Final Lab Report

Machine Learning

Subject Code -SE204a

Submitted to Ms Anjali Bansal

Submitted by Gaurav Meena
Roll Number - 2K22/SE/65
Department of software engineering

DEPARTMENT OF SOFTWARE ENGINEERING
DELHI TECHNOLOGICAL UNIVERSITY
SHAHBAD, DAULATPUR
MAIN BAWANA ROAD,
Delhi 110042

INDEX

INDEX

Experiment – 1

Aim: To implement basics of python

Theory:

Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation.

Features of Python language:

- High-level language
- Dynamically-typed
- Garbage-collected
- It supports multiple programming paradigms, such as structured (particularly procedural programming), object-oriented programming, and functional programming.
- Contains a comprehensive standard library.
- Most suitable programming language for Data Science
- One of the most popular programming languages

Numpy: NumPy is a library for the Python programming language, adding support for large, multidimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these multi-dimensional arrays.

Matplotlib: Matplotlib is a plotting library for the Python programming language. It can be used to draw a wide variety of plots including line plots, scatter plots, bar charts, and histograms.

Pandas: Pandas (derived from the term "panel data") is a software library for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.

Scipy: SciPy is a Python library used for scientific computing and technical computing. SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing,

Scikit-learn: Scikit-learn (also called sklearn) is a free software machine-learning library for the Python programming language. It features various classification, regression, and clustering algorithms including support--vector machines, random forests, gradient boosting, k-means, and DBSCAN.

Learning:

In this experiment, I was able to learn:

- the concepts of python language
- the use of various python libraries
- the use of several ML libraries for using built-in ML models.

Experiment – 2

Aim: To learn and implement data representation in Python

Theory:

THE LIBRARY KNOWN AS NUMPY, OR "NUMERICAL PYTHON," CONTAINS MULTIDIMENSIONAL ARRAY OBJECTS AND A SELECTION OF PROCEDURES FOR HANDLING SUCH ARRAYS. ARRAYS MAY BE SUBJECTED TO LOGICAL AND MATHEMATICAL OPERATIONS USING NUMPY. IT IS A LIBRARY MADE UP OF FUNCTIONS FOR HANDLING ARRAYS AND MULTIDIMENSIONAL ARRAY OBJECTS. LARGE, MULTI-DIMENSIONAL ARRAYS AND MATRICES ARE NOW SUPPORTED BY NUMPY, ALONG WITH A VAST VARIETY OF SOPHISTICATED MATHEMATICAL OPERATIONS THAT MAY BE PERFORMED ON THESE ARRAYS.

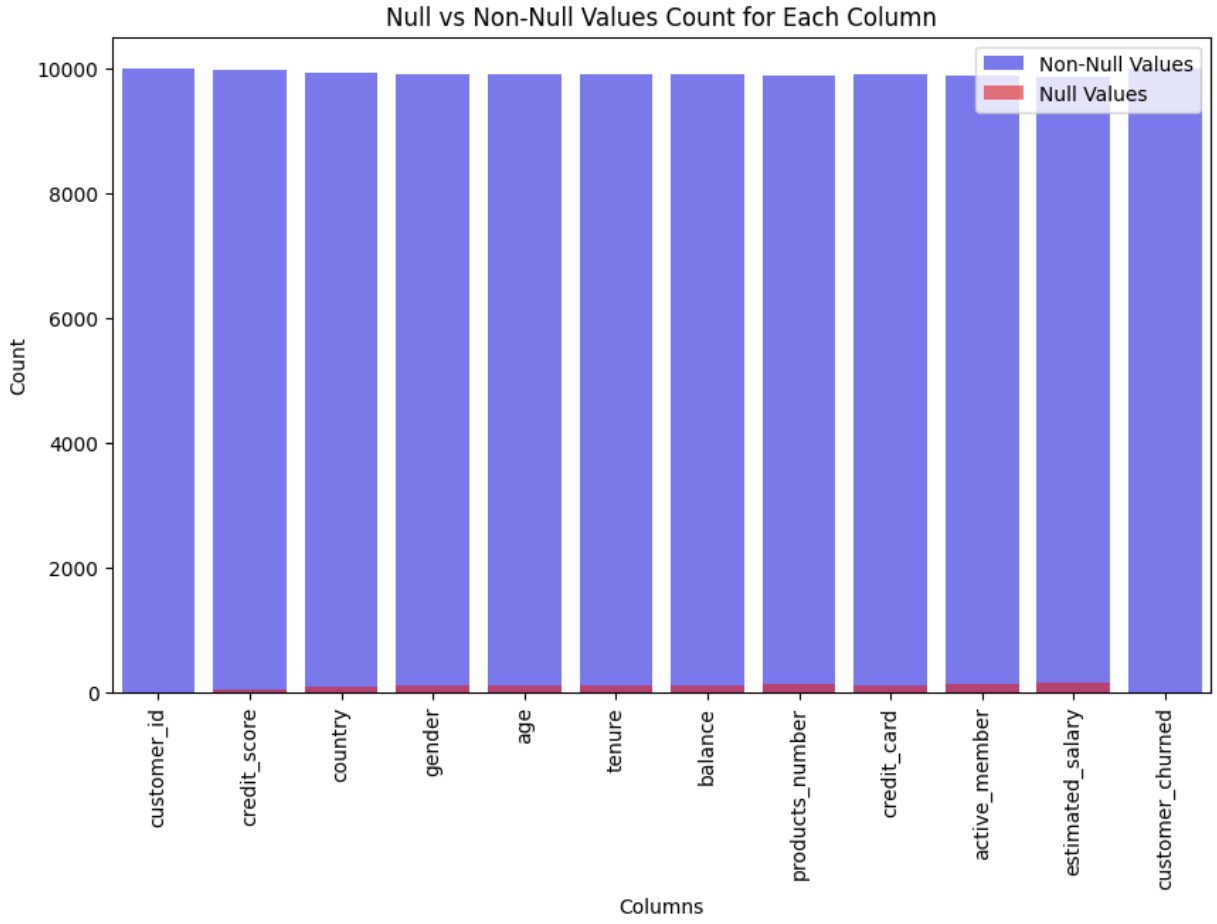
Code:

```
#importing libraries
import numpy as np
import pandas as pd
import matplotlib as plt
import seaborn as sns

#Loading the dataset
dataset = pd.read_csv("Bank Customer Churn Prediction.csv")

#printing first 5 instances of the dataset
dataset.head(5)

#number of dimensions in the dataset
print("Total number of dimension in the given dataset:")
print(dataset.ndim)

#shape of the dataset
print("Shape of the dataset:")
print(dataset.shape)

#removing null or missing values
dataset = dataset.dropna()

#Null vs Not null values graph
null_counts = dataset.isnull().sum()  # Count null values for each column
nonnull_counts = dataset.notnull().sum()  # Count non-null values for each column
# Create a plot to visualize the counts
plt.figure(figsize=(10, 6))
# Bar plot for non-null values
sns.barplot(x=nonnull_counts.index, y=nonnull_counts.values, color='blue',
alpha=0.6, label='Non-Null Values')
# Bar plot for null values
sns.barplot(x=null_counts.index, y=null_counts.values, color='red', alpha=0.6,
label='Null Values')
# Adding labels and title
plt.xlabel('Columns')
plt.ylabel('Count')
plt.title('Null vs Non-Null Values Count for Each Column')
plt.xticks(rotation=90)  # Rotate x-labels for better readability
plt.legend()
# Show plot
plt.show()
```

Output:

	customer_id	credit_score	country	gender	age	tenure	balance	products_number	credit_card	active_member	estimated_salary	customer_churned
0	15634602.0	619.0	France	Female	42.0	2.0	0.00	1.0	1.0	1.0	101348.88	1
1	15647311.0	608.0	Spain	Female	41.0	1.0	83807.86	1.0	0.0	1.0	112542.58	0
2	15619304.0	502.0	France	Female	42.0	8.0	159660.80	3.0	1.0	0.0	113931.57	1
3	15701354.0	NaN	France	Female	39.0	1.0	0.00	2.0	0.0	0.0	93826.63	0
4	15737888.0	850.0	Spain	Female	43.0	2.0	125510.82	1.0	1.0	1.0	79084.10	0

```
Total number of dimension in the given dataset:
2

Shape of the dataset:
(9995, 12)
```



Learning:

NumPy is often used along with packages like SciPy (Scientific Python) and Matplotlib (plotting library). Using NumPy, a developer can perform the following operations

- Mathematical and logical operations on arrays.
- Fourier transforms and routines for shape manipulation.
- Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.

Experiment – 3

Aim: Implementation of ID3 Decision Tree Classifier

Theory:

A decision tree is a tree-based data-structure that contains nodes (rectangular boxes) and edges (arrows) and is built from a dataset (table of columns representing features/attributes and rows corresponds to records). Each node is either used to make a decision (known as decision node) or represent an outcome (known as leaf node).

ID3 stands for Iterative Dichotomise 3 and is named such because the algorithm iteratively (repeatedly) dichotomizes(divides) features into two or more groups at each step. Invented by Ross Quinlan, ID3 uses a top-down greedy approach to build a decision tree. In simple words, the top-down approach means that we start building the tree from the top and the greedy approach means that at each iteration we select the best feature at the present moment to create a node. Most generally ID3 is only used for classification problems with nominal features only.

Metrics in ID3:

Entropy: Entropy is the measure of disorder and the Entropy of a dataset is the measure of disorder in the target feature of the dataset. In the case of binary classification (where the target column has only two types of classes) entropy is 0 if all values in the target column are homogenous(similar) and will be 1 if the target column has equal number values for both the classes.

$\text{Entropy}(S) = - \sum p_i * \log_2(p_i)$; $i = 1$ to n

where, n is the total number of classes in the target column; p_i is the probability of class 'i' or the ratio of "number of rows with class i in the target column" to the "total number of rows" in the dataset.

Information Gain: Information Gain calculates the reduction in the entropy and measures how well a given feature separates or classifies the target classes. The feature with the highest Information Gain is selected as the best one.

$IG(S, A) = \text{Entropy}(S) - \sum((|S_v| / |S|) * \text{Entropy}(S_v))$

where, S_v is the set of rows in S for which the feature column A has value v, $|S_v|$ is the number of rows in S_v and likewise $|S|$ is the number of rows in S.

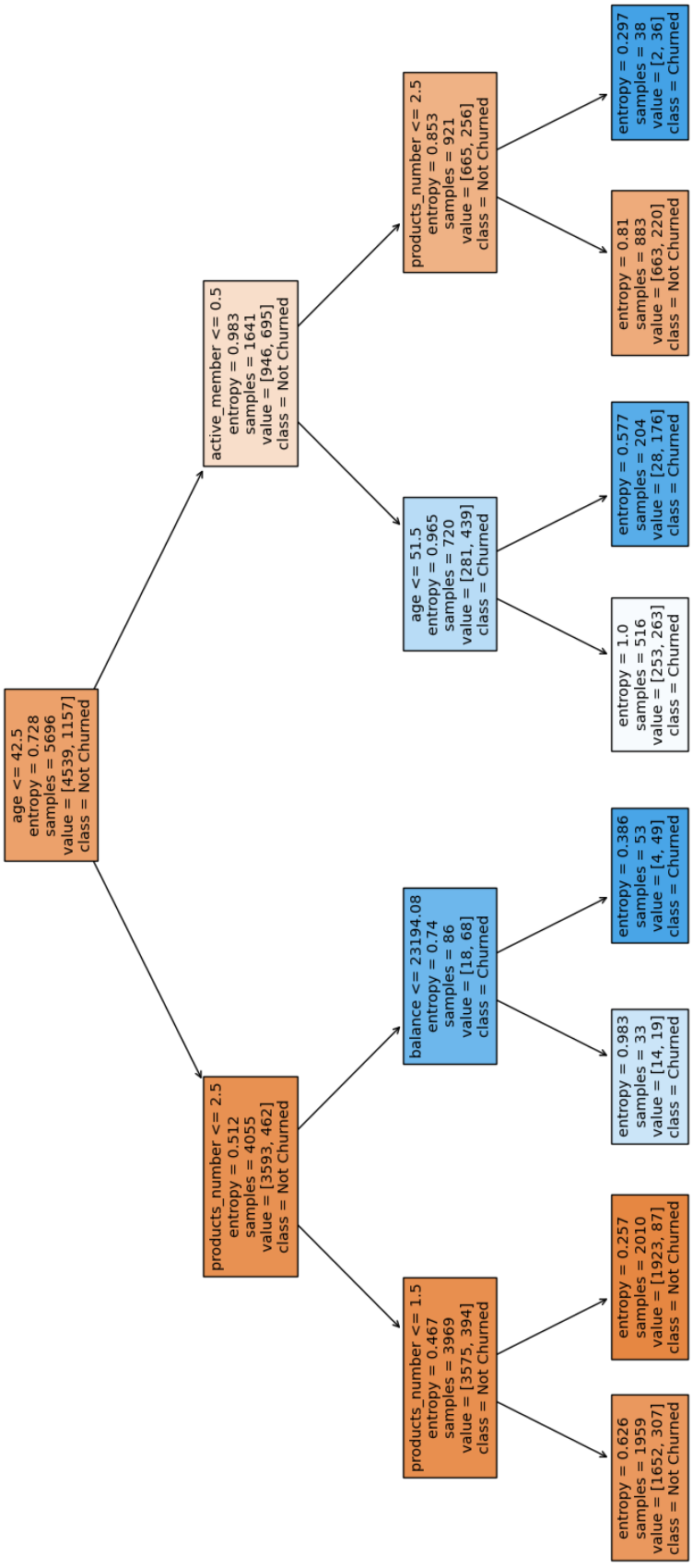
ID3 Steps:

1. Calculate the Information Gain of each feature.
2. Considering that all rows don't belong to the same class, split the dataset S into subsets using the feature for which the Information Gain is maximum.
3. Make a decision tree node using the feature with the maximum Information gain.
4. If all rows belong to the same class, make the current node as a leaf node with the class as its label.
5. Repeat for the remaining features until we run out of all features, or the decision tree has all leaf nodes.

Code:

```
dt_id3 = DecisionTreeClassifier(criterion='entropy', max_depth=3)
dt_id3.fit(X_train, Y_train)
plt.figure(figsize=(20,10))
plot_tree(dt_id3, feature_names=list(X.columns), class_names=['Not Churned',
'Churned'], filled=True)
plt.show()
```

Output:



Learning:

THE EXPERIMENT SHOULD HELP THE LEARNERS UNDERSTAND THE SIGNIFICANCE OF THE IMPLEMENTATION OF THE ID3 DECISION TREE CLASSIFIER.

Experiment – 4

Aim: Implementation of C 4.5 Decision Tree Classifier

Theory:

C4.5 is a decision tree algorithm developed by Ross Quinlan that is commonly used for machine learning and data mining tasks. It is an improvement over the earlier ID3 algorithm and can handle both continuous and discrete attributes.

The C4.5 algorithm builds a decision tree by recursively splitting the dataset based on the attribute that provides the most information gain. Information gain measures the reduction in entropy (or impurity) of the data when a given attribute is used for splitting. The algorithm stops when all instances in a node belong to the same class or when no more useful attributes can be found.

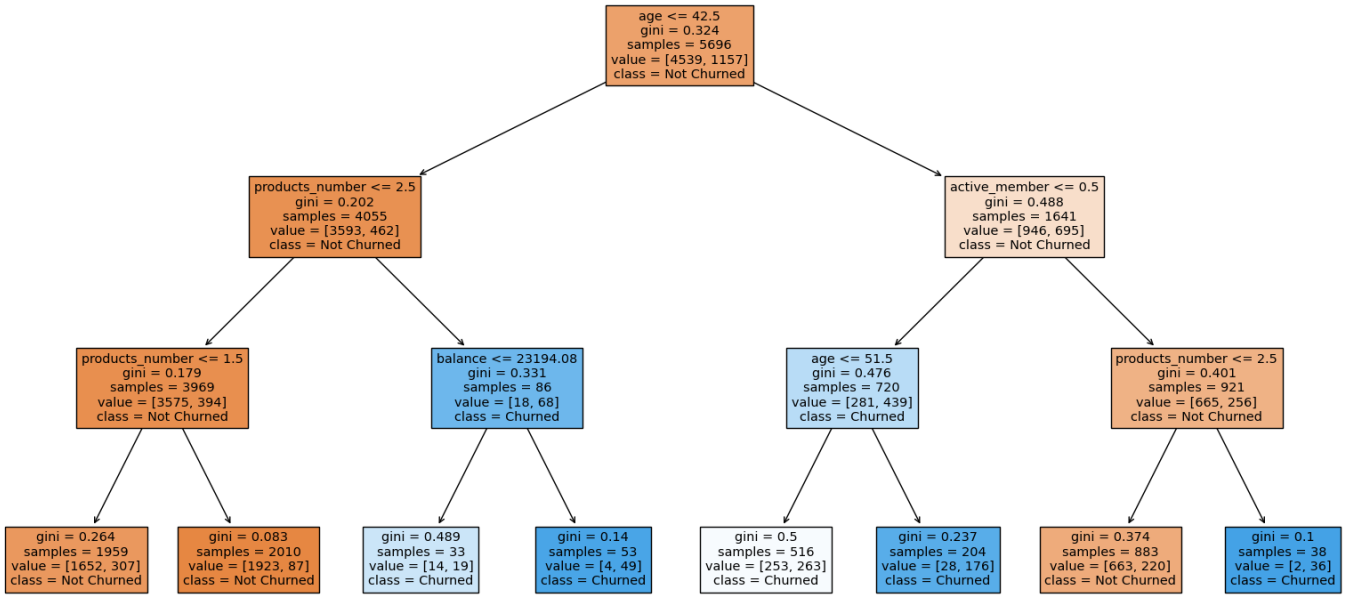
C4.5 also includes a mechanism for handling missing attribute values and for pruning the decision tree to avoid overfitting to the training data.

Once the decision tree is built, it can be used to classify new instances by traversing the tree from the root to a leaf node, where the predicted class is assigned based on the majority class of the training instances in that leaf node.

Code:

```
dt_c45 = DecisionTreeClassifier(criterion='gini', max_depth=3)
dt_c45.fit(X_train, Y_train)
plt.figure(figsize=(20,10))
plot_tree(dt_c45, feature_names=list(X.columns), class_names=['Not Churned',
'Churned'], filled=True)
plt.show()
```

Output:



Learning:

The experiment should help the learners understand the significance of the Implementation of the CART Decision Tree Classifier.

Experiment – 5

Aim: Implementation of CART Decision Tree Classifier

Theory:

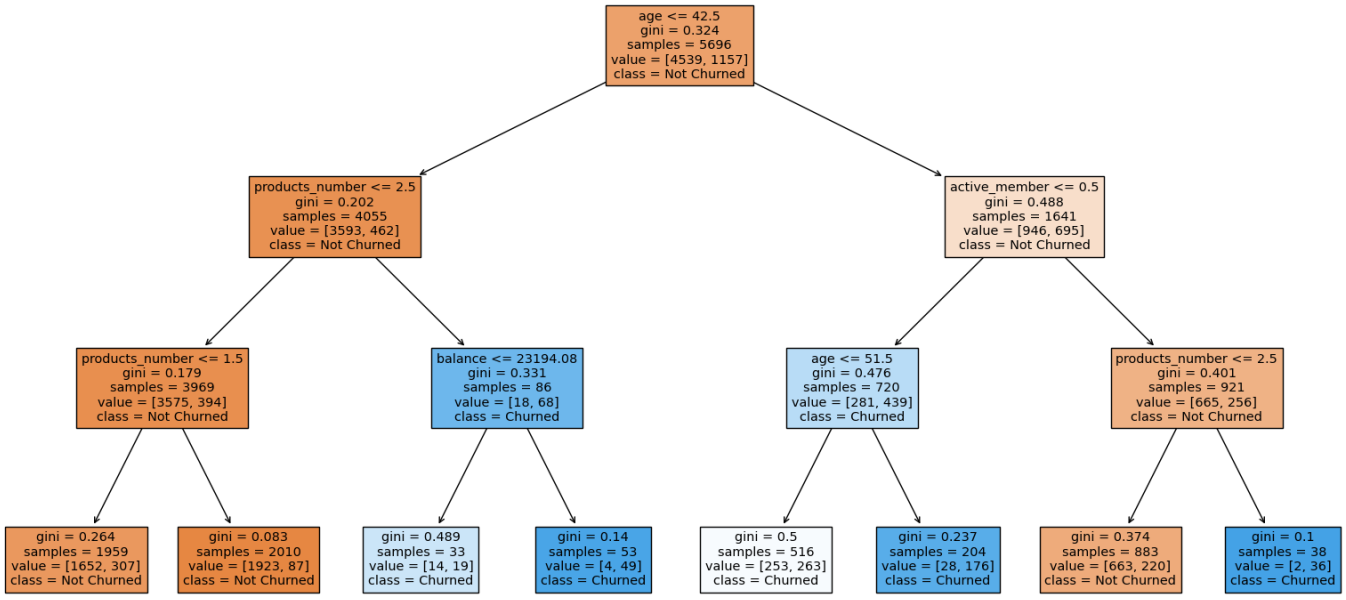
Decision tree learning uses a decision tree (as a predictive model) to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves). It is one of the predictive modelling approaches used in statistics, data mining and machine learning. Tree models where the target variable can take a discrete set of values are called classification trees; in these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. Decision trees where the target variable can take continuous values (typically real numbers) are called regression trees. Gini impurity is a measure of how often a randomly chosen element from the set would be incorrectly labelled if it was randomly labelled according to the distribution of labels in the subset.

The Gini impurity can be computed by summing the probability p(i) of an item with label i being chosen times the probability of a mistake in categorizing that item. It reaches its minimum (zero) when all cases in the node fall into a single target category.

Code:

```
dt_clf = DecisionTreeClassifier(criterion='gini', max_depth=3)
plt.figure(figsize=(20, 10))
plot_tree(dt_clf, feature_names=X.columns, class_names=['Not Churned', 'Churned'],
filled=True)
plt.show()
```

Output:



Experiment – 6

Aim: Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets

Theory:

An Artificial Neural Network (ANN) is a type of machine-learning model that is inspired by the human brain. It is composed of layers of interconnected neurons that process and transmit information. The Backpropagation algorithm is one of the most widely used algorithms for training ANN. It is a supervised learning algorithm that is used to adjust the weights of the network so that the network can learn to map inputs to outputs.

The Backpropagation algorithm works by calculating the error between the predicted output of the network and the actual output. It then adjusts the weights of the network in such a way that the error is reduced. This process is repeated over multiple iterations until the error is minimized.

Code:

```
import numpy as np

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size, learning_rate):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate

        # Initialize weights and biases randomly
        self.weights_input_hidden = np.random.randn(input_size, hidden_size)
        self.biases_input_hidden = np.random.randn(hidden_size)
        self.weights_hidden_output = np.random.randn(hidden_size, output_size)
        self.biases_hidden_output = np.random.randn(output_size)

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    def forward_pass(self, inputs):
        # Calculate hidden layer activations
        hidden_inputs = np.dot(inputs, self.weights_input_hidden) +
self.biases_input_hidden
        hidden_outputs = self.sigmoid(hidden_inputs)

        # Calculate output layer activations
        final_inputs = np.dot(hidden_outputs, self.weights_hidden_output) +
self.biases_hidden_output
        final_outputs = self.sigmoid(final_inputs)

        return final_outputs, hidden_outputs

    def backward_pass(self, inputs, targets, outputs, hidden_outputs):
        # Compute output layer error
        output_errors = targets - outputs
        output_gradients = output_errors * self.sigmoid_derivative(outputs)
```

```
        # Update weights and biases for hidden to output layer
        self.weights_hidden_output += np.dot(hidden_outputs.T, output_gradients) *
self.learning_rate
        self.biases_hidden_output += np.sum(output_gradients, axis=0) *
self.learning_rate

        # Compute hidden layer error
        hidden_errors = np.dot(output_gradients, self.weights_hidden_output.T)
        hidden_gradients = hidden_errors * self.sigmoid_derivative(hidden_outputs)

        # Update weights and biases for input to hidden layer
        self.weights_input_hidden += np.dot(inputs.T, hidden_gradients) *
self.learning_rate
        self.biases_input_hidden += np.sum(hidden_gradients, axis=0) *
self.learning_rate

    def train(self, inputs, targets, epochs):
        for epoch in range(epochs):
            outputs, hidden_outputs = self.forward_pass(inputs)
            self.backward_pass(inputs, targets, outputs, hidden_outputs)

            if epoch % 100 == 0:
                loss = np.mean(np.square(targets - outputs))
                print(f"Epoch {epoch}, Loss: {loss}")

# Example usage
# Define input, hidden, and output layer sizes
input_size = 10
hidden_size = 5
output_size = 1

# Create a neural network
nn = NeuralNetwork(input_size, hidden_size, output_size, learning_rate=0.1)

# Generate synthetic dataset
X_train = np.random.rand(100, input_size)
y_train = np.random.randint(0, 2, size=(100, output_size))

# Train the neural network
nn.train(X_train, y_train, epochs=1000)
```

Output:

```
Epoch 0, Loss: 0.37320200391211117
Epoch 100, Loss: 0.23476138652802273
Epoch 200, Loss: 0.21115199101453416
Epoch 300, Loss: 0.19607072200874393
Epoch 400, Loss: 0.18606742759657494
Epoch 500, Loss: 0.17158831366066388
Epoch 600, Loss: 0.1595023603992833
Epoch 700, Loss: 0.14707143430528563
Epoch 800, Loss: 0.13667280895709216
Epoch 900, Loss: 0.12822250773232358
```

Learning:

Implementing the Backpropagation algorithm to train an ANN can help in achieving high accuracy in predicting outcomes. By testing the ANN with appropriate datasets, one can analyse the effectiveness of the algorithm in different scenarios. The experiment can also help in understanding the impact of different hyperparameters like the number of layers, number of neurons, and learning rate on the performance of the ANN. Additionally, the experiment can also help in identifying the limitations of the algorithm and the dataset used.

Experiment – 7

Aim: Write a program to implement the Naïve Bayesian classifier for the appropriate dataset and compute the performance measures of the model

Theory:

The Naïve Bayesian classifier is a probabilistic algorithm used in machine learning for classification tasks. It is based on Bayes' theorem and assumes that the presence of a particular feature in a class is independent of the presence of any other feature in the same class. The Naïve Bayesian classifier calculates the probability of a given data point belonging to a certain class by computing the conditional probability of each feature given the class and multiplying them together.

To implement the Naïve Bayesian classifier, we first need to train the model using a dataset with labelled data. The labelled data is used to estimate the conditional probabilities of each feature given in each class. Then, the model can be used to predict the class of new data points based on their features.

Performance measures of the model can be computed using metrics such as accuracy, precision, recall, and F1 score. Accuracy is the ratio of correctly predicted data points to the total number of data points. Precision is the ratio of true positive predictions to the total number of positive predictions, while recall is the ratio of true positive predictions to the total number of actual positive instances. The F1 score is a weighted average of precision and recall.

Code:

```
import numpy as np

class NaiveBayesClassifier:
    def __init__(self):
        self.class_probabilities = None
        self.feature_probabilities = None

    def fit(self, X_train, y_train):
        num_samples, num_features = X_train.shape
        self.classes = np.unique(y_train)
        num_classes = len(self.classes)

        # Calculate class probabilities
        self.class_probabilities = np.zeros(num_classes)
        for i, c in enumerate(self.classes):
            self.class_probabilities[i] = np.sum(y_train == c) / num_samples

        # Calculate feature probabilities
        self.feature_probabilities = {}
        for c in self.classes:
            self.feature_probabilities[c] = []
            for feature_index in range(num_features):
                feature_values = X_train[y_train == c, feature_index]
                probabilities = {}
                for value in np.unique(X_train[:, feature_index]):
                    probabilities[value] = np.sum(feature_values == value) /
len(feature_values)
                self.feature_probabilities[c].append(probabilities)

    def predict(self, X_test):
        num_samples = X_test.shape[0]
        predictions = np.zeros(num_samples, dtype=int)
```

```
        for i in range(num_samples):
            probabilities = []
            for j, c in enumerate(self.classes):
                class_probability = np.log(self.class_probabilities[j])
                for feature_index, value in enumerate(X_test[i]):
                    if value in self.feature_probabilities[c][feature_index]:
                        class_probability +=
np.log(self.feature_probabilities[c][feature_index][value])
                probabilities.append(class_probability)
            predictions[i] = self.classes[np.argmax(probabilities)]
        return predictions
# Example usage
X_train = np.array([[1, 1], [1, 0], [0, 1], [0, 0]])
y_train = np.array([1, 1, 0, 0])

X_test = np.array([[1, 1], [0, 0]])

nb_classifier = NaiveBayesClassifier()
nb_classifier.fit(X_train, y_train)
predictions = nb_classifier.predict(X_test)

print("Predictions:", predictions)
```

Output:

```
Predictions: [1 0]
```

Learning:

IMPLEMENTING THE NAÏVE BAYESIAN CLASSIFIER FOR A DATASET CAN HELP IN ACHIEVING ACCURATE CLASSIFICATION OF DATA POINTS. THE PERFORMANCE MEASURES COMPUTED FOR THE MODEL CAN HELP IN ANALYSING THE EFFECTIVENESS OF THE ALGORITHM IN DIFFERENT SCENARIOS. THE EXPERIMENT CAN ALSO HELP IN UNDERSTANDING THE IMPACT OF DIFFERENT FEATURES ON THE PERFORMANCE OF THE CLASSIFIER.

ADDITIONALLY, THE EXPERIMENT CAN ALSO PROVIDE INSIGHTS INTO THE IMPORTANCE OF FEATURE SELECTION AND DATA PREPROCESSING FOR THE NAÏVE BAYESIAN CLASSIFIER. BY TESTING THE CLASSIFIER WITH DIFFERENT DATASETS, ONE CAN IDENTIFY THE STRENGTHS AND LIMITATIONS OF THE ALGORITHM AND IMPROVE ITS PERFORMANCE.

Experiment – 8

Aim: Write a program to implement the k-nearest Neighbour algorithm to classify any dataset and compute the performance measures of the model

Theory:

THE K-NEAREST NEIGHBOUR (K-NN) ALGORITHM IS A SIMPLE YET POWERFUL ALGORITHM USED FOR CLASSIFICATION TASKS. IT IS A NON-PARAMETRIC ALGORITHM THAT STORES ALL THE AVAILABLE DATA POINTS IN MEMORY AND CLASSIFIES NEW DATA POINTS BASED ON THEIR PROXIMITY TO THE EXISTING DATA POINTS. THE K-NN ALGORITHM WORKS BY FIRST CALCULATING THE DISTANCE BETWEEN THE NEW DATA POINT AND ALL THE EXISTING DATA POINTS IN THE DATASET. IT THEN SELECTS THE K NEAREST DATA POINTS AND ASSIGNS THE CLASS TO THE NEW DATA POINT BASED ON THE MAJORITY CLASS OF THESE K NEAREST DATA POINTS.

TO IMPLEMENT THE K-NN ALGORITHM, WE FIRST NEED TO TRAIN THE MODEL USING A DATASET WITH LABELLED DATA. THE LABELLED DATA IS USED TO STORE THE DATA POINTS AND THEIR CORRESPONDING CLASS LABELS IN MEMORY. THEN, THE MODEL CAN BE USED TO PREDICT THE CLASS OF NEW DATA POINTS BASED ON THEIR PROXIMITY TO THE EXISTING DATA POINTS.

PERFORMANCE MEASURES OF THE MODEL CAN BE COMPUTED USING METRICS SUCH AS ACCURACY, PRECISION, RECALL, AND F1 SCORE. ACCURACY IS THE RATIO OF CORRECTLY PREDICTED DATA POINTS TO THE TOTAL NUMBER OF DATA POINTS. PRECISION IS THE RATIO OF TRUE POSITIVE PREDICTIONS TO THE TOTAL NUMBER OF POSITIVE PREDICTIONS, WHILE RECALL IS THE RATIO OF TRUE POSITIVE PREDICTIONS TO THE TOTAL NUMBER OF ACTUAL POSITIVE INSTANCES. THE F1 SCORE IS A WEIGHTED AVERAGE OF PRECISION AND RECALL.

Code:

```
import numpy as np
from collections import Counter

class KNNClassifier:
    def __init__(self, k=3):
        self.k = k

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train

    def predict(self, X_test):
        predictions = []
        for x in X_test:
            distances = np.linalg.norm(self.X_train - x, axis=1)
            nearest_indices = distances.argsort()[:self.k]
            nearest_labels = self.y_train[nearest_indices]
            most_common_label = Counter(nearest_labels).most_common(1)[0][0]
            predictions.append(most_common_label)
        return np.array(predictions)

def accuracy(y_true, y_pred):
    return np.mean(y_true == y_pred)

# Example usage
X_train = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])
y_train = np.array([0, 0, 1, 1])
X_test = np.array([[1, 2.5], [3.5, 4]])

knn_classifier = KNNClassifier(k=2)
knn_classifier.fit(X_train, y_train)
predictions = knn_classifier.predict(X_test)
```

```
print("Predictions:", predictions)

# Compute accuracy
y_test = np.array([0, 1])
print("Accuracy:", accuracy(y_test, predictions))
```

Output:

```
Predictions: [0 1]
Accuracy: 1.0
```

Learning:

Implementing the k-NN algorithm for a dataset can help in achieving accurate classification of data points. The performance measures computed for the model can help in analysing the effectiveness of the algorithm in different scenarios. The experiment can also help in understanding the impact of different values of k on the performance of the algorithm.

Additionally, the experiment can also provide insights into the importance of feature selection and data preprocessing for the k-NN algorithm. By testing the algorithm with different datasets, one can identify the strengths and limitations of the algorithm and improve its performance.

Experiment – 9

Aim: Apply the k-Means Clustering algorithm on suitable datasets and comment on the quality of clustering

Theory:

The k-Means Clustering algorithm is a popular unsupervised machine learning algorithm used for clustering tasks. It is a simple yet effective algorithm that partitions the given dataset into k clusters based on the similarity of the data points. The algorithm works by first selecting k initial centroids randomly from the dataset. It then assigns each data point to the nearest centroid and recalculates the centroid for each cluster. This process is repeated until convergence, i.e., when the assignment of data points to clusters no longer changes.

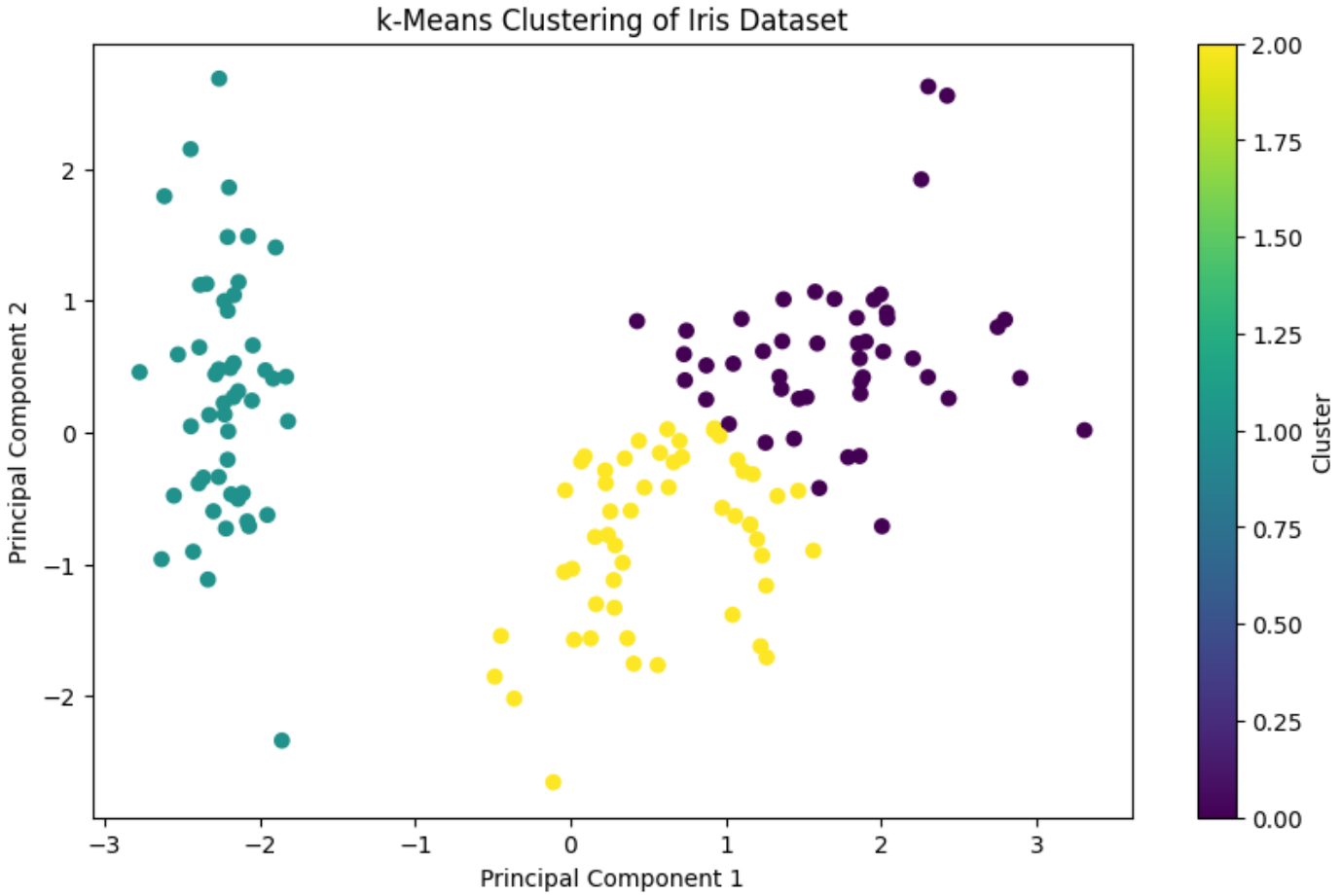
To apply the k-Means Clustering algorithm, we first need to select a suitable dataset with unlabelled data. The algorithm will automatically identify patterns in the data and cluster similar data points together. We can then evaluate the quality of clustering by using metrics such as the Silhouette score, Calinski-Harabasz index, or Davies-Bouldin index.

The Silhouette score measures the similarity of data points within a cluster compared to other clusters. The Calinski-Harabasz index measures the ratio of the between-cluster dispersion to the within-cluster dispersion. The Davies-Bouldin index measures the average similarity between each cluster and its most similar cluster.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

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

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

# Apply PCA for visualization
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# Apply k-Means clustering
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(X_scaled)
y_pred = kmeans.predict(X_scaled)

# Visualize the clusters
plt.figure(figsize=(10, 6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y_pred, cmap='viridis')
plt.title('k-Means Clustering of Iris Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.colorbar(label='Cluster')
plt.show()
```

Output:



Learning:

Applying the k-Means Clustering algorithm to a suitable dataset can provide insights into the structure of the data and the similarity of the data points. The quality of clustering can be evaluated using various metrics, which can help in selecting the optimal value of k and improving the performance of the algorithm.

Additionally, the experiment can also provide insights into the importance of feature selection and data preprocessing for the k-means Clustering algorithm. By testing the algorithm with different datasets, one can identify the strengths and limitations of the algorithm and improve its performance.

Experiment – 10

Aim: Write a program to implement ensemble algorithms- AdaBoost and Bagging using the appropriate dataset and evaluate their performance on that dataset

Theory:

Ensemble learning is a popular machine learning technique used to improve the accuracy of predictive models. It involves combining multiple models to make more accurate predictions than a single model. Two common ensemble learning algorithms are AdaBoost and Bagging.

AdaBoost (Adaptive Boosting) is a boosting algorithm that trains multiple weak classifiers sequentially. Each weak classifier is trained on a subset of the data and assigned a weight. In each subsequent round, the algorithm focuses on the misclassified samples from the previous round and adjusts the weights of the samples. The final prediction is made by taking a weighted average of the predictions of all the weak classifiers.

Bagging (Bootstrap Aggregating) is a parallel ensemble algorithm that trains multiple instances of a base classifier on random subsets of the data. The final prediction is made by taking a majority vote on the predictions of all the base classifiers.

To implement AdaBoost and Bagging, we first need to select a suitable dataset with labelled data. We can then train multiple instances of the base classifier using AdaBoost or Bagging and evaluate their performance using metrics such as accuracy, precision, recall, and F1 score.

Code:

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

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

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Define base classifier
base_classifier = DecisionTreeClassifier(max_depth=1)

# AdaBoost
ada_boost = AdaBoostClassifier(base_estimator=base_classifier, n_estimators=50,
random_state=42)
ada_boost.fit(X_train, y_train)
y_pred_ada = ada_boost.predict(X_test)
accuracy_ada = accuracy_score(y_test, y_pred_ada)
print("Accuracy of AdaBoost:", accuracy_ada)

# Bagging
bagging = BaggingClassifier(base_estimator=base_classifier, n_estimators=50,
random_state=42)
bagging.fit(X_train, y_train)
y_pred_bagging = bagging.predict(X_test)
accuracy_bagging = accuracy_score(y_test, y_pred_bagging)
```

```
print("Accuracy of Bagging:", accuracy_bagging)
```

Output:

```
Accuracy of AdaBoost: 1.0

Accuracy of Bagging: 1.0
```

Learning:

Implementing ensemble algorithms such as AdaBoost and Bagging can help in improving the accuracy and robustness of predictive models. The performance measures computed for the models can help in analysing the effectiveness of the algorithms in different scenarios. The experiment can also help in understanding the impact of different hyperparameters on the performance of the algorithms.

Additionally, the experiment can provide insights into the importance of feature selection and data preprocessing for ensemble learning algorithms. By testing the algorithms with different datasets, one can identify the strengths and limitations of the algorithms and improve their performance