Chittagong University of Engineering and Technology

Department of Computer Science & Engineering

Final Report on Machine Learning Algorithms Implementation

Course Name: Machine Learning (Sessional)

Course Code: CSE-364

Course Teachers

Md. Rashadur Rahman

Lecturer, Dept. of CSE, CUET

Hasan Murad

Lecturer, Dept of CSE, CUET

Submitted By

S.M. Rifatur Rana

1804098

Date of Submission: 31-08-2023

Contents

Table of Figures

Abstract

This report documents the practical implementation of five foundational machine learning algorithms on sample datasets. The algorithms examined include association rule mining using Apriori, predictive modeling via multivariable linear regression, clustering through K-Means, classification with decision trees, and deep learning using artificial neural networks. The hands-on implementation provided valuable experience in end-to-end workflow - from data preprocessing to model building, evaluation, and results analysis. Key outcomes and insights from each algorithm are highlighted in the report, along with a comparative discussion of their real-world applicability. Challenges faced during implementation are addressed and potential improvements proposed. Overall, the project enhanced understanding of core machine learning techniques and developed problem-solving skills through practice. The report summarizes the students' achievement in applying theoretical knowledge to construct, analyze and interpret machine learning models for given tasks.

1 Introduction

This report aims to demonstrate and evaluate practical understanding of important machine learning algorithms through hands-on implementation.Five algorithms have been implemented on sample datasets - Apriori, Linear Regression, K-Means, Decision Tree, and Artificial Neural Network. Implementing the end-to-end workflow from data preparation to model building and evaluation is an integral part of the course curriculum. The significance of this report is in exhibiting the ability to operationalize theoretical concepts learned in class. The process involves comprehending algorithm logic, coding in Python, selecting parameters, training models, assessing performance, and interpreting results. This develops key skills needed for a career in machine learning including programming, analytical thinking, and problem-solving. The algorithms represent foundational techniques for association, classification, clustering, and prediction. By constructing complete workflows, we gained a holistic perspective needed for applying models to real-world systems.

Overall, the report evaluates our grasp over the inner workings, applications, and nuances of the implemented machine learning algorithms.

A brief overview of the five machine learning algorithms is given below in concise and pointwise manner:

Apriori:

- Used for association rule mining to uncover relationships between items in large databases
- Employs a "bottom up" approach to find frequent itemsets that meet minimum support thresholds
- Iteratively generates candidate itemsets and prunes infrequent ones
- Association rules are derived from frequent itemsets meeting minimum confidence

Multivariable Linear Regression:

- Regression technique to model linear relationship between multiple predictor variables and a response variable
- Fits a linear equation by estimating coefficients for each input variable
- Coefficients are estimated using least squares to minimize the sum of squared residuals
- Useful for prediction, forecasting, and determining strength of variable relationships

K-Means Clustering:

- Unsupervised learning algorithm that partitions dataset into k clusters
- Aims to minimize within-cluster variance and maximize inter-cluster variance
- Initializes k random points as cluster centers and assigns points to their closest center
- Iteratively updates cluster centers based on current assignment of points
- Widely used for exploratory data analysis to find intrinsic grouping

Decision Tree:

- Supervised learning method for classification and regression tasks
- Recursively splits data based on feature values to construct a tree structure
- Aims to maximize information gain at each split to reduce impurity
- Makes predictions by traversing the decision tree based on input features
- Pruning helps avoid overfitting and improves generalization

Artificial Neural Network:

- Inspired by biological neural networks and used for deep learning
- Contains layers of interconnected nodes with numeric weights and biases
- Learns complex relationships between inputs and outputs through backpropagation
- Capable of modeling highly nonlinear functions and feature hierarchies
- Wide range of architectures like multi-layer perceptron, CNN, RNN etc.

2 Algorithm Descriptions and Implementations

2.1 Apriori Algorithm

Association rule mining is the process of identifying strong relationships between items in a dataset. It analyzes data to discover combinations of items that frequently appear together, known as frequent item sets. These correlations can be expressed as association rules, which have an antecedent (if) and a consequent (then).

For example, an association rule derived from grocery store data may be:

{bread, eggs} -> {milk}

This implies that if a customer buys bread and eggs together, they are also likely to buy milk.

The purpose of the Apriori algorithm is to efficiently find all frequent itemsets that satisfy a minimum support threshold. It employs a "bottom up" approach, iteratively generating candidate itemsets and pruning those below the threshold. The algorithm starts by scanning all items to find frequent 1-itemsets. It then combines these to generate 2-itemset candidates which are again pruned based on support. This continues for longer itemsets until no more frequent candidates are found.

The key advantages of Apriori are that it reduces the search space by pruning and employs a breadth-first traversal to efficiently enumerate itemsets. The final frequent itemsets can be analyzed to uncover interesting correlations which are expressed as association rules. Therefore, Apriori is extremely useful for market basket analysis and other association mining tasks.

2.1.1 Dataset

The given dataset contains transactions representing grocery items purchased together. Each transaction contains a subset of items from the overall item catalog.

Attributes:

- Transaction - Each row represents a transaction showing items purchased together
- Item - Name of the item purchased in that transaction

Characteristics:

- The dataset contains more than 7000transactions
- There are around 115 unique items represented across all transactions
- Average transaction length is 4-6 items

- Contains frequently purchased grocery items like milk, bread, eggs
- Also includes discretionary purchases like chocolate and wine
- Exhibits sparsity with not all items appearing in all transactions

7460	red wine	honey	rice	chocolate	french fries	brownies	body spray	light mayo				
7461	fromage blanc	escalope	pasta	mushroom cream sauce								
7462	frozen vegetables	tomatoes	ground beef	spaghetti	mineral water	fromage blanc	pancakes	whole wheat rice	blueberries	cereals	escalope	mayonnaise
7463	burgers	eggs	cake	cereals								
7464	eggs											
7465	herb & pepper	ground beef	eggs	whole wheat rice	cake	oil	fresh bread	low fat yogurt				
7466	eggs	french fries	brownies	low fat yogurt								
7467	yogurt cake											
7468	turkey	burgers	grated cheese	shrimp	pasta	spaghetti	mineral water	eggs	oil	chocolate	yogurt cake	zucchini
7469	herb & pepper	eggs										
7470	whole wheat pasta	mineral water	light cream	chicken	french fries							
7471	french fries	pancakes										
7472	mineral water	chocolate										
7473	chocolate	grated cheese	mineral water	olive oil	green tea							

Figure 1: Snapshot of dataset (store_data.csv)

2.1.2 Implementation

Step-01: Load data

- Read CSV file
- Store each transaction as a list in a transactions list

```
def load_data(filename):
    transactions = []
    with open(filename, 'r') as file:
        for line in file:
            transaction = line.strip().split(',')
            transactions.append(transaction)
    return transactions
transactions = load_data('store_data.csv')
```

Step-02: Generating Candidates

The code generates new potential itemsets of size k by combining pairs of itemsets from the previous iteration that share a common prefix of size k-1. These new candidates are prepared for further analysis.The generated candidate itemsets will be used in subsequent steps of a data mining algorithm, such as Apriori, to identify frequent itemsets and extract meaningful patterns from large datasets.

```
def generate_candidates(prev_candidates, k):
    candidates = {}
    prev_candidates_list = list(prev_candidates.keys())
```

```
    for i in range(len(prev_candidates_list)):
        for j in range(i + 1, len(prev_candidates_list)):
            itemset1 = prev_candidates_list[i]
            itemset2 = prev_candidates_list[j]

            items1 = sorted(list(itemset1))
            items2 = sorted(list(itemset2))

            if items1[:k-2] == items2[:k-2]:
                new_itemset = tuple(sorted(set(items1 + items2)))
                candidates[new_itemset] = 0
    return candidates
```

Step-03: Find frequent itemsets

- Count occurrences of each item
- Keep items meeting minimum support threshold
- Iteratively generate candidate itemsets of increasing size
- Prune candidates below support threshold
- Frequent itemsets are those meeting threshold

```
def generate_frequent_itemsets(transactions, min_support):
    itemsets = {}
    candidates = {}
    frequent_itemsets = {}
    for transaction in transactions:
        for item in transaction:
            itemsets[item] = itemsets.get(item, 0) + 1
    for item, count in itemsets.items():
        support = count / len(transactions)
        if support >= min_support:
            frequent_itemsets[(item,)] = support
            candidates[(item,)] = count
    k = 2
```

```
    while candidates:
        candidates = generate_candidates(candidates, k)
        for transaction in transactions:
            for candidate in candidates:
                if set(candidate).issubset(set(transaction)):
                    candidates[candidate] += 1
        candidates = {itemset: count for itemset, count in candidates.items()
                      if count / len(transactions) >= min_support}
        frequent_itemsets.update(candidates)
        k += 1
    return frequent_itemsets
```

Step-04: Generate association rules

- For each frequent itemset, create all subsets
- Check if subset and remaining items are frequent
- Calculate confidence for rule: subset -> remaining
- Retain rules meeting minimum confidence

```
def generate_association_rules(frequent_itemsets, min_confidence):
    association_rules = []

    for itemset in frequent_itemsets:
        if len(itemset) >= 2:
            subsets = list(combinations(itemset, 1))
            for i in range(1, len(itemset)):
                subsets.extend(list(combinations(itemset, i)))

            for subset in subsets:
                remaining = tuple(sorted(set(itemset) - set(subset)))
                if subset in frequent_itemsets and remaining in frequent_itemsets:
                    confidence = frequent_itemsets[itemset] / frequent_itemsets[subset]
                    if confidence >= min_confidence:
                        association_rules.append((subset, remaining, confidence))
```

```
    return association_rules
```

Step-05: Main

- Call function to find frequent itemsets
- Pass transactions and minimum support
- Call function to generate rules from frequent itemsets
- Pass minimum confidence threshold

```
min_support = 0.01
min_confidence = 0.01

frequent_itemsets = generate_frequent_itemsets(transactions, min_support)
association_rules = generate_association_rules(frequent_itemsets, min_confidence)

print("Frequent Itemsets:")
for itemset, support in frequent_itemsets.items():
    print(itemset, "Support:", support)

print("\nAssociation Rules:")
for rule in association_rules:
    antecedent, consequent, confidence = rule
    print(antecedent, "->", consequent, "Confidence:", confidence)
```

2.1.3 Performance Evaluation

Snapshot of the association rules:

```
('avocado', 'mineral water') Support: 87
('low fat yogurt', 'mineral water') Support: 180
('eggs', 'low fat yogurt') Support: 126
('low fat yogurt', 'milk') Support: 99
('french fries', 'low fat yogurt') Support: 100
('frozen vegetables', 'low fat yogurt') Support: 76
('low fat yogurt', 'spaghetti') Support: 114
('chocolate', 'low fat yogurt') Support: 111
('green tea', 'mineral water') Support: 233
('frozen smoothie', 'green tea') Support: 84
```

Figure 2: Snapshot of association rules

2.2 Multivariable Linear Regression

Multivariable linear regression is a statistical method that predicts outcomes using two or more independent variables. These are also known as explanatory variables, while the outcome is the dependent variable.

The aim is to find a linear relationship between the independent variables and the outcome. The equation looks like this:

Y = b0 + b1X1 + b2X2 + ... + bnXn

Here:

Y is the outcome.

b0 is the intercept.

b1, b2, ..., bn are coefficients for independent variables.

X1, X2, ..., Xn are the independent variables.

Coefficients (b0, b1, ..., bn) are determined through ordinary least squares (OLS), minimizing the difference between actual and predicted Y values.

This technique helps predict various outcomes, such as house prices based on features like size, bedrooms, and location.

2.2.1 Dataset

The dataset has the following columns, as follows:

title: The title of the property listing

beds: The number of bedrooms in the property

bath: The number of bathrooms in the property

area: The square footage of the property

type: The type of property (e.g., single-family home, condo, apartment)

purpose: The purpose of the property (e.g., for sale, for rent)

floorPlan: The floor plan of the property

url: The URL of the property listing

lastUpdated: The date the property listing was last updated

price: The price of the property

The target variable is the price of the property. The features are the remaining 10 columns.

The data is also relatively recent, which makes it a good representation of the current market conditions.

title	beds	bath	area	adress	type	purpose	flooPlan	url	lastUpdate	price
Eminent Apartment Of 2200 Sq Ft Is Vacant For Rent In Bashur	3	4	2,200 sqft	Block A, B	Apartment	For Rent	https://im	https://wv	########	50 Thousand
Apartment Ready To Rent In South Khulshi, Nearby South Khul	3	4	1,400 sqft	South Khul	Apartment	For Rent	https://im	https://wv	25-Jan-22	30 Thousand
Smartly priced 1950 SQ FT apartment, that you should check in	3	4	1,950 sqft	Block F, B	Apartment	For Rent	https://im	https://wv	########	30 Thousand
2000 Sq Ft Residential Apartment Is Up For Rental Purpose In S	3	3	2,000 sqft	Sector 9, L	Apartment	For Rent	https://im	https://wv	########	35 Thousand
Strongly Structured This 1650 Sq. ft Apartment Is Now Vacant	3	4	1,650 sqft	Block I, Ba	Apartment	For Rent	https://im	https://wv	########	25 Thousand
A nice residential flat of 3400 SQ FT, for rent, can be found in C	5	5	3,400 sqft	Gulshan 1,	Apartment	For Rent	https://im	https://wv	########	1.1 Lakh
1600 Square Feet Apartment With Amazing Rooms Is For Rent	3	3	1,600 sqft	Sector 6, L	Apartment	For Rent	https://im	https://wv	6-Aug-22	35 Thousand
Let Us Help You To Rent This 1250 Sq Ft Apartment Which Is N	3	3	1,250 sqft	Block K, B	Apartment	For Rent	https://im	https://wv	4-Jan-23	23 Thousand

Figure 3: Snapshot of property dataset

2.2.2 Implementation

Step-01: Data preprocessing

- Reading a CSV file containing property listing data into a panda DataFrame.
- Filtering out specific property types ('Duplex' and 'Building') from the data.
- Dropping unnecessary columns from the DataFrame.
- Extracting and converting numeric values from columns like 'bath', 'beds', and 'area'.
- Cleaning and converting 'area' values to a numeric format.
- Standardizing 'price' values to a common unit (thousands) for consistency.

```
import csv
import numpy as np
import statistics
import pandas as pd
data = pd.read_csv('property_listing_data_in_Bangladesh.csv')
new_data=data[(data['type'] != 'Duplex') & (data['type'] != 'Building')]
new_data['type']
new_data = new_data.drop(['title', 'adress', 'type', 'purpose', 'flooPlan', 'url', 'lastUpdated'] ,axis=1)
new_data['bath'] = new_data['bath'].str.extract('(\d+)')
new_data['bath'] = pd.to_numeric(new_data['bath'])
new_data['beds'] = new_data['beds'].str.extract('(\d+)')
new_data['beds'] = pd.to_numeric(new_data['beds'])
new_data['area'] = new_data['area'].str.replace(',', '')
new_data['area'] = new_data['area'].str.extract('(\d+)')
new_data['area'] = pd.to_numeric(new_data['area'])
new_data['price'] = new_data['price'].apply(lambda x: float(x.split(' ')[0]) * 100000.0 if 'Lakh' in x else float(x.split(' ')[0]) * 1000)
```

Stpe-02: Scaling, and Splitting for Predictive Modeling

- Import libraries including train_test_split from Scikit-Learn and np from NumPy.
- Define z_score_scaling function for Z-score normalization.
- Convert DataFrame new_data to NumPy array modified_list.
- Separate input features (X) and target variable (Y).
- Scale input features using z_score_scaling.
- Split data into training, validation, and testing sets.
- Create arrays X_train, y_train for training.
- Form arrays X_valid, y_valid for validation.
- Prepare arrays X_test, y_test for testing.
- Set split proportions: 60% training, 20% validation, 20% testing.

```
from sklearn.model_selection import train_test_split
def z_score_scaling(dataset):
    mean_vals = np.mean(dataset, axis=0)
    stdev_vals = np.std(dataset, axis=0)
    scaled_dataset = (dataset - mean_vals) / stdev_vals
    return scaled_dataset



modified_list=np.array(new_data)
X=modified_list[:,:-1]
Y=modified_list[:,-1]
X=z_score_scaling(X)
X=np.array(X)
X_train,X_test,y_train,y_test = train_test_split(X,Y, test_size=0.4, random_state=1)
X_valid,X_test,y_valid,y_test = train_test_split(X_test,y_test, test_size=0.5, random_state=1)
print("Train data:", len(X_train))
print("Test data:", len(X_valid))
print("Validation data:", len(X_test))
```

Step-03: Cost Computation for Linear Regression

- Define function compute_cost(X, y, w, b) to calculate the cost of linear regression.
- Compute predicted values f_wb_i using input features X, weight vector w, and bias b.
- Accumulate squared differences (f_wb_i - y[i])**2 for each data point.

- Normalize accumulated differences by dividing by twice the number of data points (2 * m).
- Return the calculated cost value.

```
b_init = 10
w_init = np.array([ .2, .2,50])
def compute_cost(X, y, w, b):
    m = X.shape[0]
    cost = 0.0
    for i in range(m):
        f_wb_i = np.dot(X[i], w) + b
        cost = cost + (f_wb_i - y[i])**2
    cost = cost / (2 * m)
    return cost
```

Step-04: Gradient Computation for Linear Regression

This section calculates the gradients necessary for updating model parameters in linear regression. It defines a function compute_gradient that takes input features X, target values y, and initial model parameters w and b. The gradients are computed based on the prediction errors and feature values, and these gradients are normalized by the number of data points. The resulting gradients are used for adjusting the model parameters to improve its fit to the data.

```
def compute_gradient(X, y, w, b):
    m,n = X.shape
    dj_dw = np.zeros((n,))
    dj_db = 0.

    for i in range(m):
        err = (np.dot(X[i], w) + b) - y[i]
        for j in range(n):
            dj_dw[j] = dj_dw[j] + err * X[i, j]
        dj_db = dj_db + err
    dj_dw = dj_dw / m
    dj_db = dj_db / m
 return dj_db, dj_dw
tmp_dj_db, tmp_dj_dw = compute_gradient(X_train, y_train, w_init, b_init)
```

Step-05: Gradient Descent for Model Training

This section contains a function called gradient_descent that trains a linear regression model using gradient descent. It updates model parameters over iterations to minimize loss.

```
import copy
import matplotlib.pyplot as plt
def gradient_descent(X, y, w_in, b_in, cost_function, gradient_function, alpha, num_iters):

    w = copy.deepcopy(w_in)
    b = b_in

    for i in range(num_iters):

        dj_db,dj_dw = gradient_function(X, y, w, b)
        w = w - alpha * dj_dw
        b = b - alpha * dj_db

        print(f"Training Loss: {compute_cost(X_train,y_train,w,b)}")
        print(f"Validation Loss: {compute_cost(X_valid,y_valid,w,b)}")
    return w, b
```

Step-06: Gradient Descent Training and Prediction

This section involves the process of training a linear regression model using gradient descent and making predictions based on the trained model.

```
initial_w = np.zeros_like(w_init)
print(initial_w)
initial_b = 1
iterations = 1500
alpha = 0.01
# run gradient descent
w_final, b_final = gradient_descent(X_train, y_train, initial_w, initial_b,compute_cost,
compute_gradient,alpha, iterations)
print(f"b,w found by gradient descent: {b_final:0.2f},{w_final} ")
```

```
def predict(X,w,b,test):
  n=X.shape[0]
  for i in range(n):
    y_pred.append(np.dot(X[i],w)+b)
    print(test[i],y_pred[i])

y_pred=[]
predict(X_test,w_final,b_final,y_test)
```

2.2.3 Performance Evaluation

The R-squared score, obtained through this code snippet as 59%, is a statistical measure that indicates the proportion of the variance in the dependent variable (target) that is explained by the independent variables (features) in the linear regression model. An R-squared value of 59% suggests that approximately 59% of the variability in the target variable can be explained by the independent variables in the model. This value gives an insight into how well the model fits the data: the closer the R-squared value is to 1, the better the model's fit.

2.3 K-Means Clustering

Clustering is the task of dividing a set of data points into groups, such that the data points in each group are similar to each other and dissimilar to the data points in other groups. The k-means algorithm is a clustering algorithm that works by iteratively assigning data points to one of k clusters, where k is a user-specified number.
The purpose of clustering is to find groups of data points that are similar to each other and dissimilar to data points in other groups. This can be useful for a variety of tasks, such as:

- Market segmentation: Clustering customers into groups based on their spending habits, demographics, or interests.
- Customer segmentation: Clustering customers into groups based on their purchase history, product preferences, or loyalty.
- Image segmentation: Clustering pixels in an image into groups based on their color or texture.
- Data compression: Clustering data points into groups and then representing each group with a single data point.

- Outlier detection: Identifying data points that are significantly different from the rest of the data.
- Dimensionality reduction: Reducing the number of dimensions in a dataset while preserving the important information.

2.3.1 Dataset

The dataset is a 2-dimensional dataset with 15 data points. Each data point is represented by a tuple of (X, Y) coordinates.

The X-coordinate represents the location of the data point on the X-axis. The Y-coordinate represents the location of the data point on the Y-axis.

The attributes of the dataset are the X-coordinate and the Y-coordinate. The relevance of these attributes is that they can be used to cluster the data points together.

1	No	X	Y
2	1	1	1
3	2	2	2
4	3	1.5	2.5
5	4	3	3
6	5	2.5	4
7			

Figure 4: Snapshot of data points dataset (K-means)

2.3.2 Implementation

Step-01: Importing the necessary libraries

This block imports the necessary libraries for clustering, namely NumPy, Matplotlib.

```
import pandas as pd
import random
import math
import matplotlib.pyplot as plt
```

Step-02: Read data

This sections read the data and stores the X and Y values of the data in X and Y.

```
data = pd.read_csv('/content/data.csv')
X = data['X'].values
```

```
Y = data['Y'].values
```

Step-03: Centroid Initialization for K-Means Clustering

This section defines a function named `initialize_centroids` that prepares initial centroids for K-Means clustering. It randomly selects data points as centroids based on the provided input features and target values. The function returns a list of initialized centroids that will serve as starting points for the K-Means clustering algorithm.

```
def initialize_centroids(X, Y, K):
    centroids = []
    for i in range(K):
        index = random.randint(0, len(X) - 1)
        centroids.append((X[index], Y[index]))
    return centroids
```

Step-04: Assigning Data Points to Clusters in K-Means

This part defines a function named assign_clusters that allocates data points to clusters in the K-Means algorithm. It performs the following steps:

- Takes input features X, target values Y, and a list of centroids.
- Initializes an empty list cluster to store data points grouped by clusters.
- For each data point (X[i], Y[i]):
- Computes distances to all centroids.
- Identifies the index of the nearest centroid.
- Appends the data point to the cluster associated with the nearest centroid.
- Returns a list of clusters, where each cluster contains data points assigned to a specific centroid.

```
def assign_clusters(X, Y, centroids):
    clusters = [[] for i in range(len(centroids))]
    for i in range(len(X)):
        point = (X[i], Y[i])
        distances = [math.sqrt((point[0] - centroid[0]) ** 2 + (point[1] - centroid[1]) ** 2)
                     for centroid in centroids]
        nearest_centroid = distances.index(min(distances))
        clusters[nearest_centroid].append(point)
    return clusters
```

Step-05: Updating Centroids in K-Means Clustering

This code section contains a function named update_centroids. Here's what it does:

- Takes a list of clusters as input, where each cluster contains data points.
- Initializes an empty list named centroids to hold updated centroid positions.
- For each cluster in the list:
- Calculates the mean of x-coordinates and y-coordinates of all points in the cluster.
- Constructs a new centroid point with the calculated mean coordinates.
- Appends the new centroid to the centroids list.
- Returns the list of updated centroids based on the mean values of the data points in each cluster.

```
def update_centroids(clusters):
    centroids = []
    for cluster in clusters:
        centroid_x = sum(point[0] for point in cluster) / len(cluster)
        centroid_y = sum(point[1] for point in cluster) / len(cluster)
        centroids.append((centroid_x, centroid_y))
    return centroids
```

Step-05: K-Means Clustering Initialization

- Calls k_means_clustering function with input features X, target values Y, and desired number of clusters K.
- Executes iterations until current centroids match previous centroids.
- Uses assign_clusters to allocate data points to clusters.
- Updates centroids by calling update_centroids.
- Visualizes data points in clusters using different colors.
- Marks cluster centroids with black 'x' markers.
- Presents the plot with labeled axes and title.
- Takes user input for the number of clusters K.
- Returns clusters and their respective centroids.

```
def k_means_clustering(X, Y, K):
    centroids = initialize_centroids(X, Y, K)
    prev_centroids = []
```

```
    while centroids != prev_centroids:
        clusters = assign_clusters(X, Y, centroids)
        prev_centroids = centroids
        centroids = update_centroids(clusters)
        colors = ['red',  'blue', 'pink','yellow',"silver","orange","green","violet","aqua","cyan"]
        plt.figure(figsize=(8, 6))

        counter = 0
        for cluster in clusters:
            for point in cluster:
                plt.scatter(point[0], point[1], c=colors[counter], marker='o')
            counter += 1

        for centroid in centroids:
            # print(centroid)
            plt.scatter(centroid[0], centroid[1], c='black', marker='x', s=100)
        plt.xlabel('X')
        plt.ylabel('Y')
        plt.title('K-means Clustering')
        plt.show()
    return clusters, centroids


K = int(input("Enter the value of K: "))
clusters, centroids = k_means_clustering(X, Y, K)
```

2.3.3 Performance Evaluation

The clusters are labeled as follows:

Cluster 1 (red): This cluster consists of data points in the upper-left corner of the graph.

Cluster 2 (green): This cluster consists of data points in the lower-left corner of the graph.

Cluster 3 (blue): This cluster consists of data points in the middle of the graph.

Cluster 4 (purple): This cluster consists of data points in the upper-right corner of the graph.

We can analyze and interpret the clusters formed as follows:
Each cluster consists of data points that are relatively close to each other and are well-separated from the other data points. This suggests that these data points may share some common characteristics. The data points in this cluster have the lowest values for both the X and Y attributes. This suggests that these data points may be the smallest data points in the dataset.

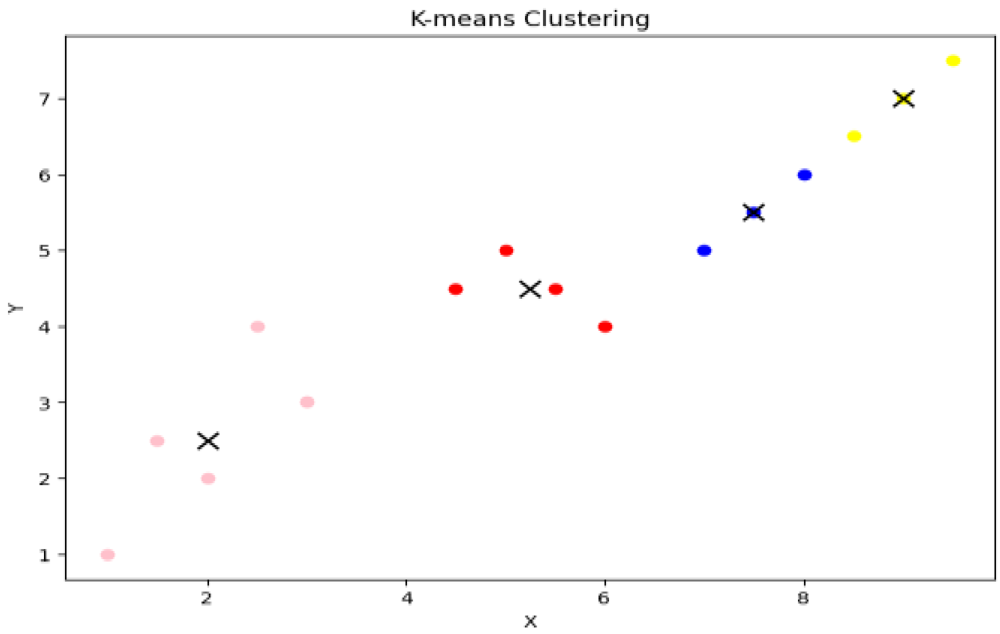

Figure 5: Final outcomes of Clusters for K=3

2.4 Decision Tree

Decision trees are vital tools in machine learning for classification and regression. They construct a tree-like model where each internal node assesses a feature based on a threshold. Mathematically, decisions are made by comparing feature values to these thresholds. The process recursively guides data to leaf nodes, which correspond to predictions.
For classification, nodes minimize impurity (e.g., Gini) by choosing thresholds that best separate classes. In regression, nodes minimize the variance of target values. These trees can become overly complex, so techniques like pruning and ensembles (e.g., Random Forests) help manage complexity and enhance predictive power.
In essence, decision trees mathematically partition data into branches, making them valuable for intuitive predictions.

2.4.1 Dataset

The dataset contains information about weather conditions and whether golf was played on certain days. It's structured as follows:

Features:

Day: The date of the recorded weather data.

Temperature: Categorizes the temperature as "hot," "mild," or "cool."

Outlook: Describes the outlook as "sunny," "overcast," or "rain."

Humidity: Indicates humidity as "high" or "normal."

Windy: Represents whether it's windy with "TRUE" or "FALSE."

Target Label:

Play Golf?: This is the target label, indicating whether golf was played on that particular day. It's labeled as "yes" if golf was played and "no" if not.

The dataset presents a set of weather conditions (temperature, outlook, humidity, windiness) and whether golf was played on those specific days. This kind of dataset is commonly used for classification tasks, where the goal is to predict categorical outcomes based on the given features.

Day	Temperatu	Outlook	Humidity	Windy	Play Golf?
5-Jul	hot	sunny	high	FALSE	no
6-Jul	hot	sunny	high	TRUE	no
7-Jul	hot	overcast	high	FALSE	yes
9-Jul	cool	rain	normal	FALSE	yes
10-Jul	cool	overcast	normal	TRUE	yes
12-Jul	mild	sunny	high	FALSE	no
14-Jul	cool	sunny	normal	FALSE	yes
15-Jul	mild	rain	normal	FALSE	yes
20-Jul	mild	sunny	normal	TRUE	yes

Figure 6: Snapshot of Weather Dataset

2.4.2 Implementation

Step-01: Load and prepocessing of data

- Read the CSV file using the "pd.read_csv" function and store it in a DataFrame called "df".
- Convert the 'Windy' column of the DataFrame to integer type using the "astype" method to represent boolean values as integers.
- Create a DataFrame "X" by dropping the 'Play Golf?' column from the original DataFrame "df".

- Create a Series "y" by selecting the 'Play Golf?' column from the original DataFrame "df"
- Utilize the "train_test_split" function to split "X" and "y" into training and testing sets.
- Specify a test size of 20% of the data using "test_size=0.2".
- Set the random seed to ensure reproducibility using "random_state=42".
- Store the resulting training and testing sets as "X_train", "X_test", "y_train", and "y_test".

```
df = pd.read_csv('data.csv')
df['Windy'] = df['Windy'].astype(int)
X = df.drop(columns=['Play Golf?'])
y = df['Play Golf?']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step-02: Calculating Information gain, Entropy and building tree

- Defines a class named "DecisionTree" responsible for building a decision tree model.
- Initializes an empty dictionary named "tree" to store the decision tree structure.
- Defines a method named "entropy(target_col)" that calculates the entropy of a target column.
- Calculates unique elements and their counts in the target column.
- Computes the entropy using the formula for information entropy.
- Defines a method named "information_gain(data, feature, target)" to compute the information gain of a feature in the dataset.
- Defines a method named "build_tree(data, features, target)" to recursively build the decision tree.
- If all instances in the data have the same target value.
- If no features remain to split on.
- Selects the feature with the highest information gain as the decision node.

```
class DecisionTree:
    def __init__(self):
```

```
        self.tree = {}

    def entropy(self, target_col):
        elements, counts = np.unique(target_col, return_counts=True)
        entropy = np.sum([(-counts[i] / np.sum(counts)) * np.log2(counts[i] / np.sum(counts)) for
i in range(len(elements))])
        return entropy

    def information_gain(self, data, feature, target):
        total_entropy = self.entropy(data[target])
        vals, counts = np.unique(data[feature], return_counts=True)
        weighted_entropy = np.sum([(counts[i] / np.sum(counts)) *
self.entropy(data[data[feature] == vals[i]][target]) for i in range(len(vals))])
        information_gain = total_entropy - weighted_entropy
        return information_gain

    def build_tree(self, data, features, target):
        if len(pd.unique(data[target])) == 1:
            return pd.unique(data[target])[0]
              if len(features) == 0:
            return pd.unique(data[target])[pd.argmax(pd.unique(data[target],
return_counts=True)[1])]

        best_feature = max(features, key=lambda x: self.information_gain(data, x, target))
        tree = {best_feature: {}}

        features = [f for f in features if f != best_feature]

        for val in pd.unique(data[best_feature]):
            sub_data = data.where(data[best_feature] == val).dropna()
            subtree = self.build_tree(sub_data, features, target)
            tree[best_feature][val] = subtree

        return tree
```

Step-03: Creating Decision Tree and Prediction Setup

- Creates a "DecisionTree" instance named "dt".
- Defines features as column names except the first and last from the DataFrame.
- Sets the target variable as 'Play Golf?'.
- Constructs a decision tree using "build_tree" method with "df", features, and target.
- Prepares a sample input dictionary for prediction.

```
dt = DecisionTree()
features = df.columns[1:-1]
target = 'Play Golf?'
dt.tree = dt.build_tree(df, features, target)
sample_input = {'Temperature': 'hot', 'Outlook': 'sunny', 'Humidity': 'high', 'Windy': 0}
```

Step-04: Decision Tree Prediction Function

Defines a function named "predict" to make predictions using a given decision tree and input data.

- For each feature key in the input dictionary:
- Checks if the key exists in the decision tree's keys.
- Attempts to access the subtree corresponding to the feature's value.
- If the subtree is a dictionary, recursively calls the "predict" function on the subtree.
- If the subtree is not a dictionary, returns the predicted value.
- Handles exceptions and returns an error message if prediction cannot be made.

```
def predict(input, tree):
    for key in input.keys():
        if key in tree.keys():
            try:
                subtree = tree[key][input[key]]
                if isinstance(subtree, dict):
                    return predict(input, subtree)
                else:
                    return subtree
            except:
                return "Unable to make a prediction."
```

Step-05: Calculate Model Accuracy

- Defines a function named "calculate_accuracy" to compute the accuracy of a decision tree model.
- For each data point in the testing set:
- Uses the "predict" function to generate predictions based on the decision tree and input data.
- Compares the predicted values to the actual target values in the testing set.
- Computes accuracy using the "accuracy_score" function from sklearn.
- Returns the calculated accuracy value.

```
from sklearn.metrics import accuracy_score


def calculate_accuracy(X_test, y_test, tree):
    predictions = [predict(X_test.iloc[i].to_dict(), tree) for i in range(len(X_test))]
    accuracy = accuracy_score(y_test, predictions)
    return accuracy
```

Sample Input & output:

Sample input: {'Temperature': 'hot', 'Outlook': 'sunny', 'Humidity': 'high', 'Windy': 0}

Prediction: no

Decision Tree Visualization:

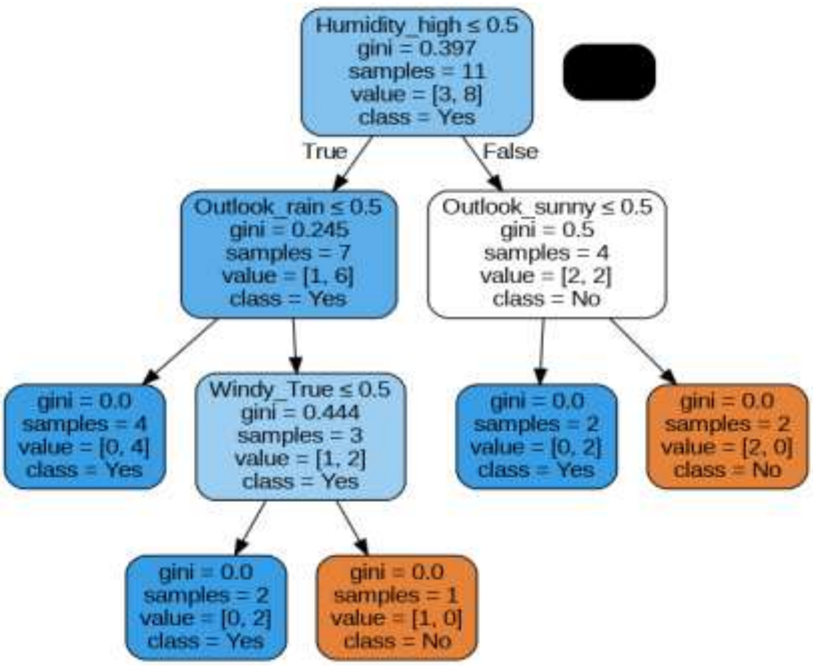

Figure 7: Decision Tree

2.4.3 Performance Evaluation

Interpretability and Insights from Decision Tree:

- **Clear Pathways:** Tree structure shows decisions based on features, offering straightforward paths for predictions.
- **Key Features:** Top nodes highlight influential features, aiding predictive understanding.
- **Feature Interactions:** Tree reveals how features work together, indicating joint effects on outcomes.
- **Visual Understanding:** Visualization simplifies comprehension for non-experts.
- **Rule Extraction:** Paths translate to understandable rules for practical decision-making.
- **Segmentation:** Data split into subsets enables targeted analysis.
- **Overfitting Risk:** Deep trees may overfit; pruning mitigates this.
- **Actionable Knowledge:** Interpretation empowers insights for various applications.

2.5 Artificial Neural Network (ANN)

Artificial neural networks are computing systems with an intricate, brain-inspired design. They consist of basic units called nodes that are analogous to the neurons in a biological brain. Each node performs a simple computation on its input and passes the output to further nodes through connections. These connections modulate the signal strength via adjustable weights, similar to synapses in the human brain.

By structuring nodes in successive layers and assigning proper connection weights, the neural network can model the flow of data from input to output. The input layer ingests raw data. Then, one or more hidden layers incrementally process parts of the data and extract meaningful patterns within it. Finally, the output layer combines these patterns and interpretations to generate predictions, classifications or detections as required by the problem.

A key ingredient that enables learning in neural networks is backpropagation. This refers to the backward flow of errors from the output layer through the network. Gradients are used to tune the weights across connections in a way that minimizes the prediction error at the output. The network learns to map arbitrary inputs to the desired outputs by iterating through training data and minimizing the loss via backpropagation.

Deep learning leverages neural networks with multiple hidden layers, hence the name “deep”. By composing many non-linear operations, deep networks can represent highly complex relationships and hierarchies within the data. This allows them to learn intricate concepts and generalize well to unseen data. Deep learning models now achieve state-of-the-art results on various machine learning tasks involving images, text, speech, and more. The ability to automatically learn useful features makes deep learning networks versatile and capable for real-world applications.

2.5.1 Dataset

The synthetic dataset was generated using the generate_synthetic_data_with_relation() method which creates randomized sample data with a predefined relationship between features and labels.

Features:

- The features matrix has shape (num_samples, num_features)
- It is filled with normally distributed random values using np.random.randn()
- The number of samples and features can be controlled via input arguments

Labels:

- The labels are derived by multiplying features with a random weight matrix
- A softmax function is applied to get probabilities over classes
- The class with maximum probability is assigned as the label

This produces a randomized synthetic dataset where the labels are statistically related to the feature values as per the choice of weight matrix. By training on this data, the model can learn this encoded relationship between features and labels.

The key aspects are - randomized features, labels based on a preset relationship, and the ability to configure the size via arguments. This procedurally generated data is a useful approach for prototyping and testing neural network models.

2.5.2 Implementation

Step-01: Sigmoid Activation Functions

sigmoid(x): Calculates the sigmoid function output, which is 1 / (1 + exp(-x)).

sigmoid_derivative(x): Computes the derivative of the sigmoid function, x * (1 - x).

```
import numpy as np
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)
```

Step-02: Initialization of Neural Network Parameters

- Initializes neural network parameters
- Generates random weights from a normal distribution for input-to-hidden connections.
- Sets biases for the input-to-hidden connections to zeros.
- Generates random weights for hidden-to-output connections.
- Sets biases for the hidden-to-output connections to zeros.
- Returns the initialized weights and biases for both layers.

```
def initialize_parameters(input_size, hidden_size, output_size):
    np.random.seed(42)
    weights_input_hidden = np.random.randn(input_size, hidden_size)
    biases_input_hidden = np.zeros((1, hidden_size))
    weights_hidden_output = np.random.randn(hidden_size, output_size)
    biases_hidden_output = np.zeros((1, output_size))
    return weights_input_hidden, biases_input_hidden, weights_hidden_output,
biases_hidden_output
```

Step-03: Forward Propagation in Neural Network

- Calculates forward propagation in a neural network
- Computes hidden layer output using sigmoid activation and input-to-hidden weights
- Calculates output layer output using sigmoid activation and hidden-to-output weights
- Returns both the hidden layer and output layer outputs

```
def forward_propagation(X, weights_input_hidden, biases_input_hidden,
weights_hidden_output, biases_hidden_output):
    hidden_layer_output = sigmoid(np.dot(X, weights_input_hidden) + biases_input_hidden)
```

```
    output_layer_output = sigmoid(np.dot(hidden_layer_output, weights_hidden_output) + biases_hidden_output)
    return hidden_layer_output, output_layer_output
```

Step-04: Backpropagation in Neural Network

- Performs backpropagation to calculate error gradients
- Computes output layer error as the difference between actual and predicted values.
- Calculates output delta using output error and sigmoid derivative of output layer output.
- Determines hidden layer error by multiplying output delta with transposed hidden-to-output weights.
- Computes hidden delta using hidden layer error and sigmoid derivative of hidden layer output.
- Returns both output delta and hidden delta for weight updates

```
def backpropagation(X, y, hidden_layer_output, output_layer_output, weights_hidden_output):
    output_error = y - output_layer_output
    output_delta = output_error * sigmoid_derivative(output_layer_output)
    hidden_error = np.dot(output_delta, weights_hidden_output.T)
    hidden_delta = hidden_error * sigmoid_derivative(hidden_layer_output)
    return output_delta, hidden_delta
```

Step-05: Training Neural Network

- Sets up input, output, and hidden layer sizes.
- Initializes parameters using initialize_parameters.
- For each epoch:
- Performs forward propagation using forward_propagation.
- Executes backpropagation to obtain deltas using backpropagation.
- Updates hidden-to-output weights and biases based on calculated deltas.
- Updates input-to-hidden weights and biases based on calculated deltas.
- Returns updated weights and biases for both layers after training.

```
def train(X, y, learning_rate, num_epochs):
    input_size = X.shape[1]
    output_size = y.shape[1]
    hidden_size = 12
    weights_input_hidden, biases_input_hidden, weights_hidden_output, biases_hidden_output = initialize_parameters(input_size, hidden_size, output_size)
    for epoch in range(num_epochs):
        hidden_layer_output, output_layer_output = forward_propagation(X, weights_input_hidden, biases_input_hidden, weights_hidden_output, biases_hidden_output)
        output_delta, hidden_delta = backpropagation(X, y, hidden_layer_output, output_layer_output, weights_hidden_output)
        weights_hidden_output += learning_rate * np.dot(hidden_layer_output.T, output_delta)
        biases_hidden_output += learning_rate * np.sum(output_delta, axis=0, keepdims=True)
        weights_input_hidden += learning_rate * np.dot(X.T, hidden_delta)
        biases_input_hidden += learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)
    return weights_input_hidden, biases_input_hidden, weights_hidden_output, biases_hidden_output
```

Step-06: Neural Network Prediction

- Makes predictions using trained neural network
- Computes hidden layer input using input-to-hidden weights and biases.
- Obtains hidden layer output via sigmoid activation.
- Calculates output layer input using hidden-to-output weights and biases.
- Generates output layer output using the softmax activation function.
- Returns the predicted output layer probabilities for each class.

```
def predict(X, weights_input_hidden, biases_input_hidden, weights_hidden_output, biases_hidden_output):
    hidden_layer_input = np.dot(X, weights_input_hidden) + biases_input_hidden
    hidden_layer_output = sigmoid(hidden_layer_input)
    output_layer_input = np.dot(hidden_layer_output, weights_hidden_output) + biases_hidden_output
    output_layer_output = softmax(output_layer_input)
```

```
    return output_layer_output
```

Step-07: Generating Dataset

- Generates synthetic data with known relationships
- Sets random seed for reproducibility.
- Generates feature matrix with dimensions (num_samples, num_features).
- Generates weight matrix with dimensions (num_features, num_classes).
- Computes logits using dot product of features and weights.
- Computes softmax probabilities from logits for each sample.
- Assigns labels as the class with highest probability.
- Returns synthesized features and corresponding labels for analysis or modeling.

```
import numpy as np


def generate_synthetic_data_with_relation(num_samples, num_features, num_classes):
    np.random.seed(42)
    features = np.random.randn(num_samples, num_features)
    weights = np.random.randn(num_features, num_classes)
    logits = np.dot(features, weights)
    softmax_probs = np.exp(logits) / np.sum(np.exp(logits), axis=1, keepdims=True)
    labels = np.argmax(softmax_probs, axis=1)
    return features, labels
```

Step-08: Neural Network Evaluation and Analysis

Generating Synthetic Data:

Generates synthetic data with specified sample, feature, and class counts using the predefined relationship.

One-Hot Encoding:

Applies one-hot encoding to convert original target labels into binary-encoded vectors.

Data Splitting:

Divides the dataset into training and testing sets for model assessment.

Neural Network Training:

- Trains the neural network using provided training data

- Defines learning rate and number of epochs.
- Initializes network parameters.
- Utilizes the train function for weight updates.

Predictions and Accuracy:

- Predicts outcomes using the trained neural network on the test set
- Converts raw predictions to class labels.
- Calculates accuracy as the proportion of correct predictions.

Confusion Matrix:

- Computes the confusion matrix to analyze model performance:
- Displays actual class labels against predicted labels.

Generates a detailed classification report:

Includes precision, recall, F1-score, and support for each class.

Provides overall metrics for the model's performance.

The provided code comprehensively evaluates the neural network's effectiveness on the test data and presents essential performance metrics.

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.metrics import confusion_matrix, classification_report



num_samples = 1000
num_features = 4
num_classes = 3
X, y = generate_synthetic_data_with_relation(num_samples=num_samples,
num_features=num_features, num_classes=num_classes)

# One-hot encode the target labels
encoder = OneHotEncoder(sparse=False)
y_encoded = encoder.fit_transform(y.reshape(-1, 1))

# Split the data into training and testing sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.2, random_state=42)

# Train the neural network
learning_rate = 0.1
num_epochs = 10000
weights_input_hidden, biases_input_hidden, weights_hidden_output, biases_hidden_output = train(X_train, y_train, learning_rate, num_epochs)

# Make predictions on the test set
predictions = predict(X_test, weights_input_hidden, biases_input_hidden, weights_hidden_output, biases_hidden_output)

# Convert predictions to class labels based on maximum value
predicted_labels = np.argmax(predictions, axis=1)

# Calculate accuracy
accuracy = np.mean(predicted_labels == np.argmax(y_test, axis=1))
print(f"Accuracy: {accuracy:.4f}")

# Calculate confusion matrix
conf_matrix = confusion_matrix(np.argmax(y_test, axis=1), predicted_labels)
print("Confusion Matrix:")
print(conf_matrix)
class_labels = [f"Class {i}" for i in range(num_classes)]
classification_rep = classification_report(np.argmax(y_test, axis=1), predicted_labels, target_names=class_labels)
```

2.5.3 Performance Evaluation

The following section is for the performance evaluation.

```
import matplotlib.pyplot as plt
import seaborn as sns
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=class_labels,
yticklabels=class_labels)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

- Generates a heatmap visualization of the confusion matrix.
- Annotations indicate values within heatmap cells.
- Applies "Blues" color map for intensity.
- Labels x-axis as "Predicted" and y-axis as "True."
- Sets title as "Confusion Matrix."
- Displays the heatmap visualization

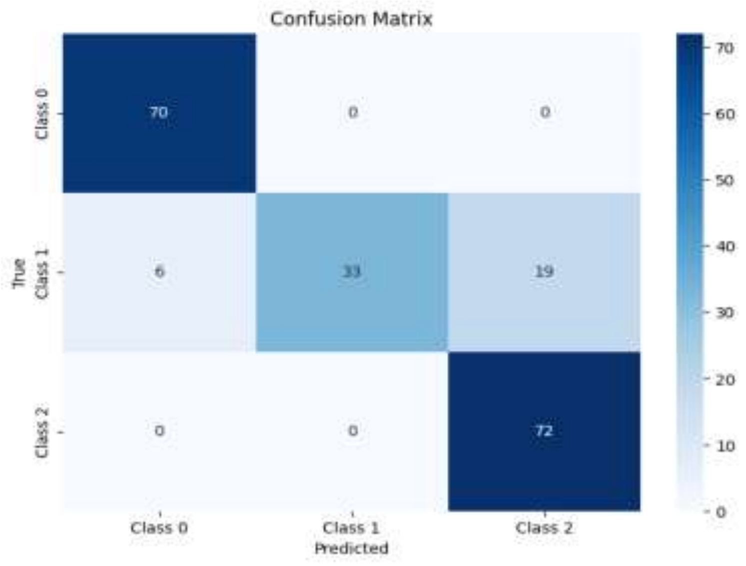

Figure 8: Confusion Matrix (Backpropagation)

3 Discussion

Strengths and Weaknesses:

- Apriori - Useful for market basket analysis but struggles with large datasets.
- Multivariable Linear Regression - Handles multiple predictor variables but assumes linear relationships.
- K-Means - Easy to implement but requires specifying k and doesn't work well on non-globular clusters.
- Decision Tree - Interpretable and handles various data types but prone to overfitting.
- Neural Network - Can model complex patterns but acts as black boxes lacking interpretability.

Common Challenges and Solutions:

Major challenges faced were tuning hyperparameters, debugging implementations, and interpreting results. Methods like cross-validation, regularization, and visualizations helped address these.

Limitations and Potential Improvements:

- Apriori's iterative approach can be optimized using FP-growth.
- Regression performance may improve with nonlinear models like SVMs.
- Neural nets could benefit from enhancements like batch normalization and dropout.

Real-World Relevance:

These algorithms enable productive applications like personalized recommendations, predictions, data mining, and pattern recognition. However, care must be taken to critically evaluate performance on real datasets and intended tasks before deployment.

Overall, the hands-on implementation experience will be invaluable in assessing the practical utility of algorithms for various analytical problems.

4 Conclusion

In summary, this project enabled students to gain invaluable hands-on experience in implementing key machine learning algorithms end-to-end. They demonstrated proficiency in computational thinking, Python programming, data preparation, model optimization, evaluation, and result interpretation.

The practical implementation enhanced conceptual understanding and developed vital analytical skills like debugging, hyperparameter tuning, and critical analysis. Students gained intuition by addressing real-world challenges and nuances first-hand.

This supplement to theoretical knowledge was an important milestone in empowering students to become adept practitioners. They are now better equipped to evaluate algorithm suitability for problems and have a firm basis for tackling machine learning tasks.

The hands-on experience strengthened problem-solving abilities and will be tremendously beneficial as students embark on applying these algorithms to build impactful solutions

5 References

1. Kaggle datasets. https://www.kaggle.com/datasets
2. Scikit-learn: Machine Learning in Python. https://scikit-learn.org/
3. Deep Learning Specialization on Coursera by Andrew Ng
4. Documentation (Numpy, pandas, Matplotlib)
5. Article on Neural Network (Neural networks and back-propagation explained in a simple way | by Assaad MOAWAD | DataThings | Medium)
6. Machine learning hands on tutorial (Machine Learning Tutorial | Tutorialspoint)

6 Appendices

The implementation of the five maching learning algorthims google colab links are attached below:

Apriori:

https://colab.research.google.com/drive/1MRlf8vUa4MSQgISx0y71IcXXbR3sDVGd?usp=sharing

Multivariant Linear Regression:

https://colab.research.google.com/drive/15wcNhS0XaWxlWAUy6_JGM0u9JDnAaDeC?usp=sharing

K-means Clustering:

https://colab.research.google.com/drive/1hrFcLOOGqcbDpA4ql_-depEwv9lwD8mI?usp=sharing

Decision Tree:

https://colab.research.google.com/drive/1-kZ_XbRgssItjZ7323el_AFXB6SpO8-Y?usp=sharing

Back Propagation:

https://colab.research.google.com/drive/1YYK78iJe5_qAxkmFkOjo45fSwidCZ830?usp=sharing

GitHub: rifaturrana/ML_Scratch_Codes_1804098 (github.com)