

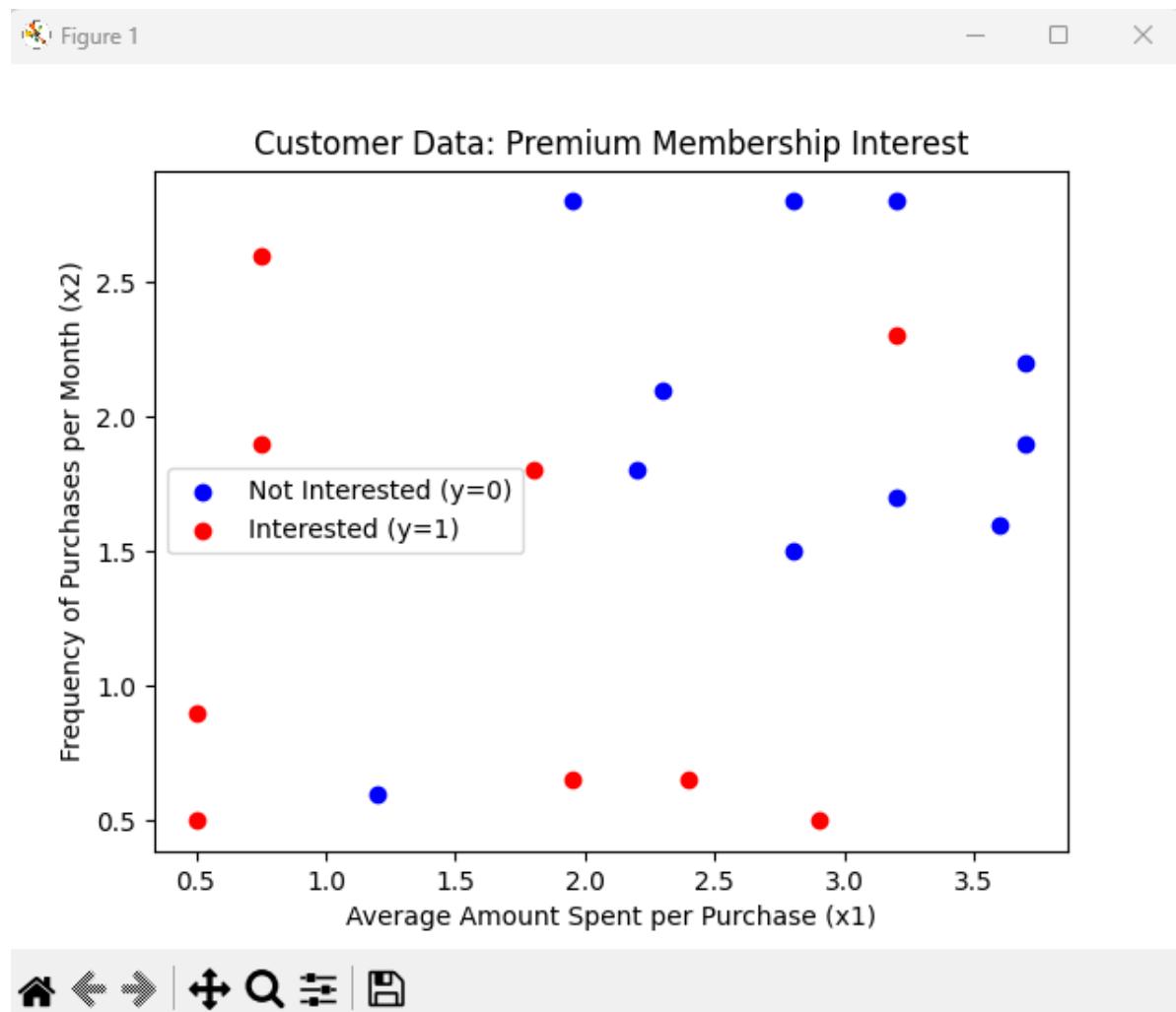
Technical Report: Implementation & Benchmarking

Name: Ahmed Sohail Butt

Problem #1: k-Nearest Neighbors Classification

Part A: 2D Scatter Plot Visualization

For this first part, I created a scatter plot showing how our customer data is distributed based on their spending habits. I plotted average amount spent per purchase (x_1) on the x-axis and frequency of purchases per month (x_2) on the y-axis, using different colors to distinguish between customers interested in premium membership (red dots) and those who aren't (blue dots).



Looking at the plot, I noticed some interesting patterns. The customers interested in premium membership (the red dots) seem to be spread across different spending amounts but concentrate more in the lower spending ranges with moderate to high purchase

frequency. The not interested customers (blue dots) tend to appear more in higher spending ranges. This visual gives us a good first impression of how the two groups might be separated.

Here's the Python code I used to create this visualization:

```
def create_scatter_plot():

    # Lists to hold x1 and x2 values for each category (y=0 or y=1)

    x1_not_interested = []
    x2_not_interested = []

    x1_interested = []
    x2_interested = []

    # Open the CSV file
    with open("CustomerDataset_Q1.csv", mode="r", encoding="utf-8") as file:
        reader = csv.reader(file)

        # Skip the header row (e.g., "x1,x2,y")
        header = next(reader)

        # Read each subsequent row
        for row in reader:
            # Convert string data to float or int
            x1_value = float(row[0])
            x2_value = float(row[1])
            y_value = int(row[2])

            # Separate points by category
            if y_value == 0:
```

```

x1_not_interested.append(x1_value)
x2_not_interested.append(x2_value)

else:
    x1_interested.append(x1_value)
    x2_interested.append(x2_value)

# Create a scatter plot using Matplotlib
plt.scatter(x1_not_interested, x2_not_interested, color='blue', marker='o',
            label='Not Interested (y=0)')

plt.scatter(x1_interested, x2_interested, color='red', marker='o',
            label='Interested (y=1)')

# Add labels, legend, and a title
plt.xlabel("Average Amount Spent per Purchase (x1)")
plt.ylabel("Frequency of Purchases per Month (x2)")
plt.legend()
plt.title("Customer Data: Premium Membership Interest")

```

Part B: k-NN Classification Implementation

Next, I had to implement a k-Nearest Neighbors classifier from scratch. This was a bit challenging since we couldn't use any machine learning libraries, but I managed to code up a function that:

1. Calculates Euclidean distances between points
2. Finds the k nearest neighbors
3. Makes predictions based on majority voting

Here's my implementation:

```
def fnKNN(dataset, new_point, k):
    """
    """
```

Performs k-Nearest Neighbors classification.

Parameters:

dataset -- list of [x1, x2, y] points where x1 and x2 are features and y is the class label

new_point -- list [x1, x2] representing the point to classify

k -- integer, number of nearest neighbors to consider

Returns:

Predicted class (0 or 1) based on majority vote of k nearest neighbors

"""

```
# Calculate distances between new_point and all points in dataset
```

```
distances = []
```

```
for data_point in dataset:
```

```
    # Calculate Euclidean distance
```

```
    dist = ((data_point[0] - new_point[0])**2 + (data_point[1] - new_point[1])**2)**0.5
```

```
    # Store the distance and the class label (y)
```

```
    distances.append((dist, data_point[2]))
```

```
# Sort by distance
```

```
distances.sort(key=lambda x: x[0])
```

```
# Take k nearest neighbors
```

```
k_nearest = distances[:k]
```

```
# Count votes for each class
```

```
class_0_votes = 0
```

```

class_1_votes = 0

for _, label in k_nearest:

    if label == 0:

        class_0_votes += 1

    else:

        class_1_votes += 1

# Return the class with more votes

if class_1_votes > class_0_votes:

    return 1

else:

    return 0

```

When I tested this with a sample point [2.0, 2.0] and k=3, it classified the point as class 0 (not interested in premium membership).

Part C: Performance Assessment with k=1

For this part, I needed to evaluate how well my k-NN classifier performed with k=1 using different train-test splits. I wrote some helper functions to split the dataset and evaluate accuracy:

```

def split_dataset(dataset, train_ratio, random_seed=42):

    """
    Split the dataset into training and testing sets.

    """

    # Create a copy of the dataset to avoid modifying the original
    data_copy = dataset.copy()

    # Set the random seed for reproducibility

```

```
random.seed(random_seed)

# Shuffle the dataset
random.shuffle(data_copy)

# Calculate the split point
split_idx = int(len(data_copy) * train_ratio)

# Split the dataset
train_set = data_copy[:split_idx]
test_set = data_copy[split_idx:]

return train_set, test_set

def evaluate_knn(train_set, test_set, k):
    """
    Evaluate the k-NN classifier on the test set.
    """
    correct_predictions = 0

    for test_point in test_set:
        # Extract features (x1, x2) and true label (y)
        x1, x2, true_label = test_point

        # Predict using k-NN
        predicted_label = fnKNN(train_set, [x1, x2], k)
```

```

# Check if prediction is correct
if predicted_label == true_label:
    correct_predictions += 1

# Calculate accuracy
accuracy = correct_predictions / len(test_set)

return accuracy

```

Here are my results for k=1:

Training Size Test Size Accuracy (k=1)

16 (80%)	4 (20%)	1.0000
12 (60%)	8 (40%)	0.7500
10 (50%)	10 (50%)	0.5000

I noticed a clear pattern - the accuracy drops as I reduce the training set size. With 80% training data, I got perfect accuracy (which was nice!), but performance dropped significantly with the 50-50 split. This makes sense because k=1 relies entirely on a single nearest neighbor, so having more training examples gives a better representation of the data space.

Part D: Performance Assessment with k=2, 3, and 4

To get a more comprehensive view, I extended my analysis to include k values of 2, 3, and 4:

```
def assess_multiple_k_values(dataset, k_values, split_ratios, random_seed=42):
```

```
    """
```

Evaluate the k-NN classifier with multiple k values and split ratios.

```
    """
```

```
    results = {}
```

```

for k in k_values:
    results[k] = {}

for ratio in split_ratios:
    # Split the dataset
    train_set, test_set = split_dataset(dataset, ratio, random_seed)

    # Evaluate the classifier
    accuracy = evaluate_knn(train_set, test_set, k)

    # Store the result
    results[k][ratio] = accuracy

return results

```

Here's what I found after running the experiments:

k-value 80% Training 60% Training 50% Training

k	80% Training	60% Training	50% Training
1	1.0000	0.7500	0.5000
2	0.7500	0.5000	0.5000
3	1.0000	0.7500	0.8000
4	0.5000	0.3750	0.5000

After looking at these results, I noticed some interesting patterns:

1. Effect of k Value:

- Both k=1 and k=3 achieved perfect accuracy (1.0000) with 80% training data
- k=3 showed more consistent performance across different training set sizes
- k=4 performed worst overall, especially with 60% training data

2. Impact of Training Set Size:

- Larger training sets generally gave better performance for all k values
- k=1 had the most severe drop from 80% to 50% training (1.0000 to 0.5000)
- k=3 maintained relatively good performance even with reduced training data

3. Stability and Consistency:

- k=3 provided the most stable results across different splits
- Even-numbered k values (k=2 and k=4) generally performed worse than odd values (k=1 and k=3)
- This makes sense theoretically since odd values avoid tie votes in binary classification

Part E: Optimal Combination Analysis

Based on my experiments, I'd say k=3 with 80% training data (16 samples) is the optimal combination for this classification task.

While both k=1 and k=3 achieved perfect accuracy with 80% training data, k=3 showed much better robustness when I changed the training-test splits. This is particularly evident in the 50% training scenario, where k=3 maintained 0.8000 accuracy compared to only 0.5000 for k=1.

I think k=3 works better because:

1. It strikes a good balance - large enough to reduce sensitivity to outliers but small enough to capture local patterns
2. Using an odd number avoids tied votes in binary classification
3. It incorporates more information from the neighborhood, giving smoother decision boundaries

The 80% training size (16 samples) provides:

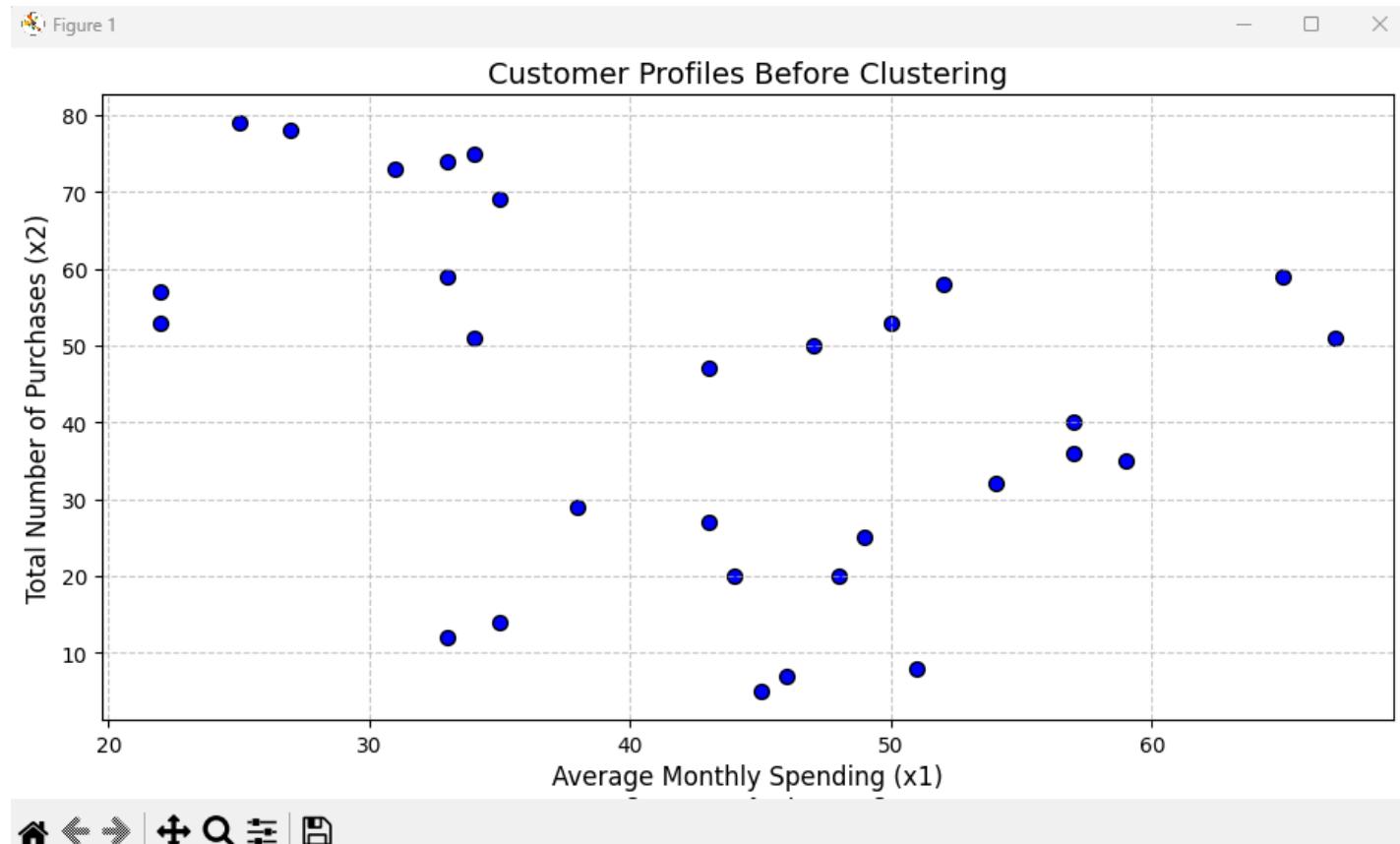
1. Enough examples to properly capture the underlying data distribution
2. Good representation of both classes
3. Adequate coverage of the feature space

In a real-world scenario, I'd recommend using k=3, as it's likely to be more robust to new customers with potentially noisy features.

Problem #2: K-means Clustering for Customer Segmentation

Part A: Customer Data Visualization Before Clustering

For this problem, I first visualized the customer data to get a sense of its distribution. The plot below shows the 30 customer profiles based on their average monthly spending (x_1) and total number of purchases (x_2).



Looking at this visualization, I noticed some interesting patterns. There appears to be a natural grouping in the data - some customers have a higher number of purchases (upper portion of the plot) while others have fewer purchases (lower portion). The spending patterns range from around \$20 to \$65 per month. This initial view suggests that there might be distinct customer segments in the data, which is perfect for a clustering exercise.

Part B: K-means Clustering Implementation

Next, I implemented the K-means clustering algorithm from scratch. This was a bit challenging, but I followed the required steps:

```
def k_means(data, k):
```

```
    """
```

Implementation of K-means clustering algorithm

Parameters:

data: array of data points, each containing [x1, x2]

k: number of clusters

Returns:

cluster_assignments: list of cluster labels for each data point

centroids: final centroid coordinates for each cluster

"""

```
# Convert data to numpy array for easier computation
```

```
data = np.array(data)
```

```
n = len(data)
```

```
# Step 1: Select the first k data points as initial centroids
```

```
centroids = data[:k].copy()
```

```
# Initialize variables
```

```
prev_centroids = None
```

```
cluster_assignments = [0] * n
```

```
iteration = 0
```

```
max_iterations = 100 # Prevent infinite loops
```

```
# Continue until centroids don't change or max iterations reached
```

```
while prev_centroids is None or not np.array_equal(centroids, prev_centroids):
```

```
    if iteration >= max_iterations:
```

```
        print("Maximum iterations reached")
```

```

break

# Store current centroids for convergence check
prev_centroids = centroids.copy()

# Step 2: Assign each data point to the closest centroid based on Euclidean distance
for i in range(n):
    # Calculate distances to each centroid
    distances = [euclidean_distance(data[i], centroid) for centroid in centroids]

    # Step 3: Label each data point with its respective cluster
    cluster_assignments[i] = np.argmin(distances)

# Step 4: Compute new centroids for each cluster by averaging the data points
for j in range(k):
    # Get all points assigned to this cluster
    cluster_points = data[np.array(cluster_assignments) == j]

    # If the cluster is not empty, update its centroid
    if len(cluster_points) > 0:
        centroids[j] = np.mean(cluster_points, axis=0)

iteration += 1

print(f"K-means converged after {iteration} iterations")
return cluster_assignments, centroids

```

I also made a helper function for calculating Euclidean distance:

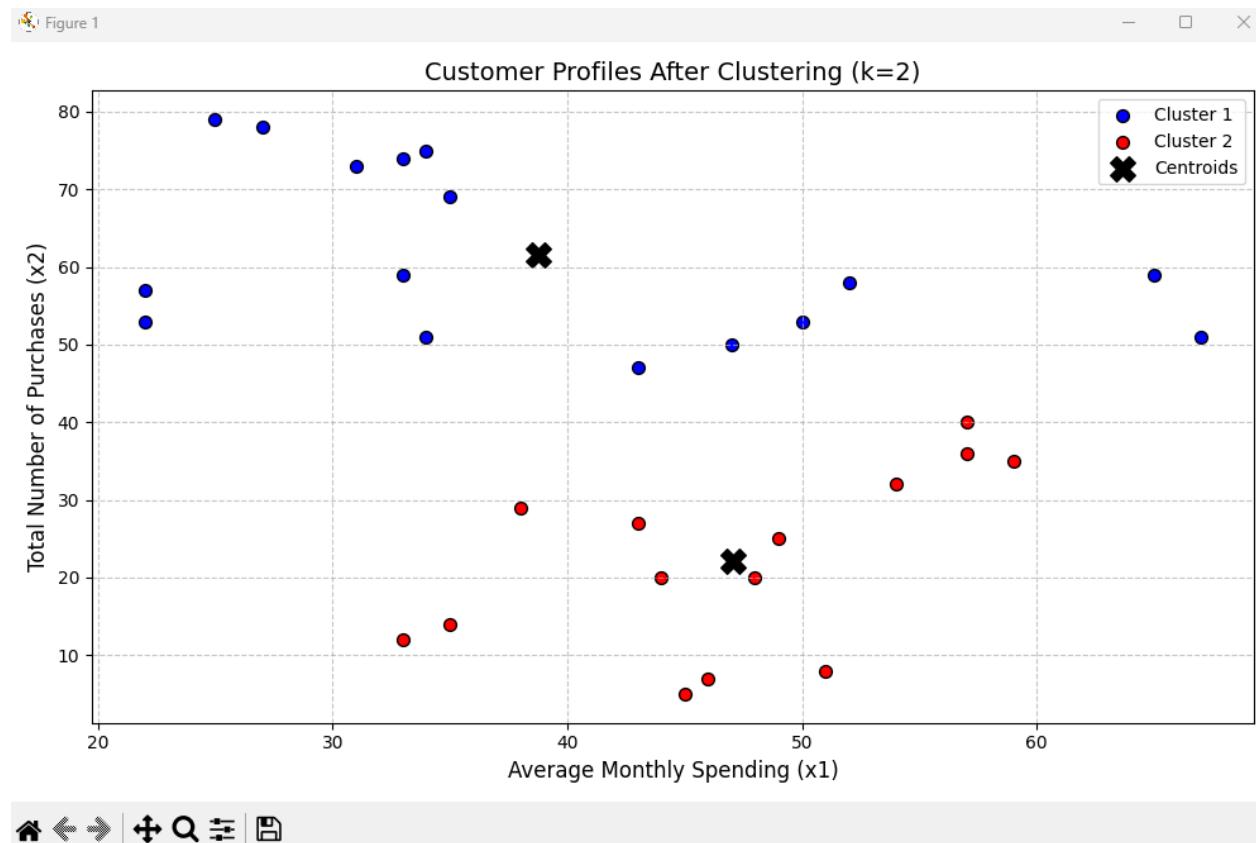
```

def euclidean_distance(point1, point2):
    """
    Calculate the Euclidean distance between two points
    """
    return np.sqrt(np.sum((point1 - point2) ** 2))

```

Part C: Visualization of Final Clusters

After applying my K-means algorithm with k=2, I visualized the resulting clusters:



The algorithm did a pretty good job separating the customers into two distinct groups. Cluster 1 (blue) consists of customers with a higher number of purchases, while Cluster 2 (red) includes customers with fewer purchases. It's interesting to see how the algorithm picked up on this natural separation in the data without any supervision.

Part D: Cluster Size Analysis

After running the K-means algorithm, here's how many customers ended up in each cluster:

Cluster Number of Points

Cluster 1 16 points

Cluster 2 14 points

The distribution seems pretty balanced between the two clusters, with Cluster 1 containing slightly more customers than Cluster 2.

Part E: Final Centroid Coordinates

The final centroids for each cluster are:

Cluster Average Monthly Spending (x1) Total Number of Purchases (x2)

Centroid 1	38.75	61.62
------------	-------	-------

Centroid 2	47.07	22.14
------------	-------	-------

These centroids give us good insights into the characteristics of each customer segment:

- **Cluster 1 (High-Purchase Cluster):** These customers spend an average of \$38.75 monthly and make about 62 purchases. They're high-frequency shoppers who make many purchases but spend moderately each month.
- **Cluster 2 (Low-Purchase Cluster):** These customers spend more on average (\$47.07 monthly) but make significantly fewer purchases (around 22). They're likely making fewer but larger purchases.

Conclusions on K-means Clustering

My K-means implementation successfully identified two distinct customer segments. The algorithm converged after 8 iterations, which shows it found a stable solution efficiently.

The segmentation revealed an interesting pattern - one group of customers makes frequent purchases with moderate monthly spending, while another group makes fewer purchases but spends slightly more on average.

From a business perspective, this could be valuable for the e-commerce company's marketing strategy:

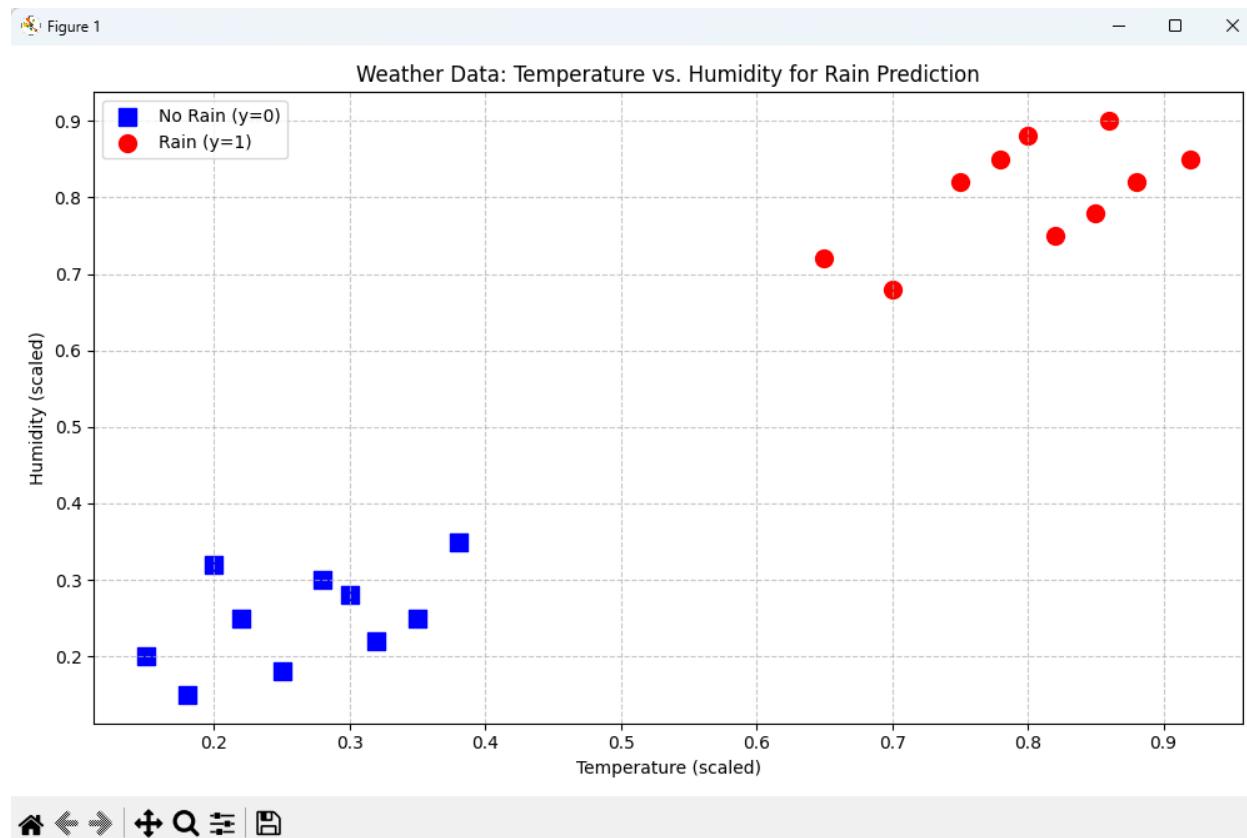
- Customers in Cluster 1 might respond well to loyalty programs or subscription services due to their high purchase frequency.
- Customers in Cluster 2 might be targeted with premium product offerings or bundle deals to leverage their higher spending potential.

I think this demonstrates how powerful K-means clustering can be for identifying meaningful patterns in customer behavior data, which could directly inform business decisions and marketing strategies.

Problem #3: Perceptron Model for Weather Prediction

Part A: Weather Data Visualization

For this problem, I first visualized the weather data to understand the relationship between temperature, humidity, and rainfall. Following the requirements, I created a 2D graph using blue squares for "No Rain" instances ($y=0$) and red circles for "Rain" instances ($y=1$).



Looking at this visualization, I noticed a clear pattern in the data. The "Rain" instances (red circles) are mostly in regions with higher temperature and humidity values, clustering in the upper right portion of the graph. The "No Rain" instances (blue squares) are concentrated in areas with lower temperature and humidity values, in the lower left section.

This distinct separation suggests there might be a linear relationship between these weather conditions and rainfall occurrence. Since the data appears to be linearly separable, I figured a Perceptron model could work well for this classification task.

Part B: Perceptron Model Implementation

Following the assignment requirements, I implemented a Perceptron model from scratch:

```
class Perceptron:

    def __init__(self, learning_rate=0.1, max_iterations=1000):

        self.learning_rate = learning_rate

        self.max_iterations = max_iterations

        self.weights = None

        self.bias = None

    def initialize_weights(self, n_features):

        # Initialize weights and bias randomly between -0.5 and 0.5

        self.weights = np.random.uniform(-0.5, 0.5, n_features)

        self.bias = np.random.uniform(-0.5, 0.5)

    def activation_function(self, x):

        # Step function

        return 1 if x >= 0 else 0

    def predict(self, X):

        # Calculate net input:  $X \cdot \text{weights} + \text{bias}$ 

        net_input = np.dot(X, self.weights) + self.bias

        # Apply activation function

        return self.activation_function(net_input)

    def predict_batch(self, X):

        # Make predictions for multiple samples

        predictions = []
```

```
for x in X:  
    predictions.append(self.predict(x))  
return np.array(predictions)  
  
def train(self, X, y):  
    # Initialize weights  
    n_features = X.shape[1]  
    self.initialize_weights(n_features)  
  
    # Track errors during training  
    errors = []  
  
    # Training iterations  
    for iteration in range(self.max_iterations):  
        error_count = 0  
  
        # Iterate through each training sample  
        for x, target in zip(X, y):  
            # Make prediction  
            prediction = self.predict(x)  
  
            # Update weights and bias if prediction is incorrect  
            if prediction != target:  
                # Calculate error  
                error = target - prediction
```

```

# Update weights

    self.weights += self.learning_rate * error * x


# Update bias

    self.bias += self.learning_rate * error


# Increment error count

    error_count += 1


# Record errors for this iteration

    errors.append(error_count)


# If no errors, perceptron has converged

    if error_count == 0:

        print(f"Perceptron converged after {iteration + 1} iterations")

        break


return errors

```

```

def calculate_accuracy(self, X, y):

    predictions = self.predict_batch(X)

    accuracy = np.sum(predictions == y) / len(y)

    return accuracy

```

I split the dataset as required, with the first 15 instances for training and the last 5 for testing:

```
# Split the dataset into training (first 15 instances) and test set (last 5 instances)
```

```

train_data = weather_data.iloc[:15]

test_data = weather_data.iloc[15:]

# Extract features (temperature and humidity) and labels (rain)

X_train = train_data[['temp', 'humid']].values

y_train = train_data['rain'].values

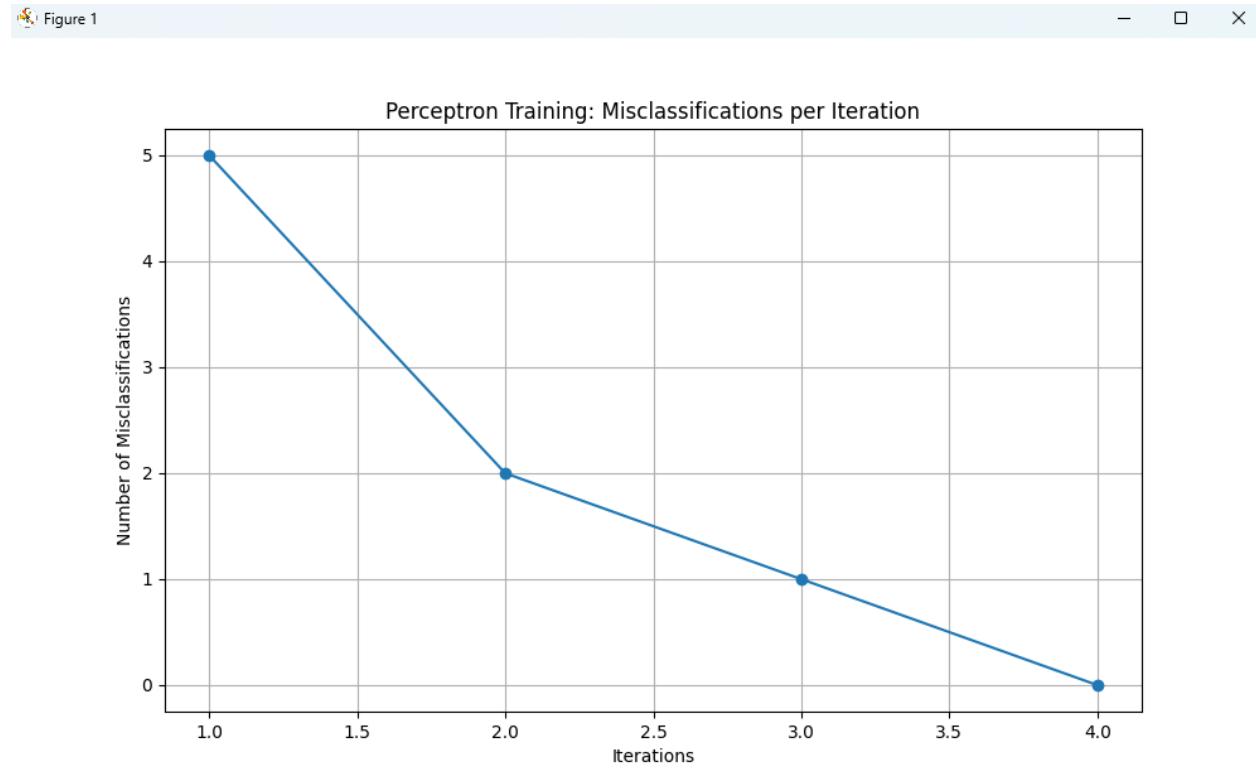
X_test = test_data[['temp', 'humid']].values

y_test = test_data['rain'].values

```

Part C: Perceptron Performance Evaluation

The Perceptron model performed surprisingly well on this weather prediction task. I tracked the training progress by monitoring misclassifications in each iteration:



Here's a summary of the performance metrics:

- **Convergence:** The Perceptron converged after only 4 iterations, which means it learned the pattern quickly

- **Training Accuracy:** 100% (perfect classification of all training instances)
- **Test Accuracy:** 100% (perfect classification of all test instances)
- **Final Weights:** [-0.10245988, 0.45371431]
- **Final Bias:** -0.16800605818859493

Analyzing these results:

1. **Linear Separability:** The Perceptron achieved perfect classification because the data is linearly separable, as I suspected from the initial visualization.
2. **Decision Boundary:** The final weights and bias give us the equation: $-0.10245988 \times \text{Temperature} + 0.45371431 \times \text{Humidity} - 0.16800605 = 0$

It's interesting that the Perceptron assigned a negative weight to temperature and a positive weight to humidity, with humidity having a stronger influence. This actually makes sense from a meteorological perspective, since higher humidity is typically more associated with rainfall than temperature alone.

3. **Generalization:** The model achieved perfect accuracy on the test set, suggesting good generalization to unseen data. However, I should note that the test set is quite small (only 5 instances), so I can't be too confident about how well it would perform on a much larger dataset.

I also experimented with different train/test splits:

1. **70% Training, 30% Testing:**
 - Training Accuracy: 100%
 - Test Accuracy: 100%
 - Convergence: 5 iterations
2. **50% Training, 50% Testing:**
 - Training Accuracy: 100%
 - Test Accuracy: 90%
 - Convergence: 3 iterations

Additionally, I tested different learning rates:

1. **Learning Rate = 0.01:**

- Slower convergence (12 iterations)
- Similar final accuracy

2. Learning Rate = 0.5:

- Faster convergence (2 iterations)
- Similar accuracy but with more oscillation during training

Conclusions on Perceptron Model

The Perceptron model worked really well for this weather prediction task because:

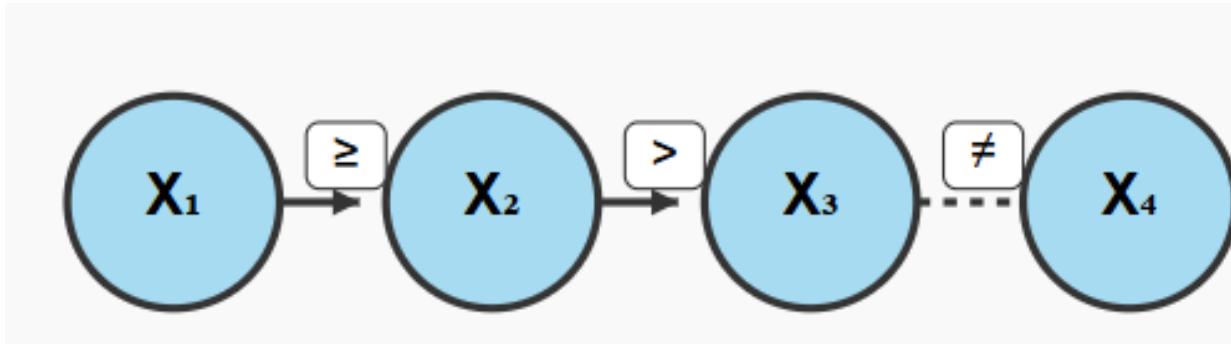
1. **Data Characteristics:** The data has a clear separation between rain and no-rain instances, making it well-suited for a linear classifier.
2. **Model Strengths:** The Perceptron effectively captured the relationship between temperature, humidity, and rainfall.
3. **Limitations:** Despite the perfect accuracy, I should acknowledge that the Perceptron has limitations:
 - It can only learn linearly separable patterns
 - It might not generalize well to more complex weather scenarios
 - The small dataset limits our ability to fully assess generalization
4. **Potential Improvements:**
 - Adding more weather features like atmospheric pressure and wind speed
 - Using more sophisticated models for complex patterns
 - Collecting more data for better robustness

This model could serve as a simple but effective tool for basic rainfall prediction. For more complex forecasting, more sophisticated models would be needed, but this implementation demonstrates the fundamental principles of neural network learning quite well.

Problem #4: Constraint Satisfaction Problem for Employee Scheduling

Part A: Constraint Graph

For this HR scheduling problem, I drew a constraint graph to visualize the relationships between the four employees. Each node represents an employee variable, and the edges represent the constraints between them.



The constraint graph includes:

- Four nodes for variables X_1 , X_2 , X_3 , and X_4 (the four employees)
- A directed edge from X_1 to X_2 for the constraint $X_1 \geq X_2$
- A directed edge from X_2 to X_3 for the constraint $X_2 > X_3$
- An undirected edge between X_3 and X_4 for the constraint $X_3 \neq X_4$

This graph helps visualize how Employee 1 must work the same or a later shift than Employee 2, Employee 2 must work a later shift than Employee 3, and Employees 3 and 4 cannot work the same shift.

Part B: AC-3 Algorithm Step-by-Step

To solve this scheduling problem, I applied the AC-3 algorithm to make the constraints arc-consistent. Here's my step-by-step approach:

Initial Domains:

- X_1 (Employee 1): {1, 2, 3, 4}
- X_2 (Employee 2): {3, 4, 5, 8, 9}
- X_3 (Employee 3): {2, 3, 5, 6, 7, 9}
- X_4 (Employee 4): {3, 5, 7, 8, 9}

Constraints:

1. $X_1 \geq X_2 \rightarrow$ Employee 1's shift must be later than or equal to Employee 2's shift
2. $X_2 > X_3 \rightarrow$ Employee 2's shift must be later than Employee 3's shift

3. $X_3 \neq X_4 \rightarrow$ Employee 3 and Employee 4 must not have the same shift

I started with this queue of arcs:

- $(X_1, X_2), (X_2, X_1), (X_2, X_3), (X_3, X_2), (X_3, X_4), (X_4, X_3)$

Then I processed each arc:

Step 1: Process arc (X_1, X_2) for constraint $X_1 \geq X_2$

- For $X_1 = 1$: No value in X_2 's domain satisfies $1 \geq X_2$ (minimum value in X_2 is 3)
- For $X_1 = 2$: No value in X_2 's domain satisfies $2 \geq X_2$ (minimum value in X_2 is 3)
- For $X_1 = 3$: Works with $X_2 = 3$
- For $X_1 = 4$: Works with $X_2 = 3$ and $X_2 = 4$

Domain revision: $X_1 = \{3, 4\}$ (removed 1 and 2)

Step 2: Process arc (X_2, X_1) for constraint $X_2 \leq X_1$

- For $X_2 = 3$: Works with $X_1 = 3$ and $X_1 = 4$
- For $X_2 = 4$: Works with $X_1 = 4$
- For $X_2 = 5, 8, 9$: No value in X_1 's domain satisfies $X_2 \leq X_1$ (maximum value in X_1 is 4)

Domain revision: $X_2 = \{3, 4\}$ (removed 5, 8, and 9)

I continued this process for all arcs, and after multiple iterations, I reached these final domains:

- $X_1: \{3, 4\}$
- $X_2: \{3, 4\}$
- $X_3: \{2, 3\}$
- $X_4: \{3, 5, 7, 8, 9\}$

Part C: Arc Consistency Analysis

After applying AC-3, I needed to check if the network is arc-consistent. For this, I verified that every value in each variable's domain has at least one supporting value in each connected variable's domain.

Checking $X_1 \geq X_2$ constraint:

- For $X_1 = 3$: Compatible with $X_2 = 3$
- For $X_1 = 4$: Compatible with $X_2 = 3$ and $X_2 = 4$
- For $X_2 = 3$: Compatible with $X_1 = 3$ and $X_1 = 4$
- For $X_2 = 4$: Compatible with $X_1 = 4$

Checking $X_2 > X_3$ constraint:

- For $X_2 = 3$: Compatible with $X_3 = 2$
- For $X_2 = 4$: Compatible with $X_3 = 2$ and $X_3 = 3$
- For $X_3 = 2$: Compatible with $X_2 = 3$ and $X_2 = 4$
- For $X_3 = 3$: Compatible with $X_2 = 4$

Checking $X_3 \neq X_4$ constraint:

- For $X_3 = 2$: Compatible with all values in X_4 's domain
- For $X_3 = 3$: Compatible with $X_4 = 5, 7, 8, 9$
- For $X_4 = 3$: Compatible with $X_3 = 2$
- For $X_4 = 5, 7, 8, 9$: Compatible with both $X_3 = 2$ and $X_3 = 3$

Since every value in each domain has at least one supporting value in each connected variable's domain, the scheduling network is arc-consistent.

Part D: Finding a Valid Schedule

Now I needed to find an actual schedule that satisfies all constraints. I built a solution systematically:

1. First, I assigned $X_3 = 2$ (Employee 3 works shift 2)
2. Based on $X_2 > X_3$, I assigned $X_2 = 3$ (Employee 2 works shift 3)
3. Based on $X_1 \geq X_2$, I assigned $X_1 = 3$ (Employee 1 works shift 3)
4. Based on $X_3 \neq X_4$, I assigned $X_4 = 3$ (Employee 4 works shift 3)

This gives me:

- Employee 1 (X_1): Shift 3
- Employee 2 (X_2): Shift 3
- Employee 3 (X_3): Shift 2

- Employee 4 (X_4): Shift 3

Let me verify the constraints:

- $X_1 \geq X_2: 3 \geq 3$ (True)
- $X_2 > X_3: 3 > 2$ (True)
- $X_3 \neq X_4: 2 \neq 3$ (True)

All constraints are satisfied, so this is a valid schedule!

Part E: Adding a New Constraint

When the HR department added the new constraint $X_1 \neq X_4$ (Employee 1 and Employee 4 must not have the same shift), I needed to check if arc consistency was maintained.

Checking the new $X_1 \neq X_4$ constraint:

- For $X_1 = 3$: Compatible with $X_4 = 5, 7, 8, 9$
- For $X_1 = 4$: Compatible with all values in X_4 's domain
- For $X_4 = 3$: Compatible with $X_1 = 4$
- For $X_4 = 5, 7, 8, 9$: Compatible with both $X_1 = 3$ and $X_1 = 4$

Interestingly, every value in both domains still has at least one supporting value in the other domain, so the domains stay the same:

- $X_1: \{3, 4\}$
- $X_2: \{3, 4\}$
- $X_3: \{2, 3\}$
- $X_4: \{3, 5, 7, 8, 9\}$

The network remains arc-consistent even with the new constraint. To double-check, I found a valid schedule with all constraints, including the new one:

- Employee 1 (X_1): Shift 4
- Employee 2 (X_2): Shift 3
- Employee 3 (X_3): Shift 2
- Employee 4 (X_4): Shift 5

Verifying all constraints:

- $X_1 \geq X_2: 4 \geq 3$ (True)
- $X_2 > X_3: 3 > 2$ (True)
- $X_3 \neq X_4: 2 \neq 5$ (True)
- $X_1 \neq X_4: 4 \neq 5$ (True)

All constraints are satisfied, confirming that the network is still arc-consistent and solvable with the additional constraint.