Experiment-1

Objective: Solve the Tic-Tac-Toe problem using the Depth First Search technique.

Source Code:

```
# Set up the game board as a list
board = ["-", "-", "-",
         "-", "-", "-",
         "-", "-", "-"]

# Define a function to print the game board
def print_board():
    print(board[0] + " | " + board[1] + " | " + board[2])
    print(board[3] + " | " + board[4] + " | " + board[5])
    print(board[6] + " | " + board[7] + " | " + board[8])

# Define a function to handle a player's turn
def take_turn(player):
    print(player + "'s turn.")
    position = input("Choose a position from 1-9: ")
    while position not in ["1", "2", "3", "4", "5", "6", "7", "8", "9"]:
        position = input("Invalid input. Choose a position from 1-9: ")
    position = int(position) - 1
    while board[position] != "-":
        position = int(input("Position already taken. Choose a different position: ")) - 1
    board[position] = player
    print_board()
```

```
# Define a function to check if the game is over
def check_game_over():
    # Check for a win
    if (board[0] == board[1] == board[2] != "-") or \
       (board[3] == board[4] == board[5] != "-") or \
       (board[6] == board[7] == board[8] != "-") or \
       (board[0] == board[3] == board[6] != "-") or \
       (board[1] == board[4] == board[7] != "-") or \
       (board[2] == board[5] == board[8] != "-") or \
       (board[0] == board[4] == board[8] != "-") or \
       (board[2] == board[4] == board[6] != "-"):
        return "win"
    # Check for a tie
    elif "-" not in board:
        return "tie"
    # Game is not over
    else:
        return "play"

# Define the main game loop
def play_game():
    print_board()
    current_player = "X"
    game_over = False
    while not game_over:
        take_turn(current_player)
        game_result = check_game_over()
        if game_result == "win":
            print(current_player + " wins!")
```

```
            game_over = True
        elif game_result == "tie":
            print("It's a tie!")
            game_over = True
        else:
            # Switch to the other player
            current_player = "O" if current_player == "X" else "X"

# Start the game
play_game()
```

Result:

```
- | - | -
- | - | -
- | - | -
X's turn.
Choose a position from 1-9: 2
- | X | -
- | - | -
- | - | -
O's turn.
Choose a position from 1-9: 3
- | X | O
- | - | -
- | - | -
X's turn.
Choose a position from 1-9: 4
- | X | O
X | - | -
- | - | -
O's turn.
Choose a position from 1-9: 5
- | X | O
X | O | -
- | - | -
X's turn.
Choose a position from 1-9: 6
- | X | O
X | O | X
- | - | -
O's turn.
Choose a position from 1-9: 7
- | X | O
X | O | X
O | - | -
O wins!
```

Experiment-2

Objective: Show that the 8-puzzle states are divided into two disjoint sets, such that any state is reachable from any other state in the same set, while no state is reachable from any state in the other set.

Source Code:

```
# Python3 program to print the path from root
# node to destination node for N*N-1 puzzle
# algorithm using Branch and Bound
# The solution assumes that instance of
# puzzle is solvable

# Importing copy for deepcopy function
import copy

# Importing the heap functions from python
# library for Priority Queue
from heapq import heappush, heappop

# This variable can be changed to change
# the program from 8 puzzle(n=3) to 15
# puzzle(n=4) to 24 puzzle(n=5)...
n = 3

# bottom, left, top, right
row = [ 1, 0, -1, 0 ]
col = [ 0, -1, 0, 1 ]
```

```
# A class for Priority Queue
class priorityQueue:

    # Constructor to initialize a
    # Priority Queue
    def __init__(self):
        self.heap = []

    # Inserts a new key 'k'
    def push(self, k):
        heappush(self.heap, k)

    # Method to remove minimum element
    # from Priority Queue
    def pop(self):
        return heappop(self.heap)

    # Method to know if the Queue is empty
    def empty(self):
        if not self.heap:
            return True
        else:
            return False

# Node structure
class node:

    def __init__(self, parent, mat, empty_tile_pos,
                 cost, level):
```

```
        # Stores the parent node of the
        # current node helps in tracing
        # path when the answer is found
        self.parent = parent

        # Stores the matrix
        self.mat = mat

        # Stores the position at which the
        # empty space tile exists in the matrix
        self.empty_tile_pos = empty_tile_pos

        # Stores the number of misplaced tiles
        self.cost = cost

        # Stores the number of moves so far
        self.level = level

    # This method is defined so that the
    # priority queue is formed based on
    # the cost variable of the objects
    def __lt__(self, nxt):
        return self.cost < nxt.cost

# Function to calculate the number of
# misplaced tiles ie. number of non-blank
# tiles not in their goal position
def calculateCost(mat, final) -> int:
```

```
    count = 0
    for i in range(n):
        for j in range(n):
            if ((mat[i][j]) and
                (mat[i][j] != final[i][j])):
                count += 1

    return count

def newNode(mat, empty_tile_pos, new_empty_tile_pos,
            level, parent, final) -> node:

    # Copy data from parent matrix to current matrix
    new_mat = copy.deepcopy(mat)

    # Move tile by 1 position
    x1 = empty_tile_pos[0]
    y1 = empty_tile_pos[1]
    x2 = new_empty_tile_pos[0]
    y2 = new_empty_tile_pos[1]
    new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2],
new_mat[x1][y1]

    # Set number of misplaced tiles
    cost = calculateCost(new_mat, final)

    new_node = node(parent, new_mat, new_empty_tile_pos,
                    cost, level)
    return new_node
```

```
# Function to print the N x N matrix
def printMatrix(mat):

    for i in range(n):
        for j in range(n):
            print("%d " % (mat[i][j]), end = " ")

        print()

# Function to check if (x, y) is a valid
# matrix coordinate
def isSafe(x, y):

    return x >= 0 and x < n and y >= 0 and y < n

# Print path from root node to destination node
def printPath(root):

    if root == None:
        return

    printPath(root.parent)
    printMatrix(root.mat)
    print()

# Function to solve N*N - 1 puzzle algorithm
# using Branch and Bound. empty_tile_pos is
# the blank tile position in the initial state.
def solve(initial, empty_tile_pos, final):
```

```
# Create a priority queue to store live
# nodes of search tree
pq = priorityQueue()

# Create the root node
cost = calculateCost(initial, final)
root = node(None, initial,
            empty_tile_pos, cost, 0)

# Add root to list of live nodes
pq.push(root)

# Finds a live node with least cost,
# add its children to list of live
# nodes and finally deletes it from
# the list.
while not pq.empty():

    # Find a live node with least estimated
    # cost and delete it from the list of
    # live nodes
    minimum = pq.pop()

    # If minimum is the answer node
    if minimum.cost == 0:

        # Print the path from root to
        # destination;
        printPath(minimum)
        return
```

```
        # Generate all possible children
        for i in range(4):
            new_tile_pos = [
                minimum.empty_tile_pos[0] + row[i],
                minimum.empty_tile_pos[1] + col[i], ]

            if isSafe(new_tile_pos[0], new_tile_pos[1]):

                # Create a child node
                child = newNode(minimum.mat,
                                minimum.empty_tile_pos,
                                new_tile_pos,
                                minimum.level + 1,
                                minimum, final,)

                # Add child to list of live nodes
                pq.push(child)

# Driver Code

# Initial configuration
# Value 0 is used for empty space
initial = [ [ 1, 2, 3 ],
            [ 5, 6, 0 ],
            [ 7, 8, 4 ] ]

# Solvable Final configuration
# Value 0 is used for empty space
final = [ [ 1, 2, 3 ],
```

```
        [ 5, 8, 6 ],
        [ 0, 7, 4 ] ]

# Blank tile coordinates in
# initial configuration
empty_tile_pos = [ 1, 2 ]

# Function call to solve the puzzle
solve(initial, empty_tile_pos, final)
```

Result:

```
1 2 3
5 6 0
7 8 4
1 2 3
5 0 6
7 8 4
1 2 3
5 8 6
7 0 4
1 2 3
5 8 6
0 7 4
```

Experiment-3

Objective: To represent and evaluate different scenarios using predicate logic and knowledge rules.

Source Code:

```
from sympy import symbols
from sympy.logic.boolalg import Implies, And, Or, Not
from sympy.logic.inference import satisfiable

# Define predicates (functions returning boolean values)
Professor = symbols('Professor')
Teaches = symbols('Teaches')
Researches = symbols('Researches')
Student = symbols('Student')
Subject = symbols('Subject')

# Define individuals
Alice = symbols('Alice')
Bob = symbols('Bob')
Math = symbols('Math')
AI = symbols('AI')

# Define knowledge rules
knowledge_base = [
    Implies(Professor, Teaches),  # Professors teach
    Implies(Teaches, Subject),    # If someone teaches, there is a subject
```

```
    Implies(Researches, Professor), # If someone researches, they are a professor
    Implies(Student, Not(Teaches))  # Students do not teach
]

# Scenario: Alice is a professor and researches AI
facts = [Professor.subs(Professor, Alice), Researches.subs(Researches, Alice)]

# Check if Alice teaches
query = Teaches.subs(Teaches, Alice)
result = satisfiable(And(*knowledge_base, *facts, query))

print(f"Does Alice teach? {'Yes' if result else 'No'}")
```

Result:

```
Does Alice teach? Yes
```

Experiment-4

Objective: To apply the Find-S and Candidate Elimination algorithms to a concept learning task and compare their inductive biases and outputs.

Source Code:

```
import numpy as np

# Sample training data: (Sky, AirTemp, Humidity, Wind, Water,
Forecast, EnjoySport)
data = np.array([
    ['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same', 'Yes'],
    ['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same', 'Yes'],
    ['Rainy', 'Cold', 'High', 'Strong', 'Warm', 'Change', 'No'],
    ['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change', 'Yes']
])

# Extract features (X) and labels (Y)
X = data[:, :-1]  # All columns except the last one
Y = data[:, -1]   # Last column (labels)

# ---------------- FIND-S Algorithm ----------------
def find_s_algorithm(X, Y):
    specific_hypothesis = X[0].copy()  # Start with the first positive
example
    for i, row in enumerate(X[1:]):
        if Y[i+1] == 'Yes':  # Consider only positive examples
```

```
        for j in range(len(specific_hypothesis)):
            if row[j] != specific_hypothesis[j]:
                specific_hypothesis[j] = '?'  # Generalize when necessary
    return specific_hypothesis

# ---------------- CANDIDATE ELIMINATION Algorithm ----------------
def candidate_elimination_algorithm(X, Y):
    num_attributes = X.shape[1]
    specific_hypothesis = X[0].copy()  # Start with first positive example
    general_hypothesis = [['?'] * num_attributes]  # Most general
hypothesis

    for i, row in enumerate(X):
        if Y[i] == 'Yes':  # Positive example → specialize S
            for j in range(num_attributes):
                if specific_hypothesis[j] != row[j]:
                    specific_hypothesis[j] = '?'
            # Remove inconsistent general hypotheses
            general_hypothesis = [gh for gh in general_hypothesis if
any(gh[k] != specific_hypothesis[k] and gh[k] != '?' for k in
range(num_attributes))]

        elif Y[i] == 'No':  # Negative example → generalize G
            new_general_hypotheses = []
            for gh in general_hypothesis:
                for j in range(num_attributes):
                    if gh[j] == '?':  # Try specializing general hypothesis
                        if specific_hypothesis[j] != '?':
                            new_hypothesis = gh.copy()
                            new_hypothesis[j] = specific_hypothesis[j]
```

```
                new_general_hypotheses.append(new_hypothesis)
        general_hypothesis += new_general_hypotheses

    # Remove overly specific hypotheses
    general_hypothesis = [gh for gh in general_hypothesis if any(h != '?'
for h in gh)]
    return specific_hypothesis, general_hypothesis

# Run both algorithms
find_s_result = find_s_algorithm(X, Y)
specific_h, general_h = candidate_elimination_algorithm(X, Y)

# Display results
print("\nFind-S Hypothesis:", find_s_result)
print("\nCandidate Elimination Algorithm:")
print("Most Specific Hypothesis:", specific_h)
print("Most General Hypotheses:", general_h)
```

Result:

```
Find-S Hypothesis: ['Sunny' 'Warm' '?' 'Strong' '?' '?']

Candidate Elimination Algorithm:
Most Specific Hypothesis: ['Sunny' 'Warm' '?' 'Strong' '?' '?']
Most General Hypotheses: []
```

Experiment-5

Objective: To construct a decision tree using the ID3 algorithm on a simple classification dataset.

Source Code:

```
import numpy as np
import pandas as pd
from collections import Counter
import math

# Sample dataset: (Outlook, Temperature, Humidity, Wind, PlayTennis)
data = pd.DataFrame([
    ['Sunny', 'Hot', 'High', 'Weak', 'No'],
    ['Sunny', 'Hot', 'High', 'Strong', 'No'],
    ['Overcast', 'Hot', 'High', 'Weak', 'Yes'],
    ['Rain', 'Mild', 'High', 'Weak', 'Yes'],
    ['Rain', 'Cool', 'Normal', 'Weak', 'Yes'],
    ['Rain', 'Cool', 'Normal', 'Strong', 'No'],
    ['Overcast', 'Cool', 'Normal', 'Strong', 'Yes'],
    ['Sunny', 'Mild', 'High', 'Weak', 'No'],
    ['Sunny', 'Cool', 'Normal', 'Weak', 'Yes'],
    ['Rain', 'Mild', 'Normal', 'Weak', 'Yes'],
    ['Sunny', 'Mild', 'Normal', 'Strong', 'Yes'],
    ['Overcast', 'Mild', 'High', 'Strong', 'Yes'],
    ['Overcast', 'Hot', 'Normal', 'Weak', 'Yes'],
    ['Rain', 'Mild', 'High', 'Strong', 'No']
], columns=['Outlook', 'Temperature', 'Humidity', 'Wind', 'PlayTennis'])
```

```
# Function to calculate entropy
def entropy(target_col):
    counts = Counter(target_col)
    total = len(target_col)
    return -sum((count / total) * math.log2(count / total) for count in
counts.values())

# Function to calculate Information Gain
def info_gain(data, split_attribute, target_name):
    total_entropy = entropy(data[target_name])  # Parent entropy
    values = data[split_attribute].unique()  # Unique values of the
attribute
    weighted_entropy = sum(
        (len(subset := data[data[split_attribute] == val]) / len(data)) *
entropy(subset[target_name])
        for val in values
    )
    return total_entropy - weighted_entropy  # Information gain

# Function to build the decision tree
def id3(data, features, target):
    # Base cases
    if len(set(data[target])) == 1:  # Pure class
        return data[target].iloc[0]
    if not features:  # No more features to split on
        return data[target].mode()[0]

    # Select the best feature
    best_feature = max(features, key=lambda f: info_gain(data, f, target))
```

```
    # Create a subtree
    tree = {best_feature: {}}
    features = [f for f in features if f != best_feature]  # Remove used feature

    for value in data[best_feature].unique():  # For each unique value of the best feature
        subset = data[data[best_feature] == value]
        tree[best_feature][value] = id3(subset, features, target)  # Recursive call

    return tree

# Build decision tree
features = list(data.columns[:-1])
decision_tree = id3(data, features, 'PlayTennis')

# Function to classify a new example
def classify(tree, sample):
    if not isinstance(tree, dict):
        return tree  # Leaf node reached
    root = next(iter(tree))
    return classify(tree[root][sample[root]], sample) if sample[root] in tree[root] else 'Unknown'

# Display the decision tree
import pprint
print("\nDecision Tree:")
pprint.pprint(decision_tree)
```

```
# Test new example
sample = {'Outlook': 'Sunny', 'Temperature': 'Cool', 'Humidity': 'High',
'Wind': 'Strong'}
print("\nNew Sample Classification:", classify(decision_tree, sample))
```

Result:

```
Decision Tree:
{'Outlook': {'Overcast': 'Yes',
             'Rain': {'Wind': {'Strong': 'No', 'Weak': 'Yes'}},
             'Sunny': {'Humidity': {'High': 'No', 'Normal': 'Yes'}}}}

New Sample Classification: No
```

Experiment-6

Objective: To assess how the ID3 algorithm performs on datasets with varying characteristics and complexity, examining overfitting, underfitting, and decision tree depth.

Source Code:

```
import numpy as np
import pandas as pd
import math
from collections import Counter
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import pprint

# Function to compute entropy
def entropy(target_col):
    counts = Counter(target_col)
    total = len(target_col)
    return -sum((count / total) * math.log2(count / total) for count in
counts.values())

# Function to compute information gain
def info_gain(data, split_attribute, target_name):
    total_entropy = entropy(data[target_name])  # Parent entropy
    values = data[split_attribute].unique()  # Unique values of the
attribute
    weighted_entropy = sum(
```

```
        (len(subset := data[data[split_attribute] == val]) / len(data)) *
entropy(subset[target_name])
        for val in values
    )
    return total_entropy - weighted_entropy  # Information gain

# Function to build the decision tree
def id3(data, features, target, depth=0, max_depth=None):
    # Base cases
    if len(set(data[target])) == 1:  # Pure class
        return data[target].iloc[0]
    if not features or (max_depth is not None and depth >= max_depth):
# No more features to split or max depth reached
        return data[target].mode()[0]

    # Select best feature
    best_feature = max(features, key=lambda f: info_gain(data, f, target))

    # Create subtree
    tree = {best_feature: {}}
    features = [f for f in features if f != best_feature]  # Remove used
feature

    for value in data[best_feature].unique():  # Split on each unique value
        subset = data[data[best_feature] == value]
        tree[best_feature][value] = id3(subset, features, target, depth + 1,
max_depth)

    return tree
```

```
# Function to classify new examples
def classify(tree, sample):
    if not isinstance(tree, dict):
        return tree  # Leaf node reached
    root = next(iter(tree))
    return classify(tree[root][sample[root]], sample) if sample[root] in
tree[root] else 'Unknown'

# Function to predict multiple samples
def predict(tree, X):
    return [classify(tree, x) for _, x in X.iterrows()]

# Function to assess performance on different datasets
def evaluate_id3(data, max_depth=None):
    X = data.iloc[:, :-1]
    y = data.iloc[:, -1]

    # Split into training and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

    # Train the decision tree
    features = list(X.columns)
    tree = id3(pd.concat([X_train, y_train], axis=1), features,
y_train.name, max_depth=max_depth)

    # Make predictions
    y_train_pred = predict(tree, X_train)
    y_test_pred = predict(tree, X_test)
```

```
    # Compute accuracy
    train_acc = accuracy_score(y_train, y_train_pred)
    test_acc = accuracy_score(y_test, y_test_pred)

    # Print results
    print(f"\nDecision Tree (Max Depth = {max_depth if max_depth else
'Full'}):")
    pprint.pprint(tree)
    print(f"\nTraining Accuracy: {train_acc:.4f}")
    print(f"Test Accuracy: {test_acc:.4f}")
    print("-" * 40)

# Sample dataset: (Simple, Medium, and Complex)
data_simple = pd.DataFrame([
    ['Sunny', 'Hot', 'High', 'Weak', 'No'],
    ['Sunny', 'Hot', 'High', 'Strong', 'No'],
    ['Overcast', 'Hot', 'High', 'Weak', 'Yes'],
    ['Rain', 'Mild', 'High', 'Weak', 'Yes'],
    ['Rain', 'Cool', 'Normal', 'Weak', 'Yes'],
    ['Rain', 'Cool', 'Normal', 'Strong', 'No'],
    ['Overcast', 'Cool', 'Normal', 'Strong', 'Yes'],
    ['Sunny', 'Mild', 'High', 'Weak', 'No'],
    ['Sunny', 'Cool', 'Normal', 'Weak', 'Yes'],
    ['Rain', 'Mild', 'Normal', 'Weak', 'Yes'],
    ['Sunny', 'Mild', 'Normal', 'Strong', 'Yes'],
    ['Overcast', 'Mild', 'High', 'Strong', 'Yes'],
    ['Overcast', 'Hot', 'Normal', 'Weak', 'Yes'],
    ['Rain', 'Mild', 'High', 'Strong', 'No']
], columns=['Outlook', 'Temperature', 'Humidity', 'Wind', 'PlayTennis'])
```

Run evaluations
evaluate_id3(data_simple, max_depth=1) # Underfitting scenario
evaluate_id3(data_simple, max_depth=3) # Balanced tree
evaluate_id3(data_simple, max_depth=None) # Full tree (potential overfitting)

Result:

```
Decision Tree (Max Depth = 1):
{'Humidity': {'High': 'No', 'Normal': 'Yes'}}

Training Accuracy: 0.7778
Test Accuracy: 0.6000
----------------------------------------

Decision Tree (Max Depth = 3):
{'Humidity': {'High': {'Outlook': {'Overcast': 'Yes',
                                   'Rain': {'Wind': {'Strong': 'No',
                                                     'Weak': 'Yes'}},
                                   'Sunny': 'No'}},
              'Normal': 'Yes'}}

Training Accuracy: 1.0000
Test Accuracy: 0.8000
----------------------------------------

Decision Tree (Max Depth = Full):
{'Humidity': {'High': {'Outlook': {'Overcast': 'Yes',
                                   'Rain': {'Wind': {'Strong': 'No',
                                                     'Weak': 'Yes'}},
                                   'Sunny': 'No'}},
              'Normal': 'Yes'}}

Training Accuracy: 1.0000
Test Accuracy: 0.8000
----------------------------------------
```

Experiment-7

Objective: To examine different types of machine learning approaches (Supervised, Unsupervised, Semi-supervised, and Reinforcement Learning) by setting up a basic classification problem and exploring how each type applies differently.

Source Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from sklearn.cluster import KMeans
from sklearn.semi_supervised import LabelPropagation
import random

# Generate a synthetic dataset (features: X1, X2, class: Y)
np.random.seed(42)
X = np.random.rand(200, 2)  # 200 samples, 2 features
Y = np.where(X[:, 0] + X[:, 1] > 1, 1, 0)  # Simple linear classification
boundary

# Split into training and test sets
```

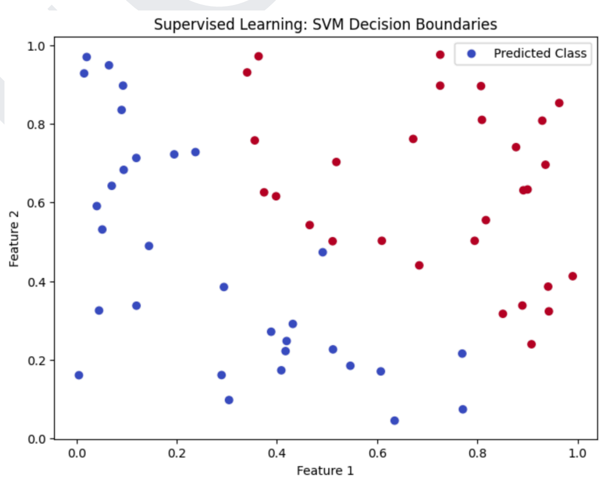
```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3,
random_state=42)

# --------------------- 1. SUPERVISED LEARNING ---------------------
print("\n--- Supervised Learning: Decision Tree ---")
dt_model = DecisionTreeClassifier()
dt_model.fit(X_train, Y_train)
Y_pred = dt_model.predict(X_test)
print("Decision Tree Accuracy:", accuracy_score(Y_test, Y_pred))

print("\n--- Supervised Learning: Support Vector Machine (SVM) ---")
svm_model = SVC()
svm_model.fit(X_train, Y_train)
Y_pred_svm = svm_model.predict(X_test)
print("SVM Accuracy:", accuracy_score(Y_test, Y_pred_svm))

# --------------------- 2. UNSUPERVISED LEARNING ---------------------
print("\n--- Unsupervised Learning: K-Means Clustering ---")
kmeans = KMeans(n_clusters=2, random_state=42)
clusters = kmeans.fit_predict(X_test)  # No true labels used
print("Cluster Assignments:", clusters[:10])  # Show first 10 cluster
labels

# --------------------- 3. SEMI-SUPERVISED LEARNING
---------------------
print("\n--- Semi-Supervised Learning: Label Propagation ---")
labels = np.copy(Y_train)
random_unlabeled = np.random.choice(len(Y_train), size=int(0.8 *
len(Y_train)), replace=False)
labels[random_unlabeled] = -1  # Hide 80% of labels
```

```
lp_model = LabelPropagation()
lp_model.fit(X_train, labels)
Y_pred_lp = lp_model.predict(X_test)
print("Semi-Supervised Accuracy:", accuracy_score(Y_test,
Y_pred_lp))

# --------------------- 4. REINFORCEMENT LEARNING
---------------------
print("\n--- Reinforcement Learning: Q-Learning for Classification ---")
Q_table = np.zeros((2, 2))  # 2 states (classes), 2 actions (predict 0 or 1)
alpha = 0.1  # Learning rate
gamma = 0.9  # Discount factor

# Q-learning process
for _ in range(1000):
    state = random.choice([0, 1])  # Randomly choose a class state
    action = np.argmax(Q_table[state]) if np.random.rand() > 0.2 else
np.random.choice([0, 1])  # ε-greedy policy
    reward = 1 if action == state else -1  # Reward is 1 if prediction is
correct
    Q_table[state, action] = (1 - alpha) * Q_table[state, action] + alpha *
(reward + gamma * np.max(Q_table[action]))

# Testing Q-learning-based classification
Y_pred_q = [np.argmax(Q_table[y]) for y in Y_test]  # Predict based on
learned Q-table
print("Q-Learning Accuracy:", accuracy_score(Y_test, Y_pred_q))

# --------------------- Plot the dataset ---------------------
```

```
plt.figure(figsize=(8, 6))
plt.scatter(X_test[:, 0], X_test[:, 1], c=Y_pred_svm, cmap='coolwarm',
label='Predicted Class')
plt.title('Supervised Learning: SVM Decision Boundaries')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.show()
```

Result:

```
--- Supervised Learning: Decision Tree ---
Decision Tree Accuracy: 0.9166666666666666

--- Supervised Learning: Support Vector Machine (SVM) ---
SVM Accuracy: 0.9833333333333333

--- Unsupervised Learning: K-Means Clustering ---
Cluster Assignments: [1 0 0 0 0 0 1 0 1 1]

--- Semi-Supervised Learning: Label Propagation ---
Semi-Supervised Accuracy: 0.9333333333333333

--- Reinforcement Learning: Q-Learning for Classification ---
Q-Learning Accuracy: 1.0
```



Experiment-8

Objective: To understand how Find-S and Candidate Elimination algorithms search through the hypothesis space in concept learning tasks, and to observe the role of inductive bias in shaping the learned concept.

Source Code:

```
import numpy as np
import pandas as pd

# Sample dataset for concept learning
data = np.array([
    ['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same', 'Yes'],
    ['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same', 'Yes'],
    ['Rainy', 'Cold', 'High', 'Strong', 'Warm', 'Change', 'No'],
    ['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change', 'Yes']
])

# Convert to Pandas DataFrame
columns = ['Sky', 'Temperature', 'Humidity', 'Wind', 'Water', 'Forecast',
'EnjoySport']
df = pd.DataFrame(data, columns=columns)

# Separate features and target
X = df.iloc[:, :-1].values
Y = df.iloc[:, -1].values
```

```
# ------------------- FIND-S ALGORITHM -------------------
def find_s(X, Y):
    hypothesis = np.array(X[0])  # Start with the first positive example
    for i, sample in enumerate(X):
        if Y[i] == 'Yes':  # Only consider positive examples
            for j in range(len(sample)):
                if sample[j] != hypothesis[j]:
                    hypothesis[j] = '?'  # Generalize differing attributes
    return hypothesis

# ------------------- CANDIDATE ELIMINATION ALGORITHM
-------------------
def candidate_elimination(X, Y):
    num_attributes = X.shape[1]

    # Initialize the most specific and most general hypotheses
    S = ['Ø'] * num_attributes  # Most specific hypothesis
    G = ['?'] * num_attributes  # Most general hypothesis

    version_space = []  # Store hypotheses during training

    for i, sample in enumerate(X):
        if Y[i] == 'Yes':  # Positive example
            for j in range(num_attributes):
                if S[j] == 'Ø':
                    S[j] = sample[j]  # Assign first positive sample
                elif S[j] != sample[j]:
                    S[j] = '?'  # Generalize
            version_space.append(tuple(S))  # Track the version space
```

```
        elif Y[i] == 'No':  # Negative example
            for j in range(num_attributes):
                if S[j] != '?' and S[j] != sample[j]:
                    G[j] = S[j]  # Specialize
            version_space.append(tuple(G))

    return S, G, version_space

# Run Find-S Algorithm
find_s_hypothesis = find_s(X, Y)
print("\nFinal Hypothesis (Find-S):", find_s_hypothesis)

# Run Candidate Elimination Algorithm
S_final, G_final, version_space = candidate_elimination(X, Y)
print("\nFinal Specific Hypothesis (S):", S_final)
print("Final General Hypothesis (G):", G_final)
print("Version Space:", version_space)
```

Result:

Final Hypothesis (Find-S): ['Sunny' 'Warm' '?' 'Strong' '?' '?']

Final Specific Hypothesis (S): ['Sunny' 'Warm' '?' 'Strong' '?' '?']
Final General Hypothesis (G): ['?' '?' '?' '?' '?' '?']
Version Space: [('Sunny', 'Warm', '?', 'Strong', '?', '?'), ('?', '?', '?', '?', '?', '?')]

Experiment-9

Objective: To go through all stages of a real-life machine learning project, from data collection to model fine-tuning, using a regression dataset like the "California Housing Prices."

Source Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder

# Load California Housing dataset
from sklearn.datasets import fetch_california_housing
california = fetch_california_housing(as_frame=True)
df = california.frame
```

```
# Display basic dataset information
print(df.info())
print(df.head())

# Check for missing values
print("\nMissing Values:\n", df.isnull().sum())

# Summary statistics
print("\nDataset Statistics:\n", df.describe())

# Visualizing correlations
plt.figure(figsize=(10, 6))
sns.heatmap(df.corr(), annot=True, cmap="coolwarm", linewidths=0.5)
plt.title("Feature Correlation Heatmap")
plt.show()

# Plot target variable distribution
sns.histplot(df['MedHouseVal'], bins=30, kde=True)
plt.title("Median House Value Distribution")
plt.show()

# Define features and target
X = df.drop(columns=["MedHouseVal"])
y = df["MedHouseVal"]

# Identify numerical and categorical columns
num_features = X.select_dtypes(include=["float64",
"int64"]).columns.tolist()
cat_features = X.select_dtypes(include=["object"]).columns.tolist()  # In
this dataset, no categorical features exist.
```

```
# Data Preprocessing Pipeline
num_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler", StandardScaler())
])

cat_transformer = Pipeline(steps=[
    ("encoder", OneHotEncoder(handle_unknown="ignore"))
])

preprocessor = ColumnTransformer(transformers=[
    ("num", num_transformer, num_features),
    ("cat", cat_transformer, cat_features)  # No categorical features in this
dataset, but kept for extensibility
])

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

# Apply transformations
X_train = preprocessor.fit_transform(X_train)
X_test = preprocessor.transform(X_test)

# Train a Linear Regression Model
lr_model = LinearRegression()
lr_model.fit(X_train, y_train)

# Predictions
```

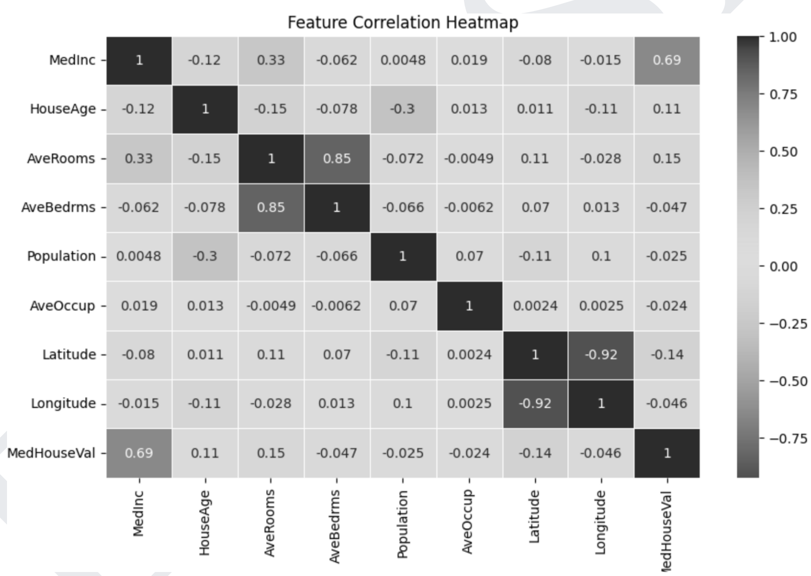
```
y_pred_lr = lr_model.predict(X_test)

# Evaluate Performance
print("\n--- Linear Regression Performance ---")
print(f"R² Score: {r2_score(y_test, y_pred_lr):.4f}")
print(f"MAE: {mean_absolute_error(y_test, y_pred_lr):.4f}")
print(f"RMSE: {np.sqrt(mean_squared_error(y_test, y_pred_lr)):.4f}")


# Train a Random Forest Model
rf_model = RandomForestRegressor(n_estimators=100,
random_state=42)
rf_model.fit(X_train, y_train)

# Predictions
y_pred_rf = rf_model.predict(X_test)

# Evaluate Performance
print("\n--- Random Forest Performance ---")
print(f"R² Score: {r2_score(y_test, y_pred_rf):.4f}")
print(f"MAE: {mean_absolute_error(y_test, y_pred_rf):.4f}")
print(f"RMSE: {np.sqrt(mean_squared_error(y_test, y_pred_rf)):.4f}")


# Define hyperparameter grid
param_grid = {
    "n_estimators": [50, 100, 200],
    "max_depth": [10, 20, None],
    "min_samples_split": [2, 5, 10]
}
```

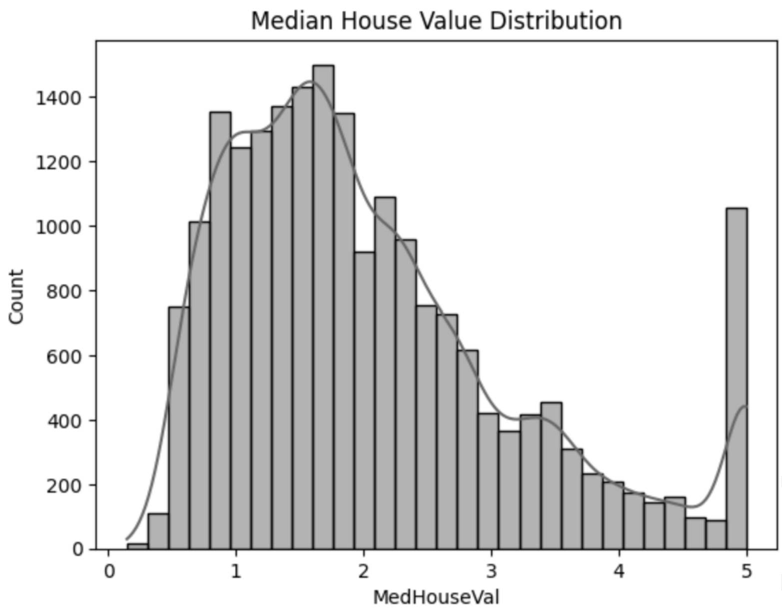
```
# Grid Search with Cross-Validation
grid_search =
GridSearchCV(RandomForestRegressor(random_state=42), param_grid,
cv=3, scoring="r2", n_jobs=-1)
grid_search.fit(X_train, y_train)

# Best model evaluation
best_rf = grid_search.best_estimator_
y_pred_best = best_rf.predict(X_test)

print("\n--- Best Random Forest Performance ---")
print(f"Best Parameters: {grid_search.best_params_}")
print(f"R² Score: {r2_score(y_test, y_pred_best):.4f}")
print(f"MAE: {mean_absolute_error(y_test, y_pred_best):.4f}")
print(f"RMSE: {np.sqrt(mean_squared_error(y_test,
y_pred_best)):.4f}")

# Extract feature importances
importances = best_rf.feature_importances_
feature_names = num_features  # No categorical features in this dataset

# Plot Feature Importance
plt.figure(figsize=(10, 5))
sns.barplot(x=importances, y=feature_names, palette="viridis")
plt.title("Feature Importance in Random Forest Model")
plt.xlabel("Importance Score")
plt.ylabel("Feature")
plt.show()
```

Result:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 9 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   MedInc       20640 non-null  float64
 1   HouseAge     20640 non-null  float64
 2   AveRooms     20640 non-null  float64
 3   AveBedrms    20640 non-null  float64
 4   Population   20640 non-null  float64
 5   AveOccup     20640 non-null  float64
 6   Latitude     20640 non-null  float64
 7   Longitude    20640 non-null  float64
 8   MedHouseVal  20640 non-null  float64
dtypes: float64(9)
memory usage: 1.4 MB
None
   MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0  8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88
1  8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86
2  7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85
3  5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85
4  3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85

   Longitude  MedHouseVal
0    -122.23        4.526
1    -122.22        3.585
2    -122.24        3.521
3    -122.25        3.413
4    -122.25        3.422
```

```
Missing Values:
 MedInc        0
HouseAge       0
AveRooms       0
AveBedrms      0
Population     0
AveOccup       0
Latitude       0
Longitude      0
MedHouseVal    0
dtype: int64

Dataset Statistics:
             MedInc      HouseAge      AveRooms     AveBedrms    Population  \
count  20640.000000  20640.000000  20640.000000  20640.000000  20640.000000
mean       3.870671     28.639486      5.429000      1.096675   1425.476744
std        1.899822     12.585558      2.474173      0.473911   1132.462122
min        0.499900      1.000000      0.846154      0.333333      3.000000
25%        2.563400     18.000000      4.440716      1.006079    787.000000
50%        3.534800     29.000000      5.229129      1.048780   1166.000000
75%        4.743250     37.000000      6.052381      1.099526   1725.000000
max       15.000100     52.000000    141.909091     34.066667  35682.000000

           AveOccup      Latitude     Longitude   MedHouseVal
count  20640.000000  20640.000000  20640.000000  20640.000000
mean       3.070655     35.631861   -119.569704      2.068558
std       10.386050      2.135952      2.003532      1.153956
min        0.692308     32.540000   -124.350000      0.149990
25%        2.429741     33.930000   -121.800000      1.196000
50%        2.818116     34.260000   -118.490000      1.797000
75%        3.282261     37.710000   -118.010000      2.647250
max     1243.333333     41.950000   -114.310000      5.000010
```






```
--- Linear Regression Performance ---
R² Score: 0.5758
MAE: 0.5332
RMSE: 0.7456

--- Random Forest Performance ---
R² Score: 0.8053
MAE: 0.3274
RMSE: 0.5051
```

Experiment-10

Objective: To perform binary and multiclass classification on the MNIST dataset, analyze performance metrics, and perform error analysis.

Source Code:

```
import numpy as np
```

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
from tensorflow.keras.datasets import mnist

# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Normalize images (scale pixel values between 0 and 1)
X_train = X_train / 255.0
X_test = X_test / 255.0

# Flatten images from 28x28 to 1D (784 features)
X_train_flat = X_train.reshape(X_train.shape[0], -1)
X_test_flat = X_test.reshape(X_test.shape[0], -1)

print("Dataset Shape:", X_train.shape, X_test.shape)

# Convert to a binary classification problem (0 vs. not 0)
y_train_binary = (y_train == 0).astype(int)
y_test_binary = (y_test == 0).astype(int)

# Train a logistic regression model
binary_clf = LogisticRegression(max_iter=1000)
binary_clf.fit(X_train_flat, y_train_binary)
```

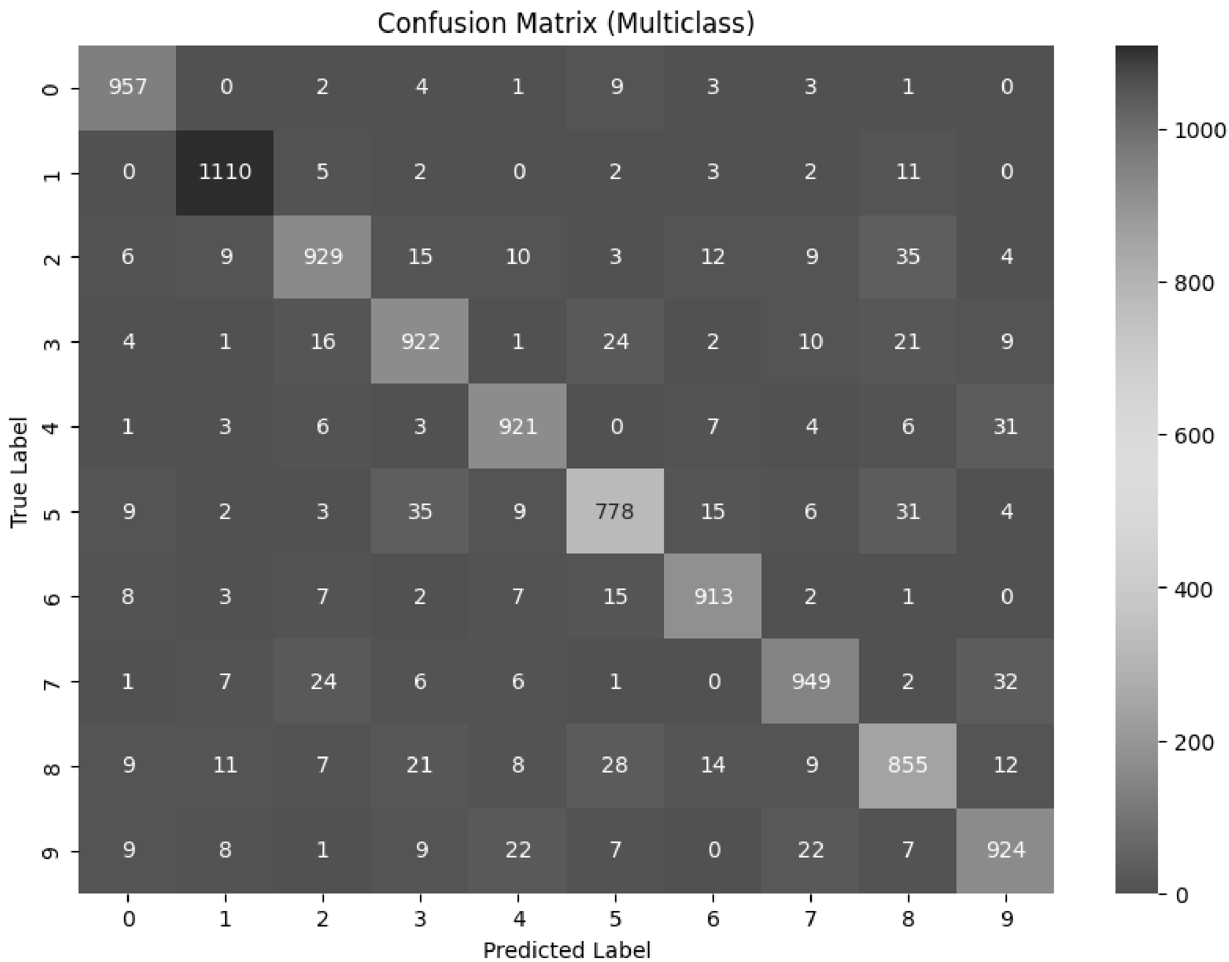
```
# Predictions
y_pred_binary = binary_clf.predict(X_test_flat)

# Evaluate Performance
print("\n--- Binary Classification (0 vs. Others) ---")
print("Accuracy:", accuracy_score(y_test_binary, y_pred_binary))
print(classification_report(y_test_binary, y_pred_binary))

# Train a logistic regression model for multiclass classification
multi_clf = LogisticRegression(max_iter=1000,
multi_class="multinomial", solver="lbfgs")
multi_clf.fit(X_train_flat, y_train)

# Predictions
y_pred_multi = multi_clf.predict(X_test_flat)

# Evaluate Performance
print("\n--- Multiclass Classification (All Digits) ---")
print("Accuracy:", accuracy_score(y_test, y_pred_multi))
print(classification_report(y_test, y_pred_multi))
# Compute Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred_multi)

# Plot Confusion Matrix
plt.figure(figsize=(10, 7))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="coolwarm",
xticklabels=range(10), yticklabels=range(10))
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix (Multiclass)")
```

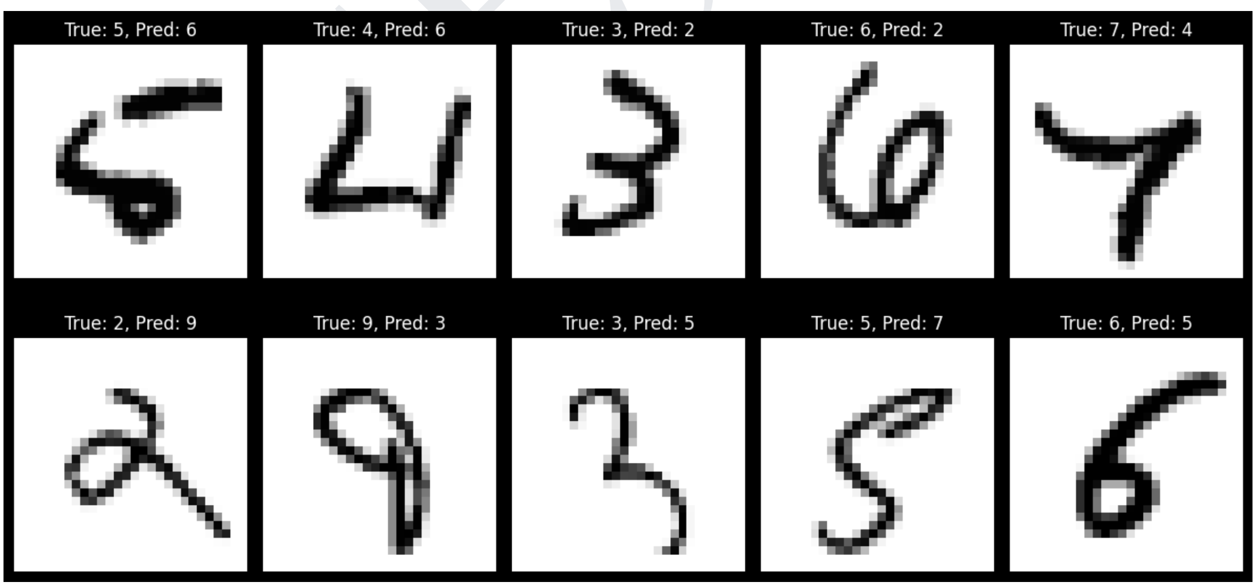
```
plt.show()

# Identify misclassified examples
misclassified_idxs = np.where(y_test != y_pred_multi)[0]

# Display some misclassified images
plt.figure(figsize=(12, 6))
for i, idx in enumerate(misclassified_idxs[:10]):
    plt.subplot(2, 5, i + 1)
    plt.imshow(X_test[idx], cmap="gray")
    plt.title(f"True: {y_test[idx]}, Pred: {y_pred_multi[idx]}")
    plt.axis("off")
plt.tight_layout()
plt.show()
```

Result:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Dataset Shape: (60000, 28, 28) (10000, 28, 28)

--- Binary Classification (0 vs. Others) ---
Accuracy: 0.9917
              precision    recall  f1-score   support

           0       1.00      0.99      1.00      9020
           1       0.95      0.96      0.96       980

    accuracy                           0.99     10000
   macro avg       0.97      0.98      0.98     10000
weighted avg       0.99      0.99      0.99     10000
```

```
--- Multiclass Classification (All Digits) ---
Accuracy: 0.9258
              precision    recall  f1-score   support

           0       0.95      0.98      0.96       980
           1       0.96      0.98      0.97      1135
           2       0.93      0.90      0.91      1032
           3       0.90      0.91      0.91      1010
           4       0.94      0.94      0.94       982
           5       0.90      0.87      0.88       892
           6       0.94      0.95      0.95       958
           7       0.93      0.92      0.93      1028
           8       0.88      0.88      0.88       974
           9       0.91      0.92      0.91      1009

    accuracy                           0.93     10000
   macro avg       0.92      0.92      0.92     10000
weighted avg       0.93      0.93      0.93     10000
```


Confusion Matrix (Multiclass)
True Label
Predicted Label
957 0 2 4 1 9 3 3 1 0
0 1110 5 2 0 2 3 2 11 0
6 9 929 15 10 3 12 9 35 4
4 1 16 922 1 24 2 10 21 9
1 3 6 3 921 0 7 4 6 31
9 2 3 35 9 778 15 6 31 4
8 3 7 2 7 15 913 2 1 0
1 7 24 6 6 1 0 949 2 32
9 11 7 21 8 28 14 9 855 12
9 8 1 9 22 7 0 22 7 924
0 1 2 3 4 5 6 7 8 9
1000
800
600
400
200
0


True: 5, Pred: 6
True: 4, Pred: 6
True: 3, Pred: 2
True: 6, Pred: 2
True: 7, Pred: 4
True: 2, Pred: 9
True: 9, Pred: 3
True: 3, Pred: 5
True: 5, Pred: 7
True: 6, Pred: 5