBF SVM Training Accuracy:", train\_accuracy\_rbf)

* **Now make predictions on the above models for each class on test data and report your results.**

# Step5: Make predictions using the linear SVM

linear\_svm\_predictions = clf\_linear\_scaled.predict(x\_test\_scaled)

# Make predictions using the RBF SVM

rbf\_svm\_predictions = clf\_rbf.predict(x\_test\_scaled)

# Report classification results for both models

print("Linear SVM Classification Report:\n", classification\_report(y\_test, linear\_svm\_predictions))

print("RBF SVM Classification Report:\n", classification\_report(y\_test, rbf\_svm\_predictions))

* **Code:**

import pandas as pd

from sklearn import svm

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import classification\_report

from sklearn.impute import SimpleImputer  # Import SimpleImputer

# Load the training and test data from CSV files

train\_data = pd.read\_csv('mnist\_train.csv')

test\_data = pd.read\_csv('mnist\_test.csv')

# Extract labels and features

x\_train = train\_data.iloc[:, 1:].values

y\_train = train\_data.iloc[:, 0].values

x\_test = test\_data.iloc[:, 1:].values

y\_test = test\_data.iloc[:, 0].values

# Create a simple imputer to replace missing values with the mean of the column

imputer = SimpleImputer(strategy='mean')

# Fit and transform the imputer on x\_train

x\_train = imputer.fit\_transform(x\_train)

x\_test = imputer.transform(x\_test)

# Train a linear SVM classifier

clf\_linear = svm.SVC(kernel='linear', decision\_function\_shape='ovr')

clf\_linear.fit(x\_train, y\_train)

# Report training accuracy

train\_accuracy = clf\_linear.score(x\_train, y\_train)

print("Linear SVM Training Accuracy:", train\_accuracy)

# Create a standard scaler

scaler = StandardScaler()

# Fit and transform the training data

x\_train\_scaled = scaler.fit\_transform(x\_train)

# Retrain the linear SVM classifier on the scaled data

clf\_linear\_scaled = svm.SVC(kernel='linear', decision\_function\_shape='ovr')

clf\_linear\_scaled.fit(x\_train\_scaled, y\_train)

# Report scaled training accuracy

train\_accuracy\_scaled = clf\_linear\_scaled.score(x\_train\_scaled, y\_train)

print("Linear SVM (Scaled) Training Accuracy:", train\_accuracy\_scaled)

# Train a non-linear SVM with the RBF kernel

clf\_rbf = svm.SVC(kernel='rbf', decision\_function\_shape='ovr')

clf\_rbf.fit(x\_train\_scaled, y\_train)

# Report RBF SVM training accuracy

train\_accuracy\_rbf = clf\_rbf.score(x\_train\_scaled, y\_train)

print("RBF SVM Training Accuracy:", train\_accuracy\_rbf)

# Transform the test data using the same scaler

x\_test\_scaled = scaler.transform(x\_test)

# Make predictions using the linear SVM

linear\_svm\_predictions = clf\_linear\_scaled.predict(x\_test\_scaled)

# Make predictions using the RBF SVM

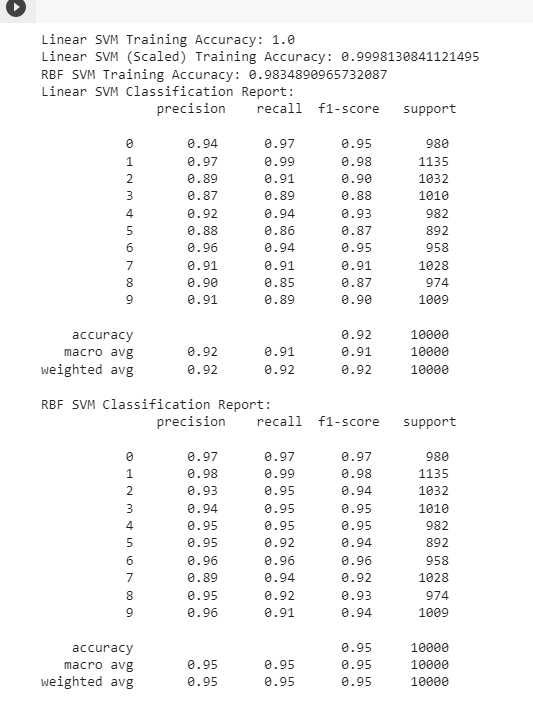
rbf\_svm\_predictions = clf\_rbf.predict(x\_test\_scaled)

# Report classification results for both models

print("Linear SVM Classification Report:\n", classification\_report(y\_test, linear\_svm\_predictions))

print("RBF SVM Classification Report:\n", classification\_report(y\_test, rbf\_svm\_predictions))

* **Output:**



**The End.**

**Thank You.**