Practical 1 - Breadth First Search & Iterative Depth First Search

- Implement the Breadth First Search algorithm to solve a given problem

```
from collections import deque
class Graph:
    def __init__(self):
        self.adj_list = {}
    def add_vertex(self,vertex):
        if vertex not in self.adj_list.keys():
            self.adj_list[vertex] = []
    def add_edge(self,v1,v2):
        if v1 in self.adj_list and v2 in self.adj_list:
            self.adj_list[v1].append(v2)
            self.adj_list[v1].append(v1)

    def dfs(self, start_vertex):
        visited = set()
        traversal_order =[]

        def dfs_helper(vertex):
            visited.add(vertex)
            traversal_order.append(vertex)
            for neighbor in self.adj_list[vertex]:
                if neighbor not in visited:
                    dfs_helper(neighbor)
        dfs_helper(start_vertex)
        return traversal_order
```

- Implement the Iterative Depth First Search algorithm to solve the same problem.

```
[12] #example usaage:
     g = Graph()
     g.add_vertex('A')
     g.add_vertex('B')
     g.add_vertex('C')
     g.add_vertex('D')
     g.add_vertex('E')
     g.add_edge('A','B')
     g.add_edge('A','C')
     g.add_edge('B','D')
     g.add_edge('C','E')
```

- Compare the performance and efficiency of both algorithms.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
print(g.dfs('A'))
```

OUTPUT –

```
['A', 'B', 'D', 'C', 'E']
```

Practical 2 - Search and Recursive Best-First Search

- Implement the A Search algorithm for solving a pathfinding problem.

```
[4] from collections import deque

    class Graph:
      def __init__(self):
        self.adj_list = {}

      def add_vertex(self, vertex):
        if vertex not in self.adj_list:
          self.adj_list[vertex] = []

      def add_edges(self, vertex1, vertex2):
        if vertex1 in self.adj_list and vertex2 in self.adj_list:
          self.adj_list[vertex1].append(vertex2)
          self.adj_list[vertex2].append(vertex1)

      def bfs(self, start_vertex):
        visited = set()
        traversal_order = []
        queue = deque()

        queue.append(start_vertex)
        visited.add(start_vertex)

        while queue:
          vertex = queue.popleft()
          traversal_order.append(vertex)

          for neighbor in self.adj_list[vertex]:
            if neighbor not in visited:
              queue.append(neighbor)
              visited.add(neighbor)

        return traversal_order
```

- Implement the Recursive Best-First Search algorithm for the same problem.

```
#Example usage:
g = Graph()
g.add_vertex('A')
g.add_vertex('B')
g.add_vertex('C')
g.add_vertex('D')
g.add_vertex('E')

g.add_edges('A', 'B')
g.add_edges('A', 'C')
g.add_edges('B', 'D')
g.add_edges('C', 'E')

print(g.bfs('A'))  #OUTPUT ['A', 'B', 'C', 'D', 'E']
```

OUTPUT –

```
['A', 'B', 'C', 'D', 'E']
```

Practical 3 - Decision Tree Learning

- Implement the Decision Tree Learning algorithm to build a decision tree for a given dataset.

```
#prompt: Implement A* search algorithm for romanian map problem.
from collections import deque
import heapq
#define the  romanian map problem

romania_map = {
    'Arad': {'Zerind': 75, 'Sibiu': 140, 'Timisoara': 118},
    'Zerind': {'Arad': 75, 'Oradea': 71},
    'Oradea': {'Zerind': 71, 'Sibiu': 151},
    'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},
    'Timisoara': {'Arad': 118, 'Lugoj': 111},
    'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
    'Mehadia': {'Lugoj': 70, 'Drobeta': 75},
    'Drobeta': {'Mehadia': 75, 'Craiova': 120},
    'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},
    'Rimnicu Vilcea': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},
    'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
    'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},
    'Bucharest': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},
    'Giurgiu': {'Bucharest': 90},
    'Urziceni': {'Bucharest': 85, 'Vaslui': 142, 'Hirsova': 98},
    'Hirsova': {'Urziceni': 98, 'Eforie': 86},
    'Eforie': {'Hirsova': 86},
    'Vaslui': {'Iasi': 92, 'Urziceni': 142},
    'Iasi': {'Vaslui': 92, 'Neamt': 87},
    'Neamt': {'Iasi': 87}

}
```

- Evaluate the accuracy and effectiveness of the decision tree on test data

```
heuristic_values = {
    'Arad': 336,
    'Zerind': 0,
    'Oradea': 380,
    'Sibiu': 2,
    'Timisoara': 329,
    'Lugoj': 244,
    'Mehadia': 241,
    'Drobeta': 2,
    'Craiova': 160,
    'Rimnicu Vilcea': 193,
    'Fagaras': 1,
    'Pitesti': 98,
    'Bucharest': 0,
    'Giurgiu': 77,
    'Urziceni': 80,
    'Hirsova': 151,
    'Eforie': 161,
    'Vaslui': 199,
    'Iasi': 226,
    'Neamt': 234,
    'Oradea': 380,
    'Zerind': 374,
    'Arad': 366,
    'Timisoara': 3,
    'Lugoj': 242
}
```

- Visualize and interpret the generated decision tree

```python
#prompt: Implement A* search algorithm for romanian map problem.
from collections import deque
import heapq
#define the  romanian map problem

# ... (rest of the code is the same)

def astar_search(graph, start, goal):
    open_list = []
    closed_list = set()
    heapq.heappush(open_list, (heuristic_values[start], start, [start], 0))  # Include initial cost
    while open_list:
        _, current_node, path, cost_so_far = heapq.heappop(open_list)  # Unpack cost_so_far
        if current_node == goal:
            return path, cost_so_far  # Return both path and cost

        closed_list.add(current_node)
        for neighbor, cost in graph[current_node].items():
            if neighbor not in closed_list:
                new_cost = heuristic_values[neighbor]
                new_path = path + [neighbor]
                new_cost_so_far = cost_so_far + cost  # Calculate accumulated cost
                f_value = new_cost + new_cost_so_far # Calculate f_value
                heapq.heappush(open_list, (f_value, neighbor, new_path, new_cost_so_far))

    return None, None # Return None, None if no path is found

path, cost =  astar_search(romania_map, 'Arad', 'Bucharest')

if path:
    print("Path found:", path)
    print("Total cost:", cost)
else:
    print("No path found.")
```

OUTPUT -

```
Path found: ['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti', 'Bucharest']
Total cost: 418
```

Practical 4 - Feed Forward Backpropagation Neural Network

- Implement the Feed Forward Backpropagation algorithm to train a neural network

```
# prompt: Implement recursive best-first search algorithm for Romanian map problem.

from collections import deque
import heapq

# Define the Romanian map problem
romania_map = {
    'Arad': {'Zerind': 75, 'Sibiu': 140, 'Timisoara': 118},
    'Zerind': {'Arad': 75, 'Oradea': 71},
    'Oradea': {'Zerind': 71, 'Sibiu': 151},
    'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},
    'Timisoara': {'Arad': 118, 'Lugoj': 111},
    'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
    'Mehadia': {'Lugoj': 70, 'Drobeta': 75},
    'Drobeta': {'Mehadia': 75, 'Craiova': 120},
    'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},
    'Rimnicu Vilcea': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},
    'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
    'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},
    'Bucharest': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},
    'Giurgiu': {'Bucharest': 90},
    'Urziceni': {'Bucharest': 85, 'Hirsova': 98, 'Vaslui': 142},
    'Hirsova': {'Urziceni': 98, 'Eforie': 86},
    'Eforie': {'Hirsova': 86},
    'Vaslui': {'Urziceni': 142, 'Iasi': 92},
    'Iasi': {'Vaslui': 92, 'Neamt': 87},
    'Neamt': {'Iasi': 87}
}
```

- Use a given dataset to train the neural network for a specific task

```
# Define the heuristic function (straight-line distance to Bucharest)
heuristic = {
    'Arad': 366,
    'Bucharest': 0,
    'Craiova': 160,
    'Drobeta': 242,
    'Eforie': 161,
    'Fagaras': 176,
    'Giurgiu': 77,
    'Hirsova': 151,
    'Iasi': 226,
    'Lugoj': 244,
    'Mehadia': 241,
    'Neamt': 234,
    'Oradea': 380,
    'Pitesti': 100,
    'Rimnicu Vilcea': 193,
    'Sibiu': 253,
    'Timisoara': 329,
    'Urziceni': 80,
    'Vaslui': 199,
    'Zerind': 374
}
```

- Evaluate the performance of the trained network on test data

```
def rbfs(graph, start, goal, f_limit):
    def expand_node(node, f_limit):
        if node == goal:
            return node, 0, True

        successors = []
        for neighbor, cost in graph[node].items():
            g = cost
            h = heuristic[neighbor]
            f = g + h
            successors.append((f, neighbor, g))

        if not successors:
            return None, float('inf'), False

        successors.sort()

        best_f = successors[0][0]
        best_node = successors[0][1]

        if best_f > f_limit:
            return None, best_f, False

        alternative_f = float('inf')
        if len(successors) > 1:
            alternative_f = successors[1][0]

        result, best_f, success = expand_node(best_node, min(f_limit, alternative_f))

        if success:
            return result, best_f, True

        return None, best_f, False

    result, _, _ = expand_node(start, f_limit)
    return result
```

```
# Find the path from Arad to Bucharest
result = rbfs(romania_map, 'Arad', 'Bucharest', float('inf'))

if result:
    print("Path found (RBFS):", result)
else:
    print("No path found (RBFS).")
```

OUTPUT –

```
Path found (RBFS): Bucharest
```

Practical 5 - Support Vector Machines (SVM)

- Implement the SVM algorithm for binary classification
- Train an SVM model using a given dataset and optimize its parameters.
- Evaluate the performance of the SVM model on test data and analyze the results

Suggested code may be subject to a license | Ishita777/Logistic-Regression | AhmedRaja1/Python-Beginner-s-Starter-Kit | | Fusica/Jupyter-Notebook

```python
# Practical-no: 5

import pandas as pd
from sklearn.tree import DecisionTreeClassifier, export_graphviz
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Sample data
data = {
    'Alt': ['Yes', 'Yes', 'No', 'Yes'],
    'Bar': ['No', 'Yes', 'No', 'No'],
    'Fri/Sat': ['Yes', 'Yes', 'No', 'Yes'],
    'Hungry': ['Yes', 'No', 'Yes', 'No'],
    'Patrons': ['Some', 'Full', 'Some', 'Full'],
    'Price': ['$$$$', '$$ - $$$', '$', '$$ - $$$'],
    'Raining': ['No', 'No', 'No', 'Yes'],
    'Reservation': ['No', 'Yes', 'No', 'Yes'],
    'Type': ['French', 'Thai', 'Burger', 'Italian'],
    'WaitEstimate': ['0 - 10', '10 - 30', '0 - 10', '30 - 60'],
    'WillWait': ['No', 'Yes', 'Yes', 'No'],
}

# Create DataFrame
df = pd.DataFrame(data)

# Encode categorical features
for column in df.columns:
    if df[column].dtype == 'object':  # Check if column is categorical
        le = LabelEncoder()
        df[column] = le.fit_transform(df[column])

# Split data into features (X) and target (y)
X = df.drop('WillWait', axis=1)  # Features (all columns except 'WillWait')
y = df['WillWait']  # Target variable ('WillWait')

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

# Create and train the Decision Tree Classifier
model = DecisionTreeClassifier(random_state=42)
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy:.2f}")
print("Confusion Matrix:\n", conf_matrix)
print("Classification Report:\n", class_report)
```

OUTPUT -

```
Accuracy: 0.00
Confusion Matrix:
 [[0 0]
 [1 0]]
Classification Report:
               precision    recall  f1-score   support

           0       0.00      0.00      0.00       0.0
           1       0.00      0.00      0.00       1.0

    accuracy                           0.00       1.0
   macro avg       0.00      0.00      0.00       1.0
weighted avg       0.00      0.00      0.00       1.0
```

Practical 6 - Adaboost Ensemble Learning

- Implement the Adaboost algorithm to create an ensemble of weak classifiers.

```
[2] import numpy as np
    import matplotlib.pyplot as plt
    from sklearn.datasets import make_classification
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.ensemble import AdaBoostClassifier
    from sklearn.metrics import accuracy_score
    from sklearn.model_selection import train_test_split

    # Generate a synthetic dataset
    X, y = make_classification(n_samples=1000, n_features=20, n_informative=10, n_redundant=5, random_state=42)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

- Train the ensemble model on a given dataset and evaluate its performance.

```
# Train individual weak classifiers (stumps)
weak_classifiers = []
n_classifiers = 5

for _ in range(n_classifiers):
    clf = DecisionTreeClassifier(max_depth=1)  # Stump
    clf.fit(X_train, y_train)
    weak_classifiers.append(clf)

print("Accuracy of individual weak classifiers:")
for i, clf in enumerate(weak_classifiers):
    y_pred = clf.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    print(f"Weak Classifier {i + 1}: {acc:.4f}")

# Train AdaBoost model
ada_boost = AdaBoostClassifier(estimator=DecisionTreeClassifier(max_depth=1), n_estimators=n_classifiers)
ada_boost.fit(X_train, y_train)
```

OUTPUT –

```
Accuracy of individual weak classifiers:
Weak Classifier 1: 0.6967
Weak Classifier 2: 0.6967
Weak Classifier 3: 0.6967
Weak Classifier 4: 0.6967
Weak Classifier 5: 0.6967
/usr/local/lib/python3.10/dist-packages/sklearn/ensemble/_weight_boosting.py:527:
  warnings.warn(
AdaBoostClassifier
AdaBoostClassifier(estimator=DecisionTreeClassifier(max_depth=1),
                   n_estimators=5)
estimator: DecisionTreeClassifier
DecisionTreeClassifier(max_depth=1)
DecisionTreeClassifier
DecisionTreeClassifier(max_depth=1)
```

- Compare the results with individual weak classifiers.

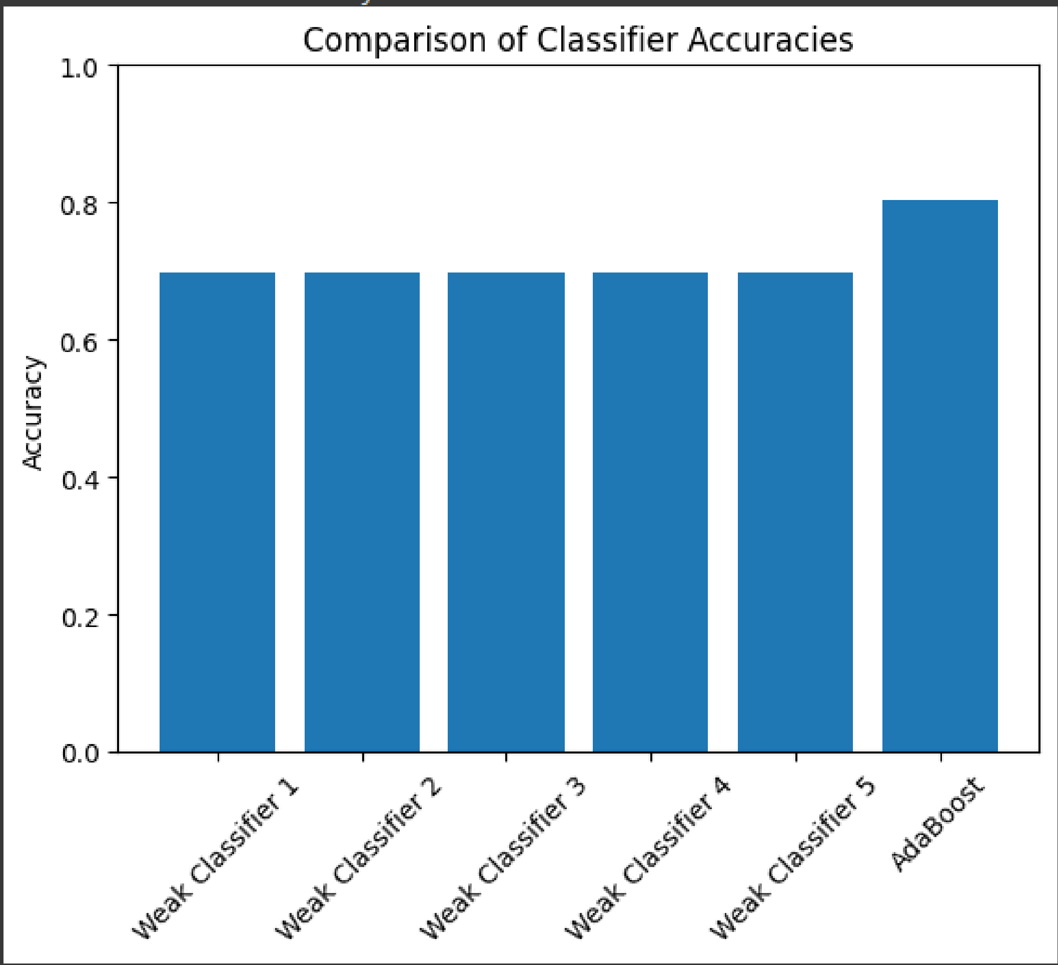
```
# Evaluate AdaBoost model
y_pred_ada = ada_boost.predict(X_test)
ada_accuracy = accuracy_score(y_test, y_pred_ada)
print(f"AdaBoost Ensemble Accuracy: {ada_accuracy:.4f}")


# Visualize the results
accuracies = [accuracy_score(y_test, clf.predict(X_test)) for clf in weak_classifiers]
accuracies.append(ada_accuracy)

labels = [f"Weak Classifier {i + 1}" for i in range(n_classifiers)] + ['AdaBoost']
plt.bar(labels, accuracies)
plt.ylabel('Accuracy')
plt.title('Comparison of Classifier Accuracies')
plt.xticks(rotation=45)
plt.ylim(0, 1)
plt.show()
```

OUTPUT –

```
AdaBoost Ensemble Accuracy: 0.8033
```



Practical 7 - Naive Bayex Classifier

- Implement the Naive Bayes algorithm for classification.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

# Load the dataset.
# Replace 'your_dataset.csv' with the actual path to your dataset.
data = pd.read_csv('/content/sample_data/dataset.csv')


# Split the dataset into training and testing sets.
X = data.drop('target_variable', axis=1)  # Features
y = data['target_variable']  # Target variable
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

# Create a Gaussian Naive Bayes classifier.
classifier = GaussianNB()

# Train the classifier.
classifier.fit(X_train, y_train)

# Make predictions on the testing set.
y_pred = classifier.predict(X_test)

# Evaluate the classifier's performance.
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")
```

OUTPUT –

```
Accuracy: 1.0
```

- Train a Naive Bayes model using a given dataset and probabilities calculate class

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB

# Load the dataset.
# Replace 'your_dataset.csv' with the actual path to your dataset.
data = pd.read_csv('/content/sample_data/dataset.csv')

# Split the dataset into training and testing sets.
X = data.drop('target_variable', axis=1)  # Features
y = data['target_variable']  # Target variable
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

# Create a Gaussian Naive Bayes classifier.
classifier = GaussianNB()

# Train the classifier.
classifier.fit(X_train, y_train)

# Calculate class probabilities.
class_probabilities = classifier.class_prior_

# Print the class probabilities.
for i, probability in enumerate(class_probabilities):
    print(f"Probability of class {classifier.classes_[i]}: {probability}")
```

OUTPUT -

```
Probability of class 0: 0.42857142857142855
Probability of class 1: 0.5714285714285714
```

- Evaluate the accuracy of the model on test data and analyze the results

```python
import pandas as pd
from sklearn. (module) naive_bayes train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Load the dataset.
data = pd.read_csv('/content/sample_data/dataset.csv')  # Replace 'your_dataset.csv'

# Split the dataset.
X = data.drop('target_variable', axis=1)  # Replace 'target_variable'
y = data['target_variable']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

# Create and train the model.
classifier = GaussianNB()
classifier.fit(X_train, y_train)

# Make predictions on the test data.
y_pred = classifier.predict(X_test)

# Evaluate the model.
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")

# Analyze the results.
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

OUTPUT –

```
Accuracy: 1.0

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         2
           1       1.00      1.00      1.00         1

    accuracy                           1.00         3
   macro avg       1.00      1.00      1.00         3
weighted avg       1.00      1.00      1.00         3


Confusion Matrix:
[[2 0]
 [0 1]]
```

Practical 8 - K-Nearest Neighbors (K-NN)

- Implement the K-NN algorithm for classification or regression.

```
#AI_Paractical_No_8

import pandas as pd
from sklearn.datasets import load_iris, make_regression
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.metrics import accuracy_score, mean_squared_error

data_classification = pd.read_csv('/content/sample_data/data.csv')

# Prepare the classification data
X_class = data_classification[['feature1', 'feature2']]
y_class = data_classification['target']
X_train_class, X_test_class, y_train_class, y_test_class = train_test_split(X_class, y_class, test_size=0.25, random_state=42)

# Implement K-NN for classification
knn_classifier = KNeighborsClassifier(n_neighbors=5)
knn_classifier.fit(X_train_class, y_train_class)

# Predict on the test data for classification
y_pred_class = knn_classifier.predict(X_test_class)

# Evaluate the classification model
accuracy = accuracy_score(y_test_class, y_pred_class)
print(f"Classification Accuracy: {accuracy:.2f}")

# K-NN for Regression

# Create synthetic regression data
X_reg, y_reg = make_regression(n_samples=100, n_features=1, noise=10, random_state=42)

# Split the dataset into training and testing sets for regression
X_train_reg, X_test_reg, y_train_reg, y_test_reg = train_test_split(X_reg, y_reg, test_size=0.25, random_state=42)

# Implement K-NN for regression
knn_regressor = KNeighborsRegressor(n_neighbors=5)
knn_regressor.fit(X_train_reg, y_train_reg)

# Predict on the test data for regression
y_pred_reg = knn_regressor.predict(X_test_reg)

# Evaluate the regression model
mse = mean_squared_error(y_test_reg, y_pred_reg)
print(f"Mean Squared Error: {mse:.2f}")
```

OUTPUT –

```
Classification Accuracy: 0.00
Mean Squared Error: 106.07
```

Practical 9 - Association Rule Mining

- Implement the Association Rule Mining algorithm (e.g., Apriori) to find frequent itemsets.

```python
#practical-9
!pip install mlxtend==0.21.0
import pandas as pd
from mlxtend.frequent_patterns import apriori, association_rules
from mlxtend.preprocessing import TransactionEncoder

# Sample transaction data
data = [
    ['Milk', 'Bread', 'Butter'],
    ['Bread', 'Butter', 'Eggs'],
    ['Milk', 'Bread', 'Eggs'],
    ['Bread', 'Butter'],
    ['Milk', 'Eggs']
]

# 1. Data Preprocessing:
# Convert the transaction data into a one-hot encoded format using TransactionEncoder
te = TransactionEncoder()
te_data = te.fit(data).transform(data)
df = pd.DataFrame(te_data, columns=te.columns_)

# 2. Frequent Itemset Mining using Apriori:
# Set the minimum support threshold (e.g., 0.5 for 50% support)
min_support = 0.5

# Apply the apriori algorithm to find frequent itemsets
frequent_itemsets = apriori(df, min_support=min_support, use_colnames=True)

# Print the frequent itemsets
print("Frequent Itemsets:")
print(frequent_itemsets)

# 3. Association Rule Generation:
# Generate association rules from the frequent itemsets
# using a metric (e.g., confidence) and a minimum threshold (e.g., 0.7)
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.7)

# Print the association rules
print("\nAssociation Rules:")
print(rules[['antecedents', 'consequents', 'support', 'confidence']])
```

OUTPUT –

```
Frequent Itemsets:
   support         itemsets
0      0.8          (Bread)
1      0.6         (Butter)
2      0.6           (Eggs)
3      0.6           (Milk)
4      0.6  (Bread, Butter)

Association Rules:
  antecedents consequents  support  confidence
0     (Bread)    (Butter)      0.6        0.75
1    (Butter)     (Bread)      0.6        1.00
```

- Generate association rules from the frequent itemsets and calculate their support and confidence

```
#practical-9.2
!pip install mlxtend==0.21.0
import pandas as pd
from mlxtend.frequent_patterns import apriori, association_rules
from mlxtend.preprocessing import TransactionEncoder
data = [
    ['Milk', 'Bread', 'Butter'],
    ['Bread', 'Butter', 'Eggs'],
    ['Milk', 'Bread', 'Eggs'],
    ['Bread', 'Butter'],
    ['Milk', 'Eggs']
]
te = TransactionEncoder()
te_data = te.fit(data).transform(data)
df = pd.DataFrame(te_data, columns=te.columns_)
min_support = 0.5
frequent_itemsets = apriori(df, min_support=min_support, use_colnames=True)
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.7)
# Print the rules with support and confidence
print(rules[['antecedents', 'consequents', 'support', 'confidence']])
rules.to_csv('association_rules_with_support_confidence.csv', index=False)
```

- Interpret and analyze the discovered association rules.

```
# practical-9.3
!pip install mlxtend==0.21.0
import pandas as pd
from mlxtend.frequent_patterns import apriori, association_rules
from mlxtend.preprocessing import TransactionEncoder

data = [
    ['Milk', 'Bread', 'Butter'],
    ['Bread', 'Butter', 'Eggs'],
    ['Milk', 'Bread', 'Eggs'],
    ['Bread', 'Butter'],
    ['Milk', 'Eggs']
]
te = TransactionEncoder()
te_data = te.fit(data).transform(data)
df = pd.DataFrame(te_data, columns=te.columns_)
min_support = 0.5
frequent_itemsets = apriori(df, min_support=min_support, use_colnames=True)
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.7)
```

OUTPUT –

```
  antecedents consequents  support  confidence
0      (Bread)    (Butter)      0.6        0.75
1     (Butter)     (Bread)      0.6        1.00
```

Practical 10 - Demo of OpenAI/TensorFlow Tools

- Explore and experiment with OpenAI or TensorFlow tools and libraries.

```
!pip install tensorflow==2.13.0
```

Show hidden output

```
import tensorflow as tf

# Example: Creating a simple TensorFlow model
model = tf.keras.Sequential([
  tf.keras.layers.Dense(10, activation='relu'),
  tf.keras.layers.Dense(1)
])
```

- Perform a demonstration or mini-project showcasing the capabilities of the tools

```
import tensorflow as tf
import tensorflow_datasets as tfds

dataset, info = tfds.load('imdb_reviews', with_info=True, as_supervised=True)
train_dataset, test_dataset = dataset['train'], dataset['test']

# Define a text encoder for tokenization
encoder = tf.keras.layers.TextVectorization(max_tokens=10000) # Create a TextVectorization layer for tokenization
encoder.adapt(train_dataset.map(lambda text, label: text)) # Fit the encoder on the text data in the training set

# Update the model with the encoder layer
model = tf.keras.Sequential([
    encoder,  # Add the encoder as the first layer
    tf.keras.layers.Embedding(10000, 16),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(16, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

model.fit(train_dataset.batch(32), epochs=5)
results = model.evaluate(test_dataset.batch(32))
print(results)
```

- Discuss and present the findings and potential applications

- **OUTPUT –**

```
Downloading and preparing dataset 80.23 MiB (download: 80.23 MiB, generated: Unknown size, total: 80.23 MiB) to /root/tensorflow_datasets/imdb_reviews/plain_text/1.0.0...
Dl Completed...: 100% 1/1 [00:44<00:00, 44.79s/ url]
Dl Size...: 100% 80/80 [00:44<00:00, 1.53 MiB/s]
Dataset imdb_reviews downloaded and prepared to /root/tensorflow_datasets/imdb_reviews/plain_text/1.0.0. Subsequent calls will reuse this data.
Epoch 1/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 9s 9ms/step - accuracy: 0.5100 - loss: 0.6917
Epoch 2/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 9ms/step - accuracy: 0.6880 - loss: 0.6109
Epoch 3/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 9s 12ms/step - accuracy: 0.8035 - loss: 0.4478
Epoch 4/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 8s 10ms/step - accuracy: 0.8439 - loss: 0.3682
Epoch 5/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 10s 10ms/step - accuracy: 0.8642 - loss: 0.3254
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 7ms/step - accuracy: 0.8723 - loss: 0.3285
[0.3246074616909027, 0.8722400069236755]
```