Bachelor of Science in Computer Science & Engineering

Final Report on Machine Learning Algorithms Implementation

by

Salman Farsi

ID: 1804102

Department of Computer Science & Engineering

Chittagong University of Engineering & Technology (CUET)

Chattogram-4349, Bangladesh.

26th August, 2023

Chittagong University of Engineering & Technology (CUET)
Department of Computer Science & Engineering
Chattogram-4349, Bangladesh.

Lab Report

Machine Learning Sessional (CSE - 464)

Student Name : Salman Farsi Session : 2021-2022
ID : 1804102

Submitted To : Md. Rashadur Rahman
Lecturer
Department of Computer Science & Engineering

Hasan Murad
Lecturer
Department of Computer Science & Engineering

Department : Computer Science & Engineering
Program : B.Sc. Engineering

Tentative Title : Final Report on Machine Learning Algorithms Implementation

Table of Contents

List of Figures

List of Code Snippets

Abstract: Through the practical implementation of a diverse array of machine learning algorithms, a wealth of practical experience has been gained from this study. A total of five algorithms namely the Apriori algorithm, Decision Tree, Multivariable Linear Regression, K-Means Clustering, and Artificial Neural Network (ANN) were implemented in this study. While implementing these algorithms a greater understanding of the inner workings of all these algorithms is understood properly. Besides, several error metrics, error calculation strategies, and how to compare algorithms by several measures, were also great pieces of learning as part of this study. It was found that some algorithms do better for classification scenarios, while others do better at certain situations involving regression. Also, some algorithms handle the continuous value efficiently while others do not or depend on more complex data preprocessing techniques. Handling missing values, data visualization techniques, feature selection, feature scaling, etc. all were part of the key learnings from this study.

1 introduction

Machine learning, a subset of AI, enables computers to learn from data and make predictions without explicit programming, driving its popularity across industries. Its versatility in supervised, unsupervised, and reinforcement learning has revolutionized healthcare diagnostics, fraud detection in finance, recommendation systems in retail, predictive maintenance in manufacturing, and NLP applications like language translation [1]. Recent advancements in deep learning, a neural network-based approach, have further boosted its capabilities. As we stand on the cusp of a data-driven era, machine learning's role in automating processes and extracting insights positions it as a transformative technology powering the digital transformation of diverse sectors.

In this study, it focuses on the implementation and understanding of the five basic machine learning algorithms namely the Apriori algorithm, Decision Tree, Multivariable Linear Regression, K-means clustering, and Artificial Neural Network (ANN).

- **Apriori:** The Apriori algorithm focuses on association rule mining, which

is key in detecting frequent item sets within datasets. Its significance lies in uncovering concealed patterns and connections among items.

- **Multivariable Linear Regression:** The method of Multivariable Linear Regression is employed to model the connection between multiple input variables and a continuous target variable. The principles of linear regression are employed to make predictions based on provided features.
- **K-Means Clustering:** Similar data points are grouped together using the clustering algorithm known as K-Means, which is extensively employed to segment the data into separate clusters according to their similarities.
- **Decision Trees:** Employed for both classification and regression tasks, Decision Trees are a versatile algorithm that offers interpretable insights and aids in decision-making by utilizing input features.
- **Artificial Neural Networks:** Derived from the neural structures of the human brain, an Artificial Neural Network (ANN) constitutes a deep learning methodology. Through the utilization of an ANN, a journey is undertaken into the realm of intricate pattern recognition and the exploration of nonlinear relationships.

The field of machine learning algorithms has been thoroughly examined in this study, aiming to present an adept understanding of these techniques and their proficient utilization on real-world datasets. This experience serves to enhance both theoretical understanding and the practical skills necessary to tackle intricate challenges in the realm of machine learning. As a beginner to machine learning the newcomer will be encouraged to practice implementing machine learning algorithms from scratch and this will allow them to deep dive into the world of artificial intelligence in the near future.

2 Algorithm Descriptions and Implementations

2.1 Apriori Algorithm

Association rule mining involves uncovering significant relationships between items in datasets, with applications ranging from market basket analysis to recommendation systems. Within this field, the Apriori algorithm stands as a pivotal tool, concentrating on the detection of frequent itemsets. These itemsets denote clusters of items that tend to co-occur. The algorithm achieves this by iteratively generating and refining candidate itemsets [2]. This efficient approach avoids exploring unlikely combinations and results in a concise set of meaningful associations. These discovered associations, quantified through measures like confidence, provide valuable insights into consumer behavior and inform decision-making in various domains. However, there are several concepts associated with the apriori algorithm.

- **Generating Frequent 1-Itemsets:** The algorithm initiates by scanning the dataset to determine the support of each individual item. Items with support above the defined threshold are classified as frequent 1-itemsets initially.
- **Generating Candidate Itemsets:** Candidate sets of items with a length of k+1 are created from the pool of the frequent k-itemsets. This is accomplished through a self-join operation, followed by the removal of candidate itemsets containing subsets that are not frequent.
- **Pruning:** The algorithm strategically prunes candidate itemsets featuring subsets that are not frequent. This step significantly reduces the number of candidate itemsets requiring further consideration.
- **Scanning and Counting:** After pruning, the algorithm scans the dataset once more to count the support for each remaining candidate itemset. Those

candidate itemsets surpassing the predefined threshold are recognized as frequent and contribute to generating the next round of candidate itemsets.

- **Finding Interesting Association Rules:** For each frequent itemset, the algorithm generates all possible non-empty subsets. After calculating the confidence for each, if it is found that any of the rules formed is below the minimum confidence threshold then that rule can't be considered as interesting.

2.1.1 Dataset

```
df = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/store_data.csv', header = None)
df
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	shrimp	almonds	avocado	vegetables mix	green grapes	whole weat flour	yams	cottage cheese	energy drink	tomato juice	low fat yogurt	green tea	honey	salad	mineral water	salmon	antioxydant juice	frozen smoothie	spinach
1	burgers	meatballs	eggs	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	chutney	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	turkey	avocado	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	mineral water	milk	energy bar	whole wheat rice	green tea	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
7496	butter	light mayo	fresh bread	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
7497	burgers	frozen vegetables	eggs	french fries	magazines	green tea	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
7498	chicken	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
7499	escalope	green tea	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
7500	eggs	frozen smoothie	yogurt cake	low fat yogurt	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Figure 2.1: Transaction Dataset for Apriori algorithm

The dataset utilized for implementation is a collection of transaction records that capture the purchasing history of customers as shown in figure 2.1. In this dataset, each transaction mirrors a customer's shopping basket, detailing whether items are present or absent in the transaction. Structured in a tabular arrangement, the dataset features rows denoting transactions and columns representing individual items. The dataset's core attribute is the binary indication of item presence within each transaction. Every column corresponds to a distinct item, and each row corresponds to an individual customer transaction.

The dataset comprises a total of 7500 rows and 19 columns. This signifies that the dataset accommodates scenarios where a customer might have purchased up to 19 items within a single transaction. It's important to note that there are instances where the number of items purchased in a single transaction may be

fewer than 19. In such cases, the corresponding columns are populated with "NaN" to signify the absence of an item in that transaction.

2.1.2 Implementation

Data Preprocessing: Data preprocessing involves handling missing values in the DataFrame by substituting them with the string "Empty". Subsequently, the processed data is transformed into a list of lists denoted as 'rows_as_list'. Each inner list within this structure relates to a row within the initial data frame. The code prepares the data for analysis by removing 'Empty' values from each transaction and storing the cleaned data in the 'data' list of lists. Each inner list contains the row number and the cleaned transaction.

```
[ ] df = df.fillna("Empty")
```

```
rows_as_list = df.values.tolist()
rows_as_list
```

```
[['shrimp',
  'almonds',
  'avocado',
  'vegetables mix',
  'green grapes',
  'whole weat flour',
  'yams',
  'cottage cheese',
  'energy drink',
  'tomato juice',
  'low fat yogurt',
  'green tea',
  'honey',
  'salad',
  'mineral water',
  'salmon',
  'antioxydant juice',
  'frozen smoothie',
  'spinach',
  'olive oil'],
 ['burgers',
  'meatballs',
  'eggs',
```

Figure 2.2: Preprocessing for NaN values

```python
data = []
for i in range(len(rows_as_list)) :
  row_num = i
  curr = []
  for j in rows_as_list[i] :
    if j != 'Empty' :
      curr.append(j)
  data.append([row_num, curr])
data
```

```
   'ground beef',
   'chocolate',
   'soup',
   'almonds',
   'eggs',
   'hot dogs',
   'cottage cheese']],
 [103, ['ham', 'spaghetti', 'chocolate', 'eggs']],
 [104, ['ground beef', 'energy bar', 'pet food', 'carrots', 'protein bar']],
 [105,
  ['ground beef',
   'tomato sauce',
   'spaghetti',
   'mineral water',
   'almonds',
   'eggs']],
```

Figure 2.3: Removing Empty Values and Stored in 'Data' list

Minimum Support and Confidence as User input: The user is prompted to input the minimum support and confidence values as percentages. These values are then converted into appropriate thresholds for further calculations.

```python
support = float(input("Enter the Minimum Support Value in(%): "))
support = support / 100

conf = float(input("Enter the Minimum Confidence Value in(%): "))
support_count = int(support*len(data))

print(support_count)
```

```
Enter the Minimum Support Value in(%): 2
Enter the Minimum Confidence Value in(%): 30
150
```

Figure 2.4: Taking User Input

Initialization of Candidate Set: The 'initialize' function generates a list of unique items present in the dataset and initializes the candidate item set by counting the occurrences of each item in the dataset. The candidate set is stored as a Counter object. The 'PrintAns' function is used to print the contents of the candidate set.

```
[ ] def PrintAns(C_or_L, K, type) :
      print(type + str(K) + ":")
      for item_set in C_or_L:
        cnt = C_or_L[item_set]
        if K == 1 and type == 'C':
          print(str([item_set]) + ": " + str(cnt))
        else:
          print(str(list(item_set)) + ": " + str(cnt))
      print()
```

Figure 2.5: A Function for printing Answer

```
def initialize(data) :
  unique = []
  for lst in data:
    for items in lst[1]:
      if(items not in unique):
        unique.append(items)
  unique = sorted(unique)
  print(unique)

  Candidate_Set = Counter()

  for items in unique:
    for transaction in data:
      if(items in transaction[1]):
        Candidate_Set[items]+=1

  PrintAns(Candidate_Set, 1, "C")

  frequent_item_set = Counter()

  for item_set in Candidate_Set:
    if(Candidate_Set[item_set] >= support_count):
      frequent_item_set[frozenset([item_set])] += Candidate_Set[item_set]
  return frequent_item_set
```

Figure 2.6: Candidate Set and Frequent Item Set Generator

Apriori Algorithm Implementation: The 'Apriori_Scratch' function implements the Apriori algorithm. It starts by initializing the candidate set and generating frequent 1-itemsets. The algorithm then iterates through multiple counts to generate candidate itemsets and prune them based on the support threshold. Frequent itemsets are generated and printed at each iteration.

2.1.3 Performance Evaluation

Association Rule Generation: The provided code produces association rules using the frequent itemsets derived from the Apriori algorithm. It iterates through the frequent itemsets, calculates confidence scores for potential association rules,

and prints the association rules along with their confidence scores.

```
This is the Final(frequent set)
L3:
['chocolate', 'eggs']: 249
['soup', 'mineral water']: 173
['mineral water', 'spaghetti']: 448
['milk', 'eggs']: 231
['chocolate', 'green tea']: 176
['milk', 'frozen vegetables']: 177
['chocolate', 'milk']: 241
['ground beef', 'spaghetti']: 294
['mineral water', 'olive oil']: 207
['spaghetti', 'burgers']: 161
['whole wheat rice', 'mineral water']: 151
['chocolate', 'mineral water']: 395
['ground beef', 'eggs']: 150
['mineral water', 'eggs']: 382
['green tea', 'french fries']: 214
['milk', 'mineral water']: 360
['mineral water', 'shrimp']: 177
['tomatoes', 'spaghetti']: 157
['ground beef', 'milk']: 165
['green tea', 'spaghetti']: 199
['green tea', 'eggs']: 191
['french fries', 'spaghetti']: 207
['mineral water', 'burgers']: 183
['spaghetti', 'olive oil']: 172
['ground beef', 'mineral water']: 307
['mineral water', 'tomatoes']: 183
['spaghetti', 'shrimp']: 159
```

Figure 2.7: Final Frequent Itemset

```
chocolate -----> eggs, eggs : 20.260374288039056
eggs -----> chocolate, chocolate : 18.47181008902077
eggs -----> chocolate, chocolate : 18.47181008902077
chocolate -----> eggs, eggs : 20.260374288039056

soup -----> mineral water, mineral water : 45.64643799472295
mineral water -----> soup, soup : 9.675615212527964
mineral water -----> soup, soup : 9.675615212527964
soup -----> mineral water, mineral water : 45.64643799472295

mineral water -----> spaghetti, spaghetti : 25.05592841163311
spaghetti -----> mineral water, mineral water : 34.30321592649311
spaghetti -----> mineral water, mineral water : 34.30321592649311
mineral water -----> spaghetti, spaghetti : 25.05592841163311

milk -----> eggs, eggs : 23.765432098765434
eggs -----> milk, milk : 17.136498516320476
eggs -----> milk, milk : 17.136498516320476
milk -----> eggs, eggs : 23.765432098765434

chocolate -----> green tea, green tea : 14.320585842148088
green tea -----> chocolate, chocolate : 17.759838546922303
green tea -----> chocolate, chocolate : 17.759838546922303
chocolate -----> green tea, green tea : 14.320585842148088
```

Figure 2.8: Association Rules With Confidence

Sorting and Displaying Association Rules: The user is asked to input a value denoted as 'N', which determines the quantity of initial association rules showcasing the greatest confidence scores. Subsequently, the code arranges the generated association rules in a descending order based on their confidence scores. The resulting display showcases the initial 'N' association rules with the highest confidence scores.

```
[ ] N=int(input("Enter a value N to find first N number of Association rules of Highest Confidence Value in(%) : "))
    from itertools import islice
    sorted_map = dict(sorted(mapping.items(), key=lambda x: x[1], reverse=True))

    first_N_values = dict(islice(sorted_map.items(), N))

    for key, value in first_N_values.items():
        formatted_value = "{:.2f}".format(value)
        print(f"Association Rule: [{key}], Confidence Score : {formatted_value}"+"%")

    Enter a value N to find first N number of Association rules of Highest Confidence Value in(%) : 5
    Association Rule: [soup -> mineral water], Confidence Score : 45.65%
    Association Rule: [olive oil -> mineral water], Confidence Score : 41.90%
    Association Rule: [ground beef -> mineral water], Confidence Score : 41.66%
    Association Rule: [ground beef -> spaghetti], Confidence Score : 39.89%
    Association Rule: [cooking oil -> mineral water], Confidence Score : 39.43%
```

Figure 2.9: Top N association rules with the highest confidence

2.2 Multivariable Linear Regression

Multivariable Linear Regression [3], also referred to as Multiple Linear Regression, is a statistical technique employed in both machine learning and statistics to facilitate establishing a relationship between a dependent variable and multiple independent (input) variables. This approach builds upon the fundamentals of simple linear regression, which is employed to model the relationship between a dependent variable and a single independent variable. The objective of Multivariable Linear Regression is to derive a linear equation that adeptly captures the impact of multiple independent variables on a dependent variable. This equation facilitates predictions of the dependent variable's value by considering the given values of the independent variables.

$$y = b_0 + b_1x_1 + b_2x_2 + \ldots + b_nx_n$$

Where:

y represents the dependent variable (the predicted outcome).

b_i is the y-intercept or bias term.

x_i is the independent variables.

2.2.1 Dataset

The dataset in figure 2.10 used for linear regression contains information about apartments for rent in Bangladesh [4]. It includes features such as the number of bedrooms, and bathrooms, area in square feet, address, type of property, purpose (rent), floor plan, URL, last updated date, and price. The dataset has 7489 rows and 11 columns.

```
[ ] df = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/property_listing_data_in_Bangladesh.csv')
    df.head()
```

	title	beds	bath	area	adress	type	purpose	flooPlan	url	lastUpdated	price
0	Eminent Apartment Of 2200 Sq Ft Is Vacant For ...	3	4	2,200 sqft	Block A, Bashundhara R-A, Dhaka	Apartment	For Rent	https://images-cdn.bproperty.com/thumbnails/10...	https://www.bproperty.com/en/property/details-...	August 13, 2022	50 Thousand
1	Apartment Ready To Rent In South Khulshi, Near...	3	4	1,400 sqft	South Khulshi, Khulshi, Chattogram	Apartment	For Rent	https://images-cdn.bproperty.com/thumbnails/44...	https://www.bproperty.com/en/property/details-...	January 25, 2022	30 Thousand
2	Smartly priced 1950 SQ FT apartment, that you ...	3	4	1,950 sqft	Block F, Bashundhara R-A, Dhaka	Apartment	For Rent	https://images-cdn.bproperty.com/thumbnails/11...	https://www.bproperty.com/en/property/details-...	February 22, 2023	30 Thousand
3	2000 Sq Ft Residential Apartment Is Up For Ren...	3	3	2,000 sqft	Sector 9, Uttara, Dhaka	Apartment	For Rent	https://images-cdn.bproperty.com/thumbnails/14...	https://www.bproperty.com/en/property/details-...	October 28, 2021	35 Thousand
4	Strongly Structured This 1650 Sq.	3	4	1,650 sqft	Block I, Bashundhara R-A, Dhaka	Apartment	For Rent	https://images-cdn.bproperty.com/thumbnails/10...	https://www.bproperty.com/en/property/details-...	February 19, 2023	25 Thousand

Figure 2.10: Original Datasets before feature Selection

2.2.2 Implementation

Data Preprocessing: Missing values in the dataset are identified and addressed and rows with 'Duplex' and 'Building' types are removed. Then The target variable 'price' and independent variables are separated.

```
unique_suffix = set()
m = len(y)
for i in range(m):
    split_strings = y[i].split()
    unique_suffix.add(split_strings[1])

print(unique_suffix)

for i in range(m):
    split_strings = y[i].split()
    if split_strings[1] == 'Thousand':
        revised_price = float(split_strings[0]) * 1000
        revised_price = int(revised_price)
    else:
        revised_price = float(split_strings[0]) * 100000
        revised_price = int(revised_price)
    y[i] = revised_price
print(y)
```

```
{'Thousand', 'Lakh'}
[50000 30000 30000 ... 22000 175000 90000]
```

Figure 2.11: 'Thousands' & 'Lakhs' are converted

```python
for i in range(X.shape[0]) :
  split_strings = X[i][2].split()
  num = ""
  for digit in split_strings[0] :
    if(digit != ',') :
      num += digit
  X[i][2] = num
X
```

```
array([['3 ', '4 ', '2200'],
       ['3 ', '4 ', '1400'],
       ['3 ', '4 ', '1950'],
       ...,
       ['2 ', '2 ', '1000'],
       ['3 ', '4 ', '3600'],
       ['4 ', '4 ', '2600']], dtype=object)
```

```python
for i in range(len(X)):
  if X[i][1] == '1 Bath' :
    X[i][1] = '1'

for i in range(len(X)):
  if X[i][0] == '1 Bed' :
    X[i][0] = '1'
```

Figure 2.12: Some string values mapped into numerical

Feature Selection: But only the bedrooms, bathrooms, and area in square feet were used in this model, the rest were dropped and resulted dataset in figure 2.13. The target variable for the linear regression would typically be the "price" column, which represents the rental price of the apartments.

```
[ ] y = df['price']
    X = df.drop(columns = ['price', 'title', 'adress', 'type', 'purpose',
    'flooPlan', 'url', 'lastUpdated'], axis=1)
    print(y.shape)
    print(X.shape)
```

```
(7489,)
(7489, 3)
```

```
[ ] X.head()
```

	beds	bath	area
0	3	4	2,200 sqft
1	3	4	1,400 sqft
2	3	4	1,950 sqft
3	3	3	2,000 sqft
4	3	4	1,650 sqft

Figure 2.13: Datasets after feature Selection

Data Visualization, Normalization: Scatter plots and histograms are created to visualize relationships between variables. Normalization involves the application of Z-score normalization to both the independent variables and the target variable. Subsequently, the dataset is divided into training, validation, and test sets using the train_test_split function.

Gradient Descent Implementation: Functions for computing cost and gradient are defined. These two functions is called from the gradient descent function. The optimization of the model's parameters (w and b) is achieved by implementing gradient descent.

```python
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
```

```python
def compute_gradient(X, y, w, b):
    m,n = X.shape
    d1 = np.zeros((n,))
    d2 = 0.
    for i in range(m):
        gap = (np.dot(X[i], w) + b) - y[i]
        for j in range(n):
            d1[j] = d1[j] + gap * X[i, j]
        d2 = d2 + gap
    d1 = d1 / m
    d2 = d2 / m

    return d2, d1
```

Figure 2.14: Compute Gradient and Cost Function

```python
def gradient_descent(X_train, y_train, X_val, y_val, w_in, b_in, LR, iteration):

    J_train = []
    J_validation = []
    w = w_in
    b = b_in

    for i in range(iteration):

        d2,d1 = compute_gradient(X_train, y_train, w, b)

        w = w - LR * d1
        b = b - LR * d2

        J_train.append(compute_cost(X_train, y_train, w, b))
        J_validation.append(compute_cost(X_val, y_val, w, b))

        print(f"Iteration {i:4d}: Training Cost {J_train[-1]:8.2f} Validation Cost {J_validation[-1]:8.2f} ")

    return w, b, J_train
```

Figure 2.15: Gradient Descent Function

2.2.3 Performance Evaluation

The gradient descent algorithm is applied to train the linear regression model. Training and validation costs are tracked over iterations. The model's performance is evaluated using predicted values on the test set.

```
Iteration    0: Training Cost 657437704.12 Validation Cost 660165110.82
Iteration    1: Training Cost 494504752.53 Validation Cost 514188257.57
Iteration    2: Training Cost 430844569.53 Validation Cost 442938769.25
Iteration    3: Training Cost 395574165.69 Validation Cost 405499439.67
Iteration    4: Training Cost 372180674.29 Validation Cost 379294150.28
Iteration    5: Training Cost 355442548.92 Validation Cost 360312398.73
Iteration    6: Training Cost 343123143.03 Validation Cost 346021769.02
Iteration    7: Training Cost 333963165.03 Validation Cost 335181891.44
Iteration    8: Training Cost 327126014.56 Validation Cost 326904645.87
Iteration    9: Training Cost 322013806.22 Validation Cost 320560048.01
Iteration   10: Training Cost 318187353.93 Validation Cost 315677564.16
Iteration   11: Training Cost 315320855.89 Validation Cost 311905261.95
Iteration   12: Training Cost 313171742.08 Validation Cost 308978219.14
Iteration   13: Training Cost 311559133.67 Validation Cost 306696698.75
Iteration   14: Training Cost 310348038.82 Validation Cost 304909769.56
Iteration   15: Training Cost 309437646.83 Validation Cost 303503146.59
Iteration   16: Training Cost 308752628.02 Validation Cost 302390087.16
Iteration   17: Training Cost 308236658.68 Validation Cost 301504580.17
Iteration   18: Training Cost 307847598.67 Validation Cost 300796244.01
Iteration   19: Training Cost 307553898.05 Validation Cost 300226502.10
Iteration   20: Training Cost 307331918.51 Validation Cost 299765712.91
Iteration   21: Training Cost 307163935.61 Validation Cost 299391013.72
Iteration   22: Training Cost 307036648.29 Validation Cost 299084698.17
Iteration   23: Training Cost 306940066.22 Validation Cost 298832993.56
Iteration   24: Training Cost 306866678.74 Validation Cost 298625137.58
Iteration   25: Training Cost 306810833.90 Validation Cost 298452679.67
Iteration   26: Training Cost 306768273.99 Validation Cost 298308951.18
Iteration   27: Training Cost 306735788.17 Validation Cost 298188662.43
```

Figure 2.16: Training and Validation Error in Each iteration

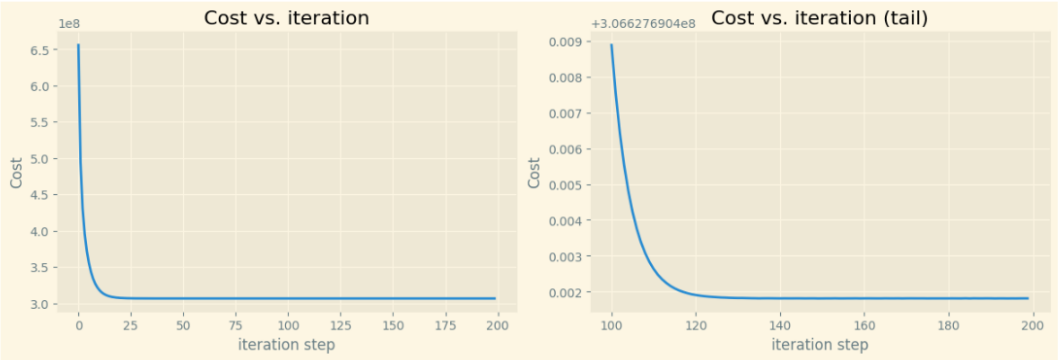

Figure 2.17: Gradient Curve

```
Mean Squared Error: 483312539.7450933
Root Mean Squared Error: 21984.370351344915
Mean Absolute Error: 12652.132016756397
R-squared Score: 0.7083114076606765
```

Figure 2.18: Error Metrics

2.3 K-Means Clustering

Clustering is a data analysis technique that groups similar data points together based on their characteristics. K-means algorithm [5], a widely used clustering method, groups data points into clusters by minimizing the distance between the points and the centroid of the cluster. It finds applications in customer grouping, image size reduction, and identification of irregularities. K-means aims to find distinct clusters and works iteratively to optimize the cluster centroids, making it an efficient and scalable algorithm for various data grouping tasks. Its purpose is to simplify data analysis, reveal patterns, and enable efficient data grouping for various applications like customer segmentation, image compression, and anomaly detection.

2.3.1 Dataset

A synthetic dataset of 15 coordinates in 3D was used as the dataset for clustering. The dataset was prepared manually, without any kind of further choice of points. The dataset is shown in figure 2.19.

	0	1	2
0	1.0	1.0	1.0
1	1.0	1.3	1.4
2	2.2	2.5	3.0
3	5.0	6.0	7.0
4	5.5	6.7	7.0
5	8.2	10.0	6.5
6	8.0	9.0	10.0
7	9.5	12.0	10.0
8	10.0	11.0	12.0
9	11.0	11.0	11.5
10	5.5	6.5	7.5
11	2.0	3.0	4.0
12	7.0	8.0	9.0
13	3.0	8.0	4.0
14	8.0	2.0	5.0

Figure 2.19: Synthetic Datasets for K-means

2.3.2 Implementation

Cluster Assignment: For the cluster assignment, at first, the initial cluster array was declared as the dummy. And for each of the centroids in the set of centroids, the distance was calculated. Then the minimum distance centroid was assigned as the cluster center.

```
def ClusterAssignment(points,centroidList):
  k = len(centroidList)
  clusters = []
  for i in range(k) : clusters.append([])
  for curr_point in points:
    temp = []
    for curr_centroid in centroidList:
      curr_distance = FindDistance(curr_point, curr_centroid)
      temp.append(curr_distance)
    cluster_index = temp.index(min(temp))
    clusters[cluster_index].append(curr_point)
  return clusters
```

Figure 2.20: Cluster Assignment Function

Centroid Update: The centroid was updated by the clusters' coordinates taking the sum over the length of the clusters. A loop iterates over the clusters to calculate each new centroid.

```
[ ] def UpdateCentroid(clusters):
      centroids = []
      for cluster in clusters:
        centroid = [sum(coordinates) / len(cluster) for coordinates in zip(*cluster)]
        centroids.append(centroid)
      return centroids
```

Figure 2.21: Centroid Update

K-means Function: In the k-means implementation, we initially took the centroid at random. Then call the corresponding cluster assignment function and centroid update function to get the clusters and their new centroid. K-means algorithm stopped when no new cluster was formed.

```
[ ]  def kmeans(points, k):
       centroids = initial_centroid(points, k)
       while True:
         clusters = ClusterAssignment(points, centroids)
         new_centroids = UpdateCentroid(clusters)
         print("old centroids",centroids)
         print("new centroids",new_centroids)
         if check_with_prev(centroids, new_centroids):
           break
         centroids = new_centroids
         plotting(clusters, centroids)
       return clusters, centroids
```

Figure 2.22: K-means Implementation

2.3.3 Performance Evaluation

The dataset took several iterations to get the final clustering. It was plotted using Matplotlib. Here is an overview of the final result given in 2.26.

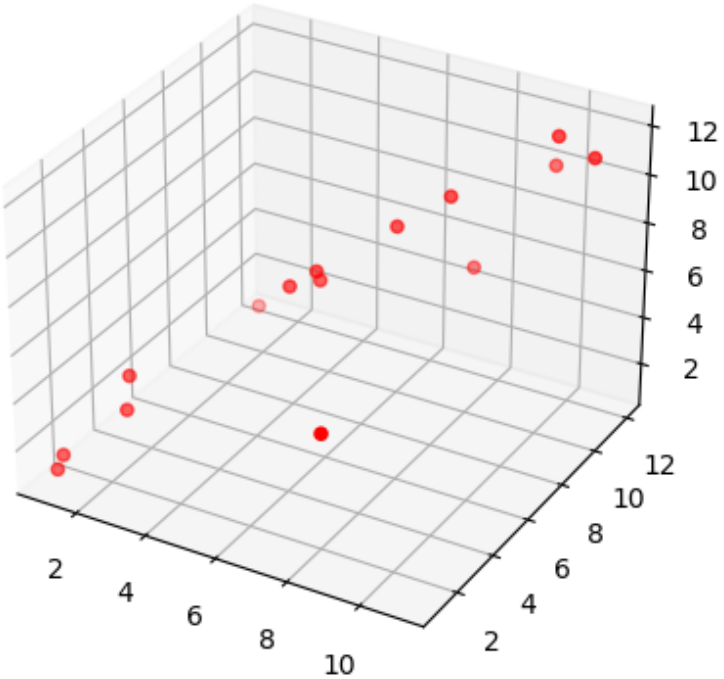

Figure 2.23: Initial Data points

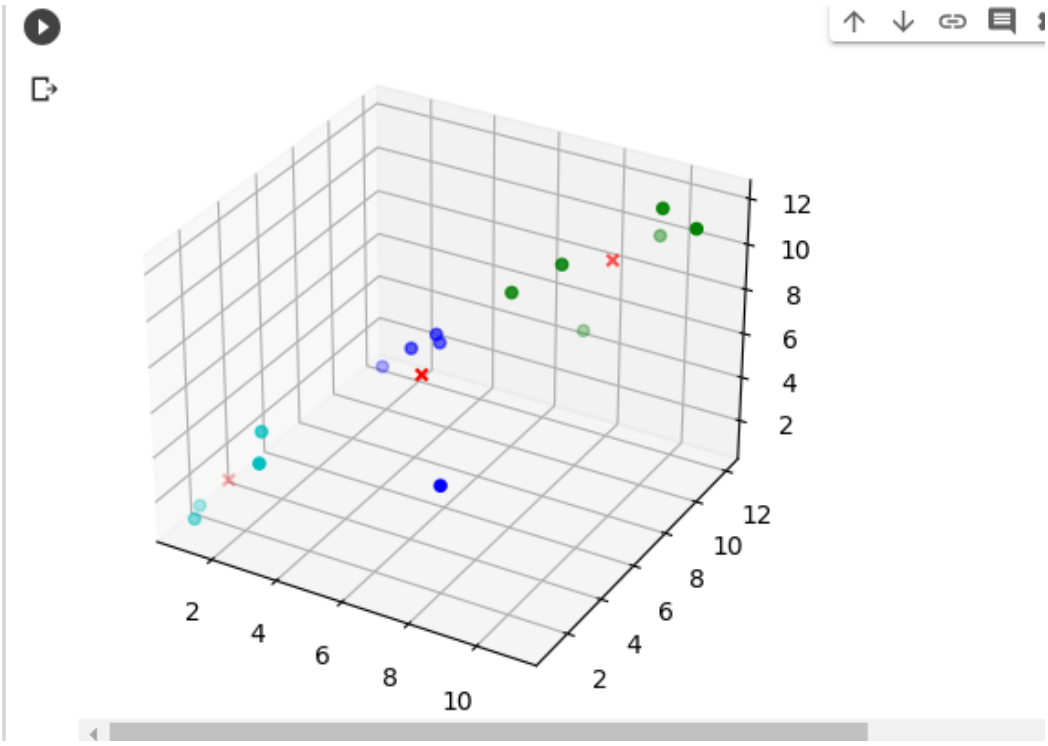

Figure 2.24: Resulting Clusters

Besides the WCSS error was also calculated and it was not so much. Which suggests a good clustering was done.

```
[ ] def calculate_ssd(clusters, centroids):
        ssd = 0
        for i, cluster in enumerate(clusters):
            for point in cluster:
                ssd += FindDistance(point, centroids[i]) ** 2
        return ssd

    # ... (Your existing code after clustering)

    ssd = calculate_ssd(clusters, centroids)
    print("Sum of Squared Distances (SSD):", ssd)
```

```
Sum of Squared Distances (SSD): 93.38366666666668
```

Figure 2.25: Sum of Squared Distance Error

```python
def calculate_silhouette_score(clusters):
    silhouette_scores = []
    for cluster_idx, cluster in enumerate(clusters):
        for point in cluster:
            a = np.mean([FindDistance(point, other_point) for other_point in cluster if other_point != point])
            b = min([
                np.mean([FindDistance(point, other_point) for other_point in other_cluster])
                for other_cluster_idx, other_cluster in enumerate(clusters) if other_cluster_idx != cluster_idx
            ])
            s = (b - a) / max(a, b)
            silhouette_scores.append(s)
    avg_silhouette_score = np.mean(silhouette_scores)
    return avg_silhouette_score

# ... (Your existing code after clustering)

silhouette_score = calculate_silhouette_score(clusters)
print("Silhouette Score (Custom Calculation):", silhouette_score)
```

```
Silhouette Score (Custom Calculation): 0.4791614903733417
```

Figure 2.26: Silhouette Score

2.4 Decision Tree

Decision trees serve as versatile instruments within machine learning, adeptly tackling classification and regression tasks by iteratively dividing input data using feature values. This process constructs a tree framework, where inner nodes signify decisions and terminal nodes hold predictions. For classification, they assign labels; for regression, they foresee numeric outcomes. These trees offer interpretability, accommodate diverse data types, and boast efficiency, yet vulnerability to overfitting, instability, and bias exists. To bolster performance, ensemble techniques like random forests and gradient boosting harness the strength of multiple trees. Despite their limitations, decision trees persist as a pivotal element, valued for their simplicity, practicality, and indispensable role in the panorama of machine learning methodologies.

2.4.1 Dataset

For the implementation of the decision tree algorithm, a golf dataset was used. The dataset contains both the categorical and numerical values. The numerical columns in the dataset was the 'Humidity'. It has three more categorical columns as well. The dataset comprises a synthetic total of 14 records.

	Temperature	Outlook	Humidity	Windy	Play?
0	hot	sunny	60	False	no
1	hot	sunny	70	True	no
2	hot	overcast	80	False	yes
3	cool	rain	40	False	yes
4	cool	overcast	30	True	yes
5	mild	sunny	100	False	no
6	cool	sunny	20	False	yes
7	mild	rain	45	False	yes
8	mild	sunny	25	True	yes
9	mild	overcast	85	True	yes
10	hot	overcast	20	False	yes
11	mild	rain	95	True	no
12	cool	rain	35	True	no
13	mild	rain	55	False	yes

Figure 2.27: Dataset For Decision Tree

2.4.2 Implementation

Preprocessing: For the categorical data one hot encoding was used. As one of the column in the dataset contains continuous values, so for that column it was scaled in a range of value either greater than 50 or less than 50. If the value in the column is greater than 50, then it is assumed that the column has higher humidity and for less than 50, the value taken is 'low'. Later it is also transformed as 0 and 1.

Entropy and Information Gain Calculation: In order for the selection of the attributes at each label, the calculation of entropy and information gain was necessary. The formula for the entropy and information gain is shown below,

$$H(S) = -p_1 \log_2(p_1) - p_2 \log_2(p_2)$$

$$IG(S, A) = H(S) - \sum_{v \in \text{values}(A)} \frac{|S_v|}{|S|} \cdot H(S_v)$$

Where:

$|S_v|$ is the number of instances with a value $v$corresponding to feature A,

$|S|$ denotes the complete count of instances within the set S,

$H(S_v)$ is the entropy of the subset of instances with a value v corresponding to the feature A.

The function in 2.28 shows the way the above two formula was implemented in the code.

```
[ ] def entropy(data):
        labels = data['Play?']
        total_instances = len(labels)
        unique_labels = labels.unique()
        entropy_val = 0

        for label in unique_labels:
            p = len(labels[labels == label]) / total_instances
            entropy_val -= p * math.log2(p)

        return entropy_val
```

```
[ ] def information_gain(data, attribute):

        total_instances = len(data)
        attribute_entropy = 0

        for value in data[attribute].unique():
            subset = data[data[attribute] == value]
            subset_entropy = entropy(subset) * len(subset) / total_instances
            attribute_entropy += subset_entropy

        return entropy(data) - attribute_entropy
```

Figure 2.28: Entropy and Information Gain Function

Decision Tree Build Up and Tree Pruning Criteria: In the decision tree building process as it goes deeper at each level it is calculating the information gain for each attributes and doing that for all the nodes. But when the information gain becomes 1, then it implies that a decision or a split based on a particular feature has resulted in a perfect separation of the data into their respective classes.

```python
def build_decision_tree(data, attributes):

    if len(data['Play?'].unique()) == 1:
        return data['Play?'].iloc[0]

    if len(attributes) == 0:
        return data['Play?'].value_counts().idxmax()

    max_gain = -1
    best_attribute = None
    for attribute in attributes:
        gain = information_gain(data, attribute)
        if gain > max_gain:
            max_gain = gain
            best_attribute = attribute

    tree = {best_attribute: {}}
    remaining_attributes = [attr for attr in attributes if attr != best_attribute]

    for value in data[best_attribute].unique():
        subset = data[data[best_attribute] == value]
        subtree = build_decision_tree(subset, remaining_attributes)
        tree[best_attribute][value] = subtree

    return tree
```

Figure 2.29: Decision Tree Implementation

2.4.3 Performance Evaluation

The resulting decision tree shows how the tree is constructed based on the attributes and their splitting way. Besides, a prediction function was built to show how the prediction of the implemented model comes. Figure 2.30 shows the decision tree in the dictionary format where in the implemented dataset the root that is gotten is 'Outlook'.

```
{'Outlook': {'overcast': 'yes',
             'rain': {'Windy': {False: 'yes', True: 'no'}},
             'sunny': {'Humidity': {'High': 'no', 'Low': 'yes'}}}}
```

Figure 2.30: Resulting Decision Tree

2.5 Artificial Neural Network (ANN)

Artificial Neural Networks (ANNs) [6] form the foundational basis of deep learning, which represents a specialized field within machine learning. ANNs are composed of intricate layers of interconnected neurons, utilizing activation functions to process input data. Neurons are linked by weights and biases, and the process

of forward propagation calculates outcomes as they traverse the network. Training ANNs involve backpropagation, a mechanism where optimization techniques modify weights to minimize a loss function that gauges the variance between predictions and actual values. Deep learning extends the power of ANNs by incorporating multiple hidden layers, facilitating the automatic extraction of salient features from input data. Deeper layers have the capacity to capture complex, nuanced patterns, endowing deep neural networks with exceptional prowess in several tasks.

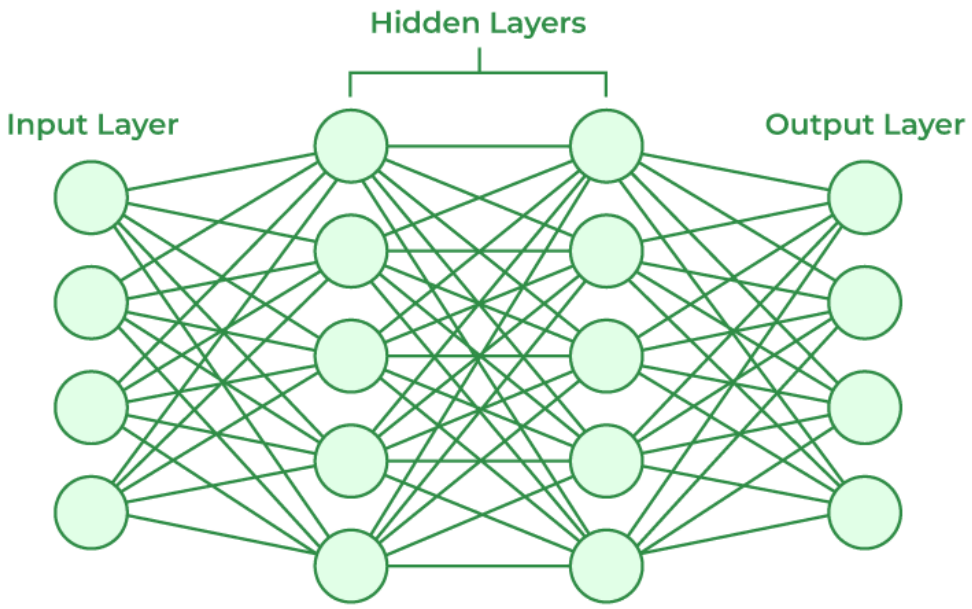

Figure 2.31: Artificial Neural Networks

2.5.1 Dataset

A dataset of Breast Cancer comprises 569 instances and encompasses 32 attributes taken from Wisconsin [7], capturing essential breast tumor characteristics for classification analysis was used in this implementation. The attributes cover a diverse set of measurements, incorporating parameters like average radius, average texture, average perimeter, and average area of the tumors. Additionally, features such as average smoothness, average compactness, average concavity, and average number of concave points on the contour contribute to the comprehensive portrayal of tumor characteristics. Significantly, the 'diagnosis' column assumes the role of the target variable, signifying whether tumors are categorized

as malignant (M) or benign (B). This dataset provides a valuable resource for the development of predictive models to discern tumor malignancy based on an array of tumor-related attributes.

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.30010
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08690
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19740
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.24140
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.19800
...	...	...	...	...	...	...	...	...	...
564	926424	M	21.56	22.39	142.00	1479.0	0.11100	0.11590	0.24390
565	926682	M	20.13	28.25	131.20	1261.0	0.09780	0.10340	0.14400
566	926954	M	16.60	28.08	108.30	858.1	0.08455	0.10230	0.09251
567	927241	M	20.60	29.33	140.10	1265.0	0.11780	0.27700	0.35140
568	92751	B	7.76	24.54	47.92	181.0	0.05263	0.04362	0.00000

569 rows × 32 columns

Figure 2.32: A Snapshot of the Dataset

2.5.2 Implementation

The process of building and training a neural network involves key elements that contribute to its effectiveness. The architecture design of the neural network determines the structure and connections between its layers. The network's capacity to understand complex links inside the data is made possible by activation functions, which play a crucial role in adding non-linearity into the system. Mostly used activation functions like the sigmoid function transform input signals into output activations, enhancing the network's ability to model intricate patterns. Optimization algorithms, such as backpropagation, are employed to adjust the network's weights and biases during training. Backpropagation calculates gradients that indicate the direction and magnitude of parameter updates, helping the network minimize errors and improve its predictions. These interconnected components—architecture design, activation functions, and optimization algorithms—work in harmony to create a neural network capable of learning and making accurate predictions from the input data.

Architecture Design: The architecture of a neural network defines its layout, including the number of layers and units in each layer. It's crucial to strike a balance between complexity and efficiency.

```python
def initialize_parameters(input_size, hidden_size, output_size):
    hidden_weights = np.random.randn(input_size, hidden_size)
    hidden_biases = np.zeros((1, hidden_size))
    output_weights = np.random.randn(hidden_size, output_size)
    output_biases = np.zeros((1, output_size))

    return hidden_weights, hidden_biases, output_weights, output_biases
```

Figure 2.33: Initialized the Size of Hidden Layers

Activation Functions: The network's capacity to understand complex links inside the data is made possible by activation functions, which play a crucial role in adding non-linearity into the system. The sigmoid activation function is commonly used, mapping input values to values between 0 and 1.

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)
```

Figure 2.34: Activation Functions Used

Optimization Algorithms: Optimization algorithms update the network's weights and biases during training to minimize errors. Backpropagation calculates the derivatives of the loss function concerning the network's parameters, allowing us to adjust them in the right direction.

```python
def forward_propagation(X, hidden_weights, hidden_biases, output_weights, output_biases):
    hidden_layer_input = np.dot(X, hidden_weights) + hidden_biases
    hidden_layer_output = sigmoid(hidden_layer_input)
    output_layer_input = np.dot(hidden_layer_output, output_weights) + output_biases
    output_layer_output = sigmoid(output_layer_input)

    return hidden_layer_output, output_layer_output
```

```python
def backpropagation(X, y, hidden_layer_output, output_layer_output, hidden_weights, hidden_biases, output_weights, output_biases, learning_rate):
    output_error = y - output_layer_output
    output_delta = output_error * sigmoid_derivative(output_layer_output)

    hidden_error = np.dot(output_delta, output_weights.T)
    hidden_delta = hidden_error * sigmoid_derivative(hidden_layer_output)

    output_weights += np.dot(hidden_layer_output.T, output_delta) * learning_rate
    output_biases += np.sum(output_delta, axis=0, keepdims=True) * learning_rate

    hidden_weights += np.dot(X.T, hidden_delta) * learning_rate
    hidden_biases += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate

    return hidden_weights, hidden_biases, output_weights, output_biases
```

Figure 2.35: Forward Propagation and Back Propagation

2.5.3 Performance Evaluation

The neural network achieved an accuracy of approximately 98.25% on the test data. The confusion matrix reveals that out of 71 benign cases, 70 were correctly classified, and out of 43 malignant cases, 42 were correctly classified. The classification report provides further insights: for the "Benign" class, the precision, recall, and F1-score are all around 0.99, indicating high accuracy. Similarly, for the "Malignant" class, precision, recall, and F1-score are approximately 0.98, indicating effective performance. The overall weighted average F1-score is also 0.98, confirming the model's strong predictive capability. This demonstrates that the neural network architecture, employing appropriate activation functions and optimization algorithms, effectively classified the breast cancer dataset with impressive accuracy.

```
Accuracy: 0.9824561403508771
Confusion Matrix:
[[70  1]
 [ 1 42]]
Classification Report:
              precision    recall  f1-score   support

      Benign       0.99      0.99      0.99        71
   Malignant       0.98      0.98      0.98        43

    accuracy                           0.98       114
   macro avg       0.98      0.98      0.98       114
weighted avg       0.98      0.98      0.98       114
```

Figure 2.36: Performance Metrics

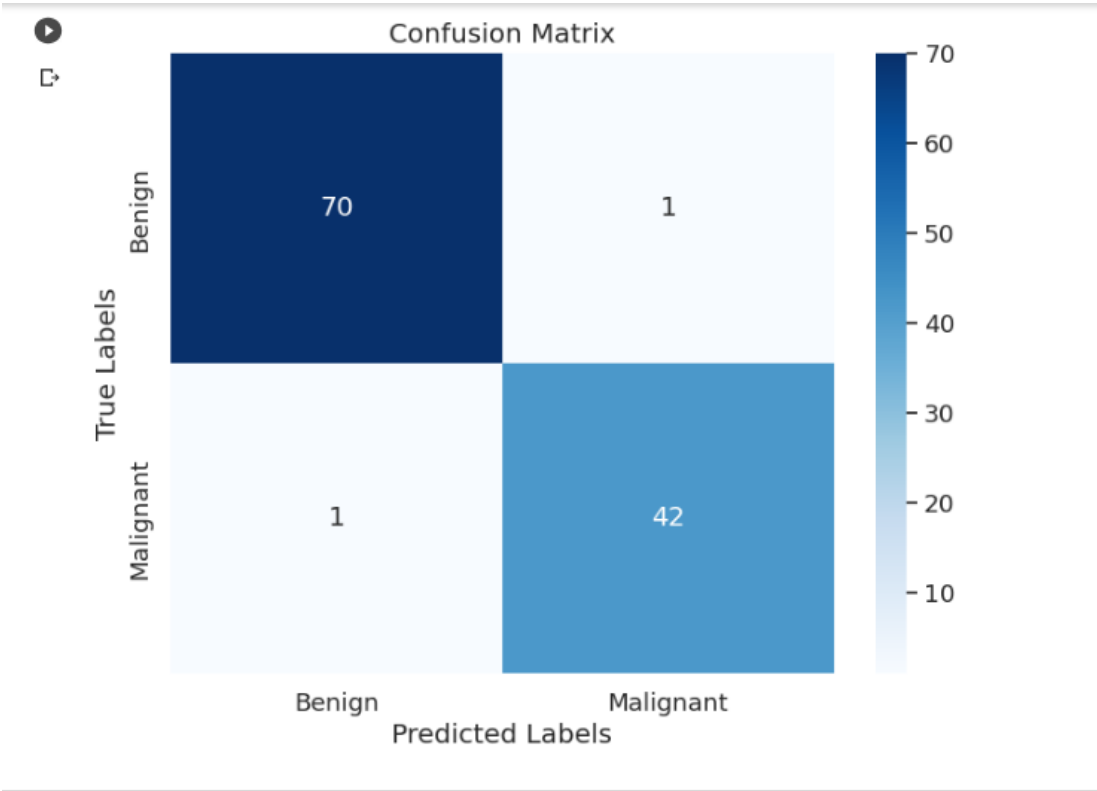

Figure 2.37: Confusion Metrics

3 Discussion

This study entails the implementation of five basic machine-learning algorithms. Each algorithm exhibits distinct strengths and inherent weaknesses, catering to a range of practical applications. The Apriori algorithm efficiently discovers association rules, aiding market basket analysis and personalized recommendations. Multivariable Linear Regression's interpretability suits predicting continuous outcomes like real estate prices. K-Means Clustering efficiently segments customers and compresses images, yet relies on preset cluster counts and initial centroids. Decision Trees excel in interpretable models for medical diagnosis and fraud detection, though prone to overfitting. Artificial Neural Networks master intricate patterns for image recognition and language processing, but demand substantial resources and challenge interpretability. Algorithm choice depends on data characteristics and problem specifics. During the implementation, one of the key challenges was the preprocessing part. It seems that the preprocessing is more important than just feeding a dataset into the model and getting the output. While implementing the linear regression, it was found difficult to handle different matrix operations, in clustering mostly the implementation difficulty was the key issue. On the other hand in the decision tree, understanding the nature of the recursive functions and finally, in the backpropagation hidden layer's activation value generation method was all those were faced in each of the separate labs. However, by several trials to understand the algorithm and write code, knowledge gathering about numpy to remove implementation difficulty helped us a lot.

4 Conclusion

The report underscores the notable achievement of the students in successfully implementing and analyzing five prominent machine learning algorithms: the Apriori algorithm, Decision Tree, Multivariable Linear Regression, K-means clustering, and Artificial Neural Network (ANN). The students have showcased their ability to navigate the intricacies of these algorithms, effectively applying them to real-world scenarios. This hands-on experience has proven invaluable in not

only deepening their understanding of machine-learning concepts but also enhancing their problem-solving skills. Through the practical application of these algorithms, the students have gained insights into their strengths, weaknesses, and practical applications. This practical approach to learning reinforces the importance of experiential learning in the realm of machine learning, enabling students to bridge the gap between theoretical knowledge and practical implementation.

5 Appendices

Code 1: Apriori Implementation

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from collections import Counter
from itertools import combinations, islice

def load_data(file_path):
    df = pd.read_csv(file_path, header=None)
    return df

def preprocess_data(df):
    df = df.fillna("Missing")
    data = []
    for i in range(len(df)):
        row_num = i
        curr = []
        for j in df[i]:
            if j != 'Missing':
                curr.append(j)
        data.append([row_num, curr])
    return data

def input_support_confidence():
    min_support = float(input("Input Minimum Support(%): ")) /
   100
    min_confidence = float(input("Input Minimum Confidence(%): ")
   )
```

```
    support_count = int(min_support * len(data))
    return min_support, min_confidence, support_count

def print_results(item_set_counter, k, result_type):
    print(result_type + str(k) + ":")
    for item_set in item_set_counter:
        count = item_set_counter[item_set]
        if k == 1 and result_type == 'C':
            print(str([item_set]) + ": " + str(count))
        else:
            print(str(list(item_set)) + ": " + str(count))
    print()

def initialize_candidate_set(data):
    unique_items = []
    for lst in data:
        for items in lst[1]:
            if items not in unique_items:
                unique_items.append(items)
    unique_items = sorted(unique_items)
    candidate_set = Counter()
    for items in unique_items:
        for transaction in data:
            if items in transaction[1]:
                candidate_set[items] += 1
    print_results(candidate_set, 1, "C")
    frequent_item_set = Counter()
    for item_set in candidate_set:
        if candidate_set[item_set] >= support_count:
            frequent_item_set[frozenset([item_set])] +=
   candidate_set[item_set]
    return frequent_item_set

def apriori_algorithm(data):
    count = 1
    while count <= 10000:
        if count == 1:
            frequent_item_set = initialize_candidate_set(data)
            print_results(frequent_item_set, 1, "L")
```

```
            previous_L = frequent_item_set
            sequence = 1
            count += 1
            continue
        possible_candidate_set = set()
        prev_L = list(previous_L)
        for i in range(0, len(prev_L)):
            int st = i + 1;
            for j in range(st, len(prev_L)):
                total = prev_L[i].union(prev_L[j])
                if len(total) == count:
                    possible_candidate_set.add(prev_L[i].union(prev_L[j]))
        possible_candidate_set = list(possible_candidate_set)
        candidate_set = Counter()
        for item_set in possible_candidate_set:
            candidate_set[item_set] = 0
            for row in data:
                transaction = set(row[1])
                if item_set.issubset(transaction):
                    candidate_set[item_set] += 1
        print_results(candidate_set, count, "C")
        print()
        frequent_item_set = Counter()
        for item_set in candidate_set:
            if candidate_set[item_set] >= support_count:
                frequent_item_set[item_set] += candidate_set[item_set]
        if len(frequent_item_set) != 0:
            print_results(frequent_item_set, count, "L")
        else:
            print("This is the Final(frequent set)")
            print_results(previous_L, sequence, "L")
            break
        count += 1
        previous_L = frequent_item_set
        sequence = count
    return previous_L, sequence

```

```
def print_rules(lst1, lst2, confidence):
    for i in range(0, len(lst1)):
        curr = lst1[i]
        if i == len(lst1) - 1:
            print(curr, end=" ")
            print("----->", end=' ')
        else:
            print(curr, end=", ")
    for i in range(0, len(lst2)):
        curr = lst2[i]
        print(curr, end=", ")
        if i == len(lst2) - 1:
            print(curr, end=" ")
            print(": " + str(confidence))
        else:
            print(curr, end=", ")

def generate_association_rules(frequent_item_sets):
    mapping = {}
    for item_set in frequent_item_sets:
        for r in range(1, len(item_set)):
            combination = []
            for i in combinations(item_set, r):
                combination.append(frozenset(i))
            for x in combination:
                y = item_set - x
                xy = item_set
                support_xy = support_x = support_y = 0
                for j in data:
                    transaction = set(j[1])
                    if x.issubset(transaction):
                        support_x += 1
                    if y.issubset(transaction):
                        support_y += 1
                    if xy.issubset(transaction):
                        support_xy += 1
                confidence_x = support_xy / support_x * 100
                confidence_y = support_xy / support_y * 100
                print_rules(list(x), list(y), confidence_x)
```

```
                print_rules(list(y), list(x), confidence_y)
                plus = ["->"]
                merged_string = " ".join(list(x) + plus + list(y))
                mapping[merged_string] = confidence_x
                merged_string = " ".join(list(y) + plus + list(x))
                mapping[merged_string] = confidence_y
        print()
    return mapping

def main():
    file_path = '/content/drive/MyDrive/Colab Notebooks/store_data.csv'
    df = load_data(file_path)
    data = preprocess_data(df)
    min_support, min_confidence, support_count = input_support_confidence()
    frequent_item_sets, sequence = apriori_algorithm(data)
    mapping = generate_association_rules(frequent_item_sets)
    N = int(input("Enter a value N to find first N number of Association rules of Highest Confidence Value in(%): "))
    sorted_map = dict(sorted(mapping.items(), key=lambda x: x[1], reverse=True))
    first_N_values = dict(islice(sorted_map.items(), N))
    for key, value in first_N_values.items():
        formatted_value = "{:.2f}".format(value)
        print(f"Association Rule: [{key}], Confidence Score : {formatted_value} %")

if __name__ == "__main__":
    main()
```

Code 2: Linear Regression Implementation

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error,
    mean_absolute_error, r2_score
import seaborn as sn
import copy
import math
plt.style.use('Solarize_Light2')


df = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/
    property_listing_data_in_Bangladesh.csv')
df.head()

df.shape

df.describe()

df.info()

df.isnull().sum()

rows_with_nan = df[df.isna().any(axis=1)]
rows_with_nan



print(df[df['type'] == 'Duplex'].shape[0])

print(df[df['type'] == 'Building'].shape[0])

df = df[~df['type'].isin(['Duplex', 'Building'])]
df.shape

y = df['price']
X = df.drop(columns = ['price', 'title', 'adress', 'type', '
    purpose',
'flooPlan', 'url', 'lastUpdated'], axis=1)
print(y.shape)
```

```
print(X.shape)

X.head()

y

X['beds'].unique()

X['bath'].unique()

X = np.asarray(X)
print(X)
y = np.asarray(y)
print(y)

type(y[0])

print(type(X[0][0]))
print(type(X[0][1]))
print(type(X[0][2]))

unique_suffix = set()
m = len(y)
for i in range(m):
    split_strings = y[i].split()
    unique_suffix.add(split_strings[1])

print(unique_suffix)

for i in range(m):
    split_strings = y[i].split()
    if split_strings[1] == 'Thousand':
        revised_price = float(split_strings[0]) * 1000
        revised_price = int(revised_price)
    else:
        revised_price = float(split_strings[0]) * 100000
        revised_price = int(revised_price)
    y[i] = revised_price
print(y)
```

```

for i in range(X.shape[0]) :
  split_strings = X[i][2].split()
  num = ""
  for digit in split_strings[0] :
    if(digit != ',') :
      num += digit
  X[i][2] = num
X

for i in range(len(X)):
  if X[i][1] == '1 Bath' :
    X[i][1] = '1'

for i in range(len(X)):
  if X[i][0] == '1 Bed' :
    X[i][0] = '1'

X.shape

X = X.astype(float)
X[0:100, 0]



fig,ax=plt.subplots(1, 3, figsize=(12, 3), sharey=True)

for i in range(len(ax)):
    ax[i].scatter(X[:,i],y, color = 'blue')
    if i == 0 :
      ax[i].set_xlabel('Beds')
    elif i == 1 :
      ax[i].set_xlabel('Bath')
    else :
      ax[i].set_xlabel('size(sqft)')

ax[0].set_ylabel("Price")
ax[1].set_ylabel("Price")
ax[1].set_ylabel("Price")
```

```
plt.show()

plt.hist(X[:, 0], color = 'green', rwidth = 0.95)
plt.xlabel("Beds")
plt.ylabel("Count")

plt.hist(X[:, 1], rwidth = 0.95, label = 'Bath')
plt.xlabel("Bath")
plt.ylabel("Count")

plt.hist(X[:, 2], bins = 1000, color = 'black')
plt.xlabel("size(sqft)")
plt.ylabel("Count")
# plt.xlim(0, 100000)



def zscore_normalize_features(X):
  mu     = np.mean(X, axis=0)
  sigma  = np.std(X, axis=0)
  # print(mu)
  # print(sigma)
  MeanData_X = (X - mu)
  NormalizedData_X = (X - mu) / sigma
  return (NormalizedData_X, MeanData_X)

NormalizedData_X, MeanData_X = zscore_normalize_features(X)
NormalizedData_y, MeanData_y = zscore_normalize_features(y)
print(NormalizedData_X)
print(NormalizedData_y)

TrainData_X, TestData_X, TrainData_y, TestData_y =
    train_test_split(NormalizedData_X, y, test_size=0.4,
    random_state=42)
X_test, X_val, y_test, y_val = train_test_split(TestData_X,
    TestData_y, test_size=0.5, random_state=42)
print(TrainData_X)
print(TrainData_y)

```

```
print(X_test)
print(y_test)


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

def GradientComputation(X, y, w, b):
  m,n = X.shape
  d1 = np.zeros((n,))
  d2 = 0.
  for i in range(m):
      gap = (np.dot(X[i], w) + b) - y[i]
      for j in range(n):
          d1[j] = d1[j] + gap * X[i, j]
      d2 = d2 + gap
  d1 = d1 / m
  d2 = d2 / m

  return d2, d1

def GradientDescent_Fun(TrainData_X, TrainData_y, X_val, y_val,
    w_in, b_in, LR, iteration):

    J_train = []
    J_validation = []
    w = w_in
    b = b_in

    for i in range(iteration):

        d2,d1 = GradientComputation(TrainData_X, TrainData_y, w,
    b)
```

```python

        w = w - LR * d1
        b = b - LR * d2

        J_train.append(compute_cost(TrainData_X, TrainData_y, w, b))
        J_validation.append(compute_cost(X_val, y_val, w, b))

        print(f"Iteration {i:4d}: Training Cost {J_train[-1]:8.2f} Validation Cost {J_validation[-1]:8.2f} ")

    return w, b, J_train



starting_w = np.zeros((3))
starting_w = starting_w.astype(float)
starting_b = 0.
NormalizedData_w, NormalizedData_b, hist = GradientDescent_Fun(TrainData_X, TrainData_y, X_val, y_val, starting_w, starting_b, 0.5, 200)

print(f"b,w found by gradient descent: {NormalizedData_b:0.2f}, {NormalizedData_w}\n\n")
m, n = X_test.shape

PredictedData_ytest = []
TotalCost = []

for i in range(m):
    predicted_val = np.dot(X_test[i], NormalizedData_w) + NormalizedData_b
    PredictedData_ytest.append(predicted_val)

for i in range(m):
    print(f"prediction: {PredictedData_ytest[i]:0.2f}, target value: {y_test[i]}")

print()
```

```python

Cost = compute_cost(X_test, y_test, NormalizedData_w, NormalizedData_b)
print(f"Test cost computed : {Cost}\n\n")

r2 = r2_score(y_test, PredictedData_ytest)
print(f"Test R-squared Score: {r2}\n\n")

print(f"b,w found by gradient descent: {NormalizedData_b:0.2f}, {NormalizedData_w}\n\n")

sample = [4, 3, 1700] # price = 100000
sample_norm, sample_mean = zscore_normalize_features(sample)
predicted_val = np.dot(sample_norm, NormalizedData_w) + NormalizedData_b
print(predicted_val)

fig, (axiss1, axiss2) = plt.subplots(1, 2, constrained_layout=True, figsize=(12, 4))
axiss1.plot(hist)
axiss2.plot(100 + np.arange(len(hist[100:])), hist[100:])
axiss1.set_title("Cost vs. iteration");  axiss2.set_title("Cost vs. iteration (tail)")
axiss1.set_ylabel('Cost')             ;  axiss2.set_ylabel('Cost')
axiss1.set_xlabel('iteration step')   ;  axiss2.set_xlabel('iteration step')
plt.show()


model = LinearRegression()
model.fit(TrainData_X, TrainData_y)

PredictedData_ytest = model.predict(X_test)

mse = mean_squared_error(y_test, PredictedData_ytest)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_test, PredictedData_ytest)
r2 = r2_score(y_test, PredictedData_ytest)
```

```python

print("Mean Squared Error:", mse)
print("Root Mean Squared Error:", rmse)
print("Mean Absolute Error:", mae)
print("R-squared Score:", r2)
```

Code 3: K-Means Clustering

```python
import pandas as pd
import matplotlib.pyplot as plt
import random
import math
import numpy as np

df_data = pd.read_csv("/content/dataset.csv", header=None)
data_points = df_data.values.tolist()

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(df_data[0], df_data[1], df_data[2], c='blue')

def calculate_distance(point, centroid):
    squared_distance = 0
    for i in range(len(point)):
        squared_distance += (point[i] - centroid[i]) ** 2
    distance = math.sqrt(squared_distance)
    return distance

def assign_to_clusters(points, centroid_list):
    k_clusters = len(centroid_list)
    clusters = [[] for _ in range(k_clusters)]
    for curr_point in points:
        distances = []
        for curr_centroid in centroid_list:
            curr_distance = calculate_distance(curr_point, curr_centroid)
            distances.append(curr_distance)
        cluster_index = distances.index(min(distances))
        clusters[cluster_index].append(curr_point)
    return clusters
```

```python

def update_centroids(clusters):
    centroids = []
    for cluster in clusters:
        centroid = [sum(coords) / len(cluster) for coords in zip(*cluster)]
        centroids.append(centroid)
    return centroids

def check_convergence(prev_centroids, new_centroids):
    for prev_centroid, new_centroid in zip(prev_centroids, new_centroids):
        if prev_centroid != new_centroid:
            return False
    return True

def initialize_centroids(points, k):
    centroids = random.sample(points, k)
    return centroids

def k_means_clustering(points, k):
    centroids = initialize_centroids(points, k)
    while True:
        clusters = assign_to_clusters(points, centroids)
        new_centroids = update_centroids(clusters)
        if check_convergence(centroids, new_centroids):
            break
        centroids = new_centroids
        plot_clusters(clusters, centroids)
    return clusters, centroids

def plot_clusters(clusters, centroids):
    colors = ['r', 'g', 'b', 'c', 'm', 'y', 'v']
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    for i, cluster in enumerate(clusters):
        color = colors[(i + 1) % len(colors)]
        x_coords = [point[0] for point in cluster]
        y_coords = [point[1] for point in cluster]
```

```
        z_coords = [point[2] for point in cluster]
        ax.scatter(x_coords, y_coords, z_coords, c=color)

    x_centroids = [centroid[0] for centroid in centroids]
    y_centroids = [centroid[1] for centroid in centroids]
    z_centroids = [centroid[2] for centroid in centroids]
    ax.scatter(x_centroids, y_centroids, z_centroids, marker='x', c='r')

num_clusters = int(input('Enter the number of clusters '))
result_clusters, result_centroids = k_means_clustering(data_points, num_clusters)
plot_clusters(result_clusters, result_centroids)

def calculate_sum_squared_distances(clusters, centroids):
    ssd = 0
    for i, cluster in enumerate(clusters):
        for point in cluster:
            ssd += calculate_distance(point, centroids[i]) ** 2
    return ssd

ssd_result = calculate_sum_squared_distances(result_clusters, result_centroids)
print("Sum of Squared Distances (SSD):", ssd_result)

def calculate_silhouette_score(clusters):
    silhouette_scores = []
    for cluster_idx, cluster in enumerate(clusters):
        for point in cluster:
            a = np.mean([calculate_distance(point, other_point) for other_point in cluster if other_point != point])
            b = min([
                np.mean([calculate_distance(point, other_point) for other_point in other_cluster])
                for other_cluster_idx, other_cluster in enumerate(clusters) if other_cluster_idx != cluster_idx
            ])
            s = (b - a) / max(a, b)
            silhouette_scores.append(s)
```

```
    avg_silhouette_score = np.mean(silhouette_scores)
    return avg_silhouette_score

silhouette_result = calculate_silhouette_score(result_clusters)
print("Silhouette Score (Custom Calculation):", silhouette_result
    )
```

Code 4: Decision Tree

```

import pandas as pd, pprint, numpy as np, math

Data__Frame= pd.read_csv('/content/DecisionTreeTableOfDataset.csv
    ')
Data__Frame

from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder

Data__Frame['Humidity'] = Data__Frame['Humidity'].apply(lambda x:
     'High' if x > 50 else 'Low')
print(Data__Frame)
le = LabelEncoder()
Data__Frame['Humidity'] = le.fit_transform(Data__Frame['Humidity'
    ])
Data__Frame['Temperature'] = le.fit_transform(Data__Frame['
    Temperature'])
Data__Frame['Outlook'] = le.fit_transform(Data__Frame['Outlook'])
Data__Frame['Windy'] = le.fit_transform(Data__Frame['Windy'])
Data__Frame['Play?'] = le.fit_transform(Data__Frame['Play?'])
Data__Frame

X = Data__Frame.drop('Play?', axis=1)
y = Data__Frame['Play?']

TrainData_X, TestData_X, TrainData_y, TestData_y =
    train_test_split(X, y, test_size=0.2, random_state=42)

clf = DecisionTreeClassifier(random_state=42)
```

```
clf.fit(TrainData_X, TrainData_y)

TestData_X

PredictData_y = clf.predict(TestData_X)

PredictData_y

from sklearn.metrics import accuracy_score

score = accuracy_score(TestData_y, PredictData_y)
score



import pandas as pd
import numpy as np


Data__Frame= pd.read_csv('DecisionTreeTableOfDataset.csv')
Data__Frame

Data__Frame['Humidity'] = Data__Frame['Humidity'].apply(lambda x:
    'High' if x > 50 else 'Low')
Data__Frame

Data__Frame.shape

Data__Frame.info()

def entropy(TableOfData):
    ClassVal = TableOfData['Play?']
    Tot_Instance = len(ClassVal)
    unique_ClassVal = ClassVal.unique()
    entropy_val = 0

    for label in unique_ClassVal:
        p = len(ClassVal[ClassVal == label]) / Tot_Instance
        entropy_val -= p * math.log2(p)
```

```
    return entropy_val

def IG(TableOfData, atriButes):

    Tot_Instance = len(TableOfData)
    atriButes_entropy = 0

    for value in TableOfData[atriButes].unique():
        subset = TableOfData[TableOfData[atriButes] == value]
        subset_entropy = entropy(subset) * len(subset) / Tot_Instance
        atriButes_entropy += subset_entropy

    return entropy(TableOfData) - atriButes_entropy

def BuildDT(TableOfData, atriButess):

    if len(TableOfData['Play?'].unique()) == 1:
        return TableOfData['Play?'].iloc[0]

    if len(atriButess) == 0:
        return TableOfData['Play?'].value_counts().idxmax()

    max_gain = -1
    best_atriButes = None
    for atriButes in atriButess:
        gain = IG(TableOfData, atriButes)
        if gain > max_gain:
            max_gain = gain
            best_atriButes = atriButes

    tree = {best_atriButes: {}}
    remaining_atriButess = [attr for attr in atriButess if attr != best_atriButes]

    for value in TableOfData[best_atriButes].unique():
        subset = TableOfData[TableOfData[best_atriButes] == value]
```

```
        subtree = BuildDT(subset, remaining_atriButess)
        tree[best_atriButes][value] = subtree

    return tree

import pprint
atriButess = ['Outlook', 'Temperature', 'Humidity', 'Windy']

decision_tree = BuildDT(Data__Frame, atriButess)

pprint.pprint((decision_tree))

def predict(instance, tree):

    atriButes = next(iter(tree))
    value = instance[atriButes]
    subtree = tree[atriButes][value]

    if isinstance(subtree, dict):
        return predict(instance, subtree)
    else:
        return subtree

Data__Frame_= pd.read_csv('/content/Test TableOfData.csv')
Data__Frame_

Data__Frame_['Humidity'] = Data__Frame_['Humidity'].apply(lambda
    x: 'High' if x > 50 else 'Low')
Data__Frame_

predictions = []
for i in range(len(Data__Frame_)):
    instance = Data__Frame_.iloc[i]
    prediction = predict(instance, decision_tree)
    predictions.append(prediction)

Data__Frame_['Play?'] = predictions

Data__Frame_
```

```python

from sklearn.metrics import accuracy_score, classification_report, confusion_matrix


accuracy_scratch = accuracy_score(Data__Frame_['Play?'], Data__Frame_['Play?'])  # Use the correct column name here

confusion_mat_scratch = confusion_matrix(Data__Frame_['Play?'], Data__Frame_['Play?'])  # Use the correct column name here
class_report_scratch = classification_report(Data__Frame_['Play?'], Data__Frame_['Play?'], target_names=["No", "Yes"])  # Use the correct column name here

print("Accuracy (From Scratch):", accuracy_scratch)
print("\nConfusion Matrix (From Scratch):")
print(confusion_mat_scratch)
print("\nClassification Report (From Scratch):")
print(class_report_scratch)
```

Code 5: Artifical Neural Networks

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_csv("/content/breast_cancer_dataset.csv")
df

X = df.drop(["diagnosis", "id"], axis=1)
y = df['diagnosis']
X.head()

def OneHotsEncoder(labels):
    num_classes = len(labels.unique())
    one_hot_labels = pd.get_dummies(labels, columns=labels.name, drop_first=False)
    return one_hot_labels

def convert_labels_to_numeric(labels):
```

```
    return labels.map({'B': 0, 'M': 1})

def preprocess_cancer_data(X, y):
    y_numeric = OneHotsEncoder(y)

    X_normalized = (X - np.mean(X, axis=0)) / np.std(X, axis=0)

    return X_normalized, y_numeric

X, y = preprocess_cancer_data(X, y)
print(X)
print(y)

X = X.values
y = y.values
print(type(X))
print(type(y))

def split_data(X, y, test_size=0.2, random_state=None):
    return train_test_split(X, y, test_size=test_size,
   random_state=random_state)

TrainData_X, TestData_x, TrainData_y, TestData_y = split_data(X,
   y, test_size=0.2, random_state=42)


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

def SigmoidDerivatives(x):
    return x * (1 - x)

def initialize_parameters(Input_Sz, hidden_Sz, Output_Sz):
    hd_wght = np.random.randn(Input_Sz, hidden_Sz)
    hd_bias = np.zeros((1, hidden_Sz))
    outputWeight = np.random.randn(hidden_Sz, Output_Sz)
    outputBiase = np.zeros((1, Output_Sz))

    return hd_wght, hd_bias, outputWeight, outputBiase
```

```python

def forward_propagation(X, hd_wght, hd_bias, outputWeight, outputBiase):
    HiddenLayerInput = np.dot(X, hd_wght) + hd_bias
    HiddenLayerOutput = sigmoid(HiddenLayerInput)
    output_layer_input = np.dot(HiddenLayerOutput, outputWeight) + outputBiase
    output_layer_output = sigmoid(output_layer_input)

    return HiddenLayerOutput, output_layer_output

def backpropagation(X, y, HiddenLayerOutput, output_layer_output, hd_wght, hd_bias, outputWeight, outputBiase, LearningRate):
    OutError = y - output_layer_output
    DelOut = OutError * SigmoidDerivatives(output_layer_output)

    HiddenError = np.dot(DelOut, outputWeight.T)
    HiddenDelta = HiddenError * SigmoidDerivatives(HiddenLayerOutput)

    outputWeight += np.dot(HiddenLayerOutput.T, DelOut) * LearningRate
    outputBiase += np.sum(DelOut, axis=0, keepdims=True) * LearningRate

    hd_wght += np.dot(X.T, HiddenDelta) * LearningRate
    hd_bias += np.sum(HiddenDelta, axis=0, keepdims=True) * LearningRate

    return hd_wght, hd_bias, outputWeight, outputBiase

def train(X, y, hidden_Sz, Output_Sz, LearningRate, num_epochs):
    Input_Sz = X.shape[1]
    hd_wght, hd_bias, outputWeight, outputBiase = initialize_parameters(Input_Sz, hidden_Sz, Output_Sz)

    for epoch in range(num_epochs):
```

```
        HiddenLayerOutput, output_layer_output = forward_propagation(X, hd_wght, hd_bias, outputWeight, outputBiase)
        hd_wght, hd_bias, outputWeight, outputBiase = backpropagation(X, y, HiddenLayerOutput, output_layer_output, hd_wght, hd_bias, outputWeight, outputBiase, LearningRate)

    return hd_wght, hd_bias, outputWeight, outputBiase

def calculate_accuracy(y_true, y_pred):
    RightPredictions = np.sum(y_true == y_pred)
    TotalPredictions = len(y_true)
    accuracy = RightPredictions / TotalPredictions
    return accuracy

def predict(X, hd_wght, hd_bias, outputWeight, outputBiase):
    _, output_layer_output = forward_propagation(X, hd_wght, hd_bias, outputWeight, outputBiase)
    y_pred = np.argmax(output_layer_output, axis=1)
    return y_pred

def evaluate_model(TestData_x, TestData_y, hd_wght, hd_bias, outputWeight, outputBiase):
    y_pred = predict(TestData_x, hd_wght, hd_bias, outputWeight, outputBiase)
    accuracy = calculate_accuracy(np.argmax(TestData_y, axis=1), y_pred)
    return accuracy

LearningRate = 0.01
num_epochs = 1000
hidden_Sz = 10
Output_Sz = 2
hd_wght, hd_bias, outputWeight, outputBiase = train(TrainData_X, TrainData_y, hidden_Sz, Output_Sz, LearningRate, num_epochs)
test_accuracy = evaluate_model(TestData_x, TestData_y, hd_wght, hd_bias, outputWeight, outputBiase)
print("Test Accuracy:", test_accuracy)
```

```

def predict_single_instance(instance, hd_wght, hd_bias, outputWeight, outputBiase):
    _, output_layer_output = forward_propagation(instance, hd_wght, hd_bias, outputWeight, outputBiase)
    predicted_class = np.argmax(output_layer_output)
    return predicted_class

user_instance = TestData_x[15]
predicted_class = predict_single_instance(user_instance, hd_wght, hd_bias, outputWeight, outputBiase)
if predicted_class == 0:
    print("Predicted Class: Benign")
else:
    print("Predicted Class: Malignant")

from sklearn.metrics import accuracy_score, confusion_matrix, classification_report


y_pred = predict(TestData_x, hd_wght, hd_bias, outputWeight, outputBiase)

accuracy = accuracy_score(np.argmax(TestData_y, axis=1), y_pred)
print("Accuracy:", accuracy)


conf_matrix = confusion_matrix(np.argmax(TestData_y, axis=1), y_pred)
print("confusion matrix:")
print(conf_matrix)

class_report = classification_report(np.argmax(TestData_y, axis=1), y_pred, target_names=["Benign", "Malignant"])
print("Classification Report:")
print(class_report)

import seaborn as sns
import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(8, 6))
sns.set(font_scale=1.2)
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues",
    xticklabels=["Benign", "Malignant"], yticklabels=["Benign", "
    Malignant"])
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix")
plt.show()
```

References

[1] I. Sarker, 'Machine learning: Algorithms, real-world applications and research directions,' *SN Computer Science*, vol. 2, Mar. 2021. DOI: 10.1007/s42979-021-00592-x (cit. on p. 1).

[2] J. Han, M. Kamber and J. Pei, *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2006 (cit. on p. 3).

[3] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, 2019 (cit. on p. 10).

[4] I. Ahmed, *8k+ property listing data in bangladesh*, Feb. 2023. [Online]. Available: https://www.kaggle.com/datasets/ijajdatanerd/property-listing-data-in-bangladesh (cit. on p. 10).

[5] I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*. MIT Press, 2016 (cit. on p. 16).

[6] S. Alqethami, B. Almutanni and M. AlGhamdi, 'Fraud detection in e-commerce,' *International Journal of Computer Science & Network Security*, vol. 21, no. 6, pp. 312–318, 2021 (cit. on p. 24).

[7] U. M. Learning, *Breast cancer wisconsin (diagnostic) data set*, Sep. 2016. [Online]. Available: https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data (cit. on p. 25).