Part A - Supervised Learning (SL) - Regression

A.I. Linear regression

Part 1: Our group is working with data rows from 242 to 261:

X	Y
23,1	452,17
27,13	434,47
26,95	436,8
19,4	459,76
13,5	468,85
11,97	468,81
18,11	454,29
24,1	442,8
20,29	445,52
21,82	447,43
29,45	437,31
10,87	477,03
17,36	456,57
24,97	440,03
13,72	466,24
26,49	440,21
8,25	475,17
12,8	466,95
17,08	457,81
31,4	433,34

Part 2: We performed Min-Max normalization on our data, as follows:

$x' = \frac{x - min(x)}{max(x) - min(x)}$, where x is our data.

In our code, the formula is implemented as follows:

```
min_max = (dataA1 - dataA1.min()) / (dataA1.max() - dataA1.min())
```

We used Min-Max normalization, as it preserves the relationships between the values, while being able to scale them such that they are all on a similar scale. This way, each value contributes proportionally, and comparisons and analysis between features are made easier. Furthermore, the gradient descent should be able to converge quicker thanks to this normalization.

The transformed data is:

X	Y
0.641469	0.430991

0.815551	0.025864
0.807775	0.079194
0.481641	0.604715
0.226782	0.812772
0.160691	0.811856
0.425918	0.479515
0.684665	0.216526
0.520086	0.278782
0.586177	0.322499
0.915767	0.090867
0.113175	1.000000
0.393521	0.531701
0.722246	0.153124
0.236285	0.753033
0.787905	0.157244
0.000000	0.957427
0.196544	0.769284
0.381425	0.560082
1.000000	0.000000

Part 3: The formula for least squares (closed form solution) is as follows:

$\theta = (X^T X)^{-1} X^T y$, which is implemented in our code as:

theta = np.dot((np.dot(np.linalg.inv(np.dot(X.T, X)), X.T)), y)

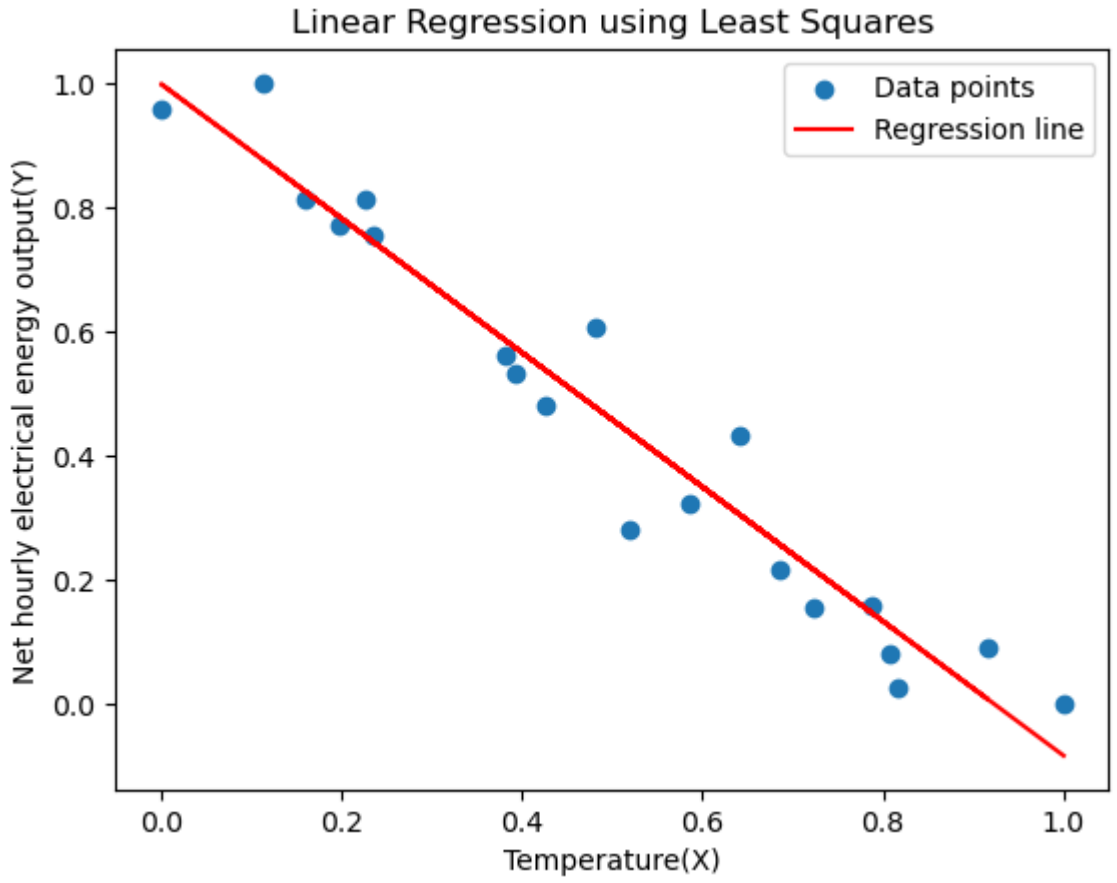
The obtained regression parameters are:

Theta:[-1.08267452 0.99839588], where

Slope:-1.082674517460251 and y-intercept:0.9983958811691596

The regression line with precision to 2 decimal places is: Y = -1.08 * X + 1.00

The plot of the regression line and the data points is:



Part 4: The cost function we chose is the Mean Square Error (MSE) cost function:

$MSE = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \widehat{y_i})^2$, where i iterates over all data points n, y_i is the actual value of data point i, and $\widehat{y_i}$ is the predicted value of data point i. Note, for gradient descent, a factor of $\frac{1}{2}$ is added to $\frac{1}{n}$ in order to facilitate computation when taking the derivatives.

We have implemented this in our code as follows:

```
def costFunction(X, y, theta):
    m = len(y)
    y_pred = X.dot(theta)  # predicted values
    cost = (1 / 2 * m) * np.sum(np.square(y_pred - y))
    return cost
```

We used the Mean Square Error cost function because it is the most commonly used cost function for linear regression. It is easy to interpret it, and determine how the model is performing. Furthermore, MSE is a differentiable and convex function, which helps for a more efficient optimization of the gradient descent.

The partial derivatives of the cost function with respect to each of the regression parameters, bias term (b) and weight (m), are calculated as follows:

$\frac{\delta MSE}{\delta b} = \frac{1}{n}\sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})$, where b is the bias term; and

$\frac{\delta MSE}{\delta m} = \frac{1}{n}\sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})x^{(i)}$, where m is the weight;

where i iterates over all data points n, $x^{(i)}$ represents the input of a data point (i), $y^{(i)}$ represents the true output of a data point (i).

The update rules are as follows:

$\theta_0 = \theta_0 - \alpha\frac{\delta MSE}{\delta b}$, and

$\theta_1 = \theta_1 - \alpha\frac{\delta MSE}{\delta m}$,

where α is the learning rate.

In our code, we have implemented the partial derivatives with the update rule simultaneously as follows:

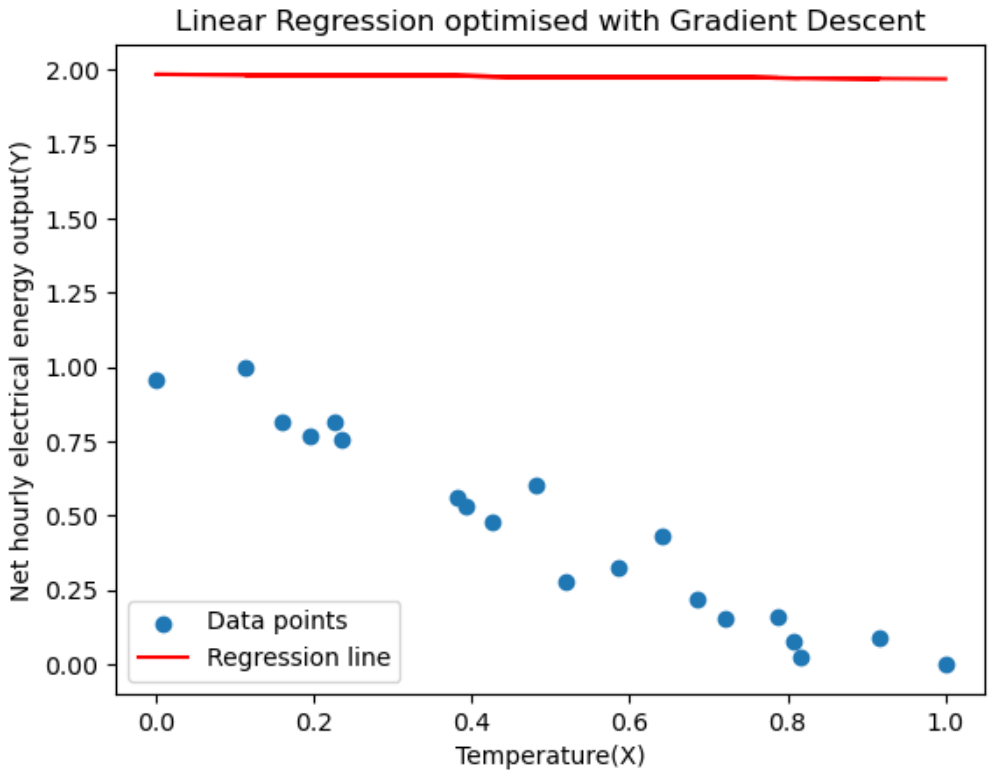
```
for i in range(0, iter):
    y_pred = X.dot(theta)  # predicted values
    loss = y_pred - y  # predicted-actual

    # Partial derivatives of the cost function
    # with respect to each of the regression parameters,
    # and update parameters
    theta[0] = theta[0] - alpha * (np.sum(loss) / m)
    theta[1] = theta[1] - alpha * (np.dot(X_Trans[1], loss) / m)

    # Update Mean Square Error Cost Function
    costHist[i] = costFunction(X, y, theta)
```

Part 5: Our initial regression parameter, θ, is [0,2]. The chosen learning rate $\alpha = 0.01$.

After one iteration of the gradient descent, we have:

Updated theta[0]:-0.015482261387045084
Updated theta[1]:1.9845177386129549
Cost:[484.3801293281862]

Plotting the regression line and the data points:



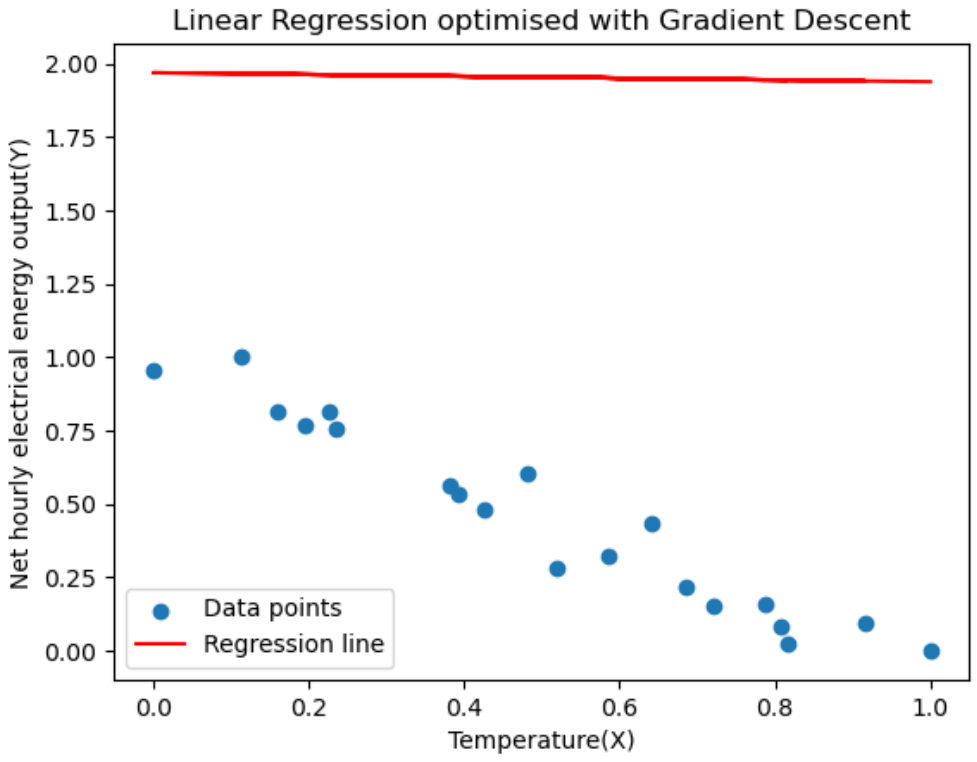
Part 6: After the second iteration of the gradient descent, we have:

Updated theta[0]:-0.03073153313167035

Updated theta[1]:1.9692684668683296

Cost:[469.9727947869858]

Plotting the regression line and the data points:



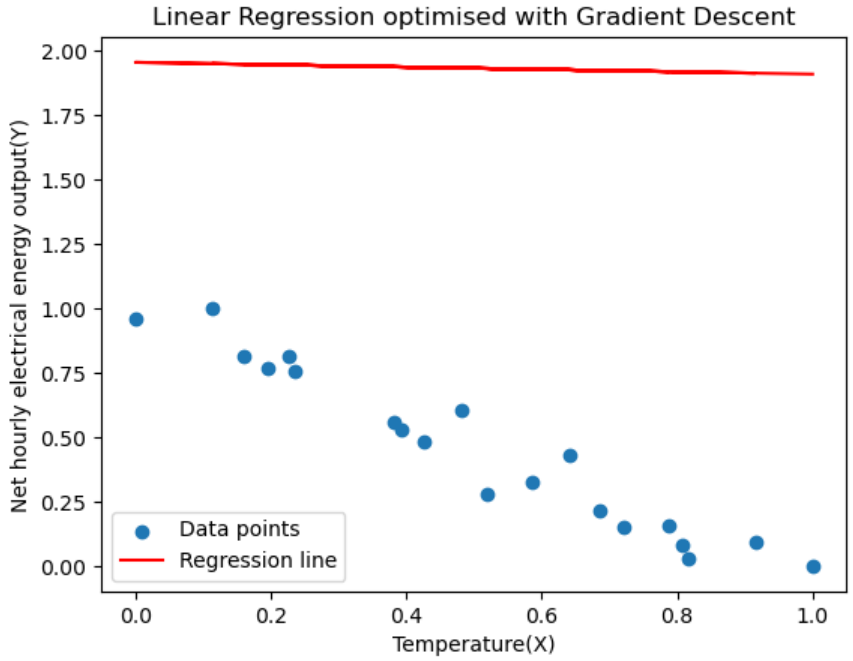
Part 7: After the third iteration of the gradient descent, we have:

Updated theta0:-0.045751321451224666

Updated theta1:1.9542486785487752

Cost:[455.9954360146059]

Plotting the regression line and the data points:

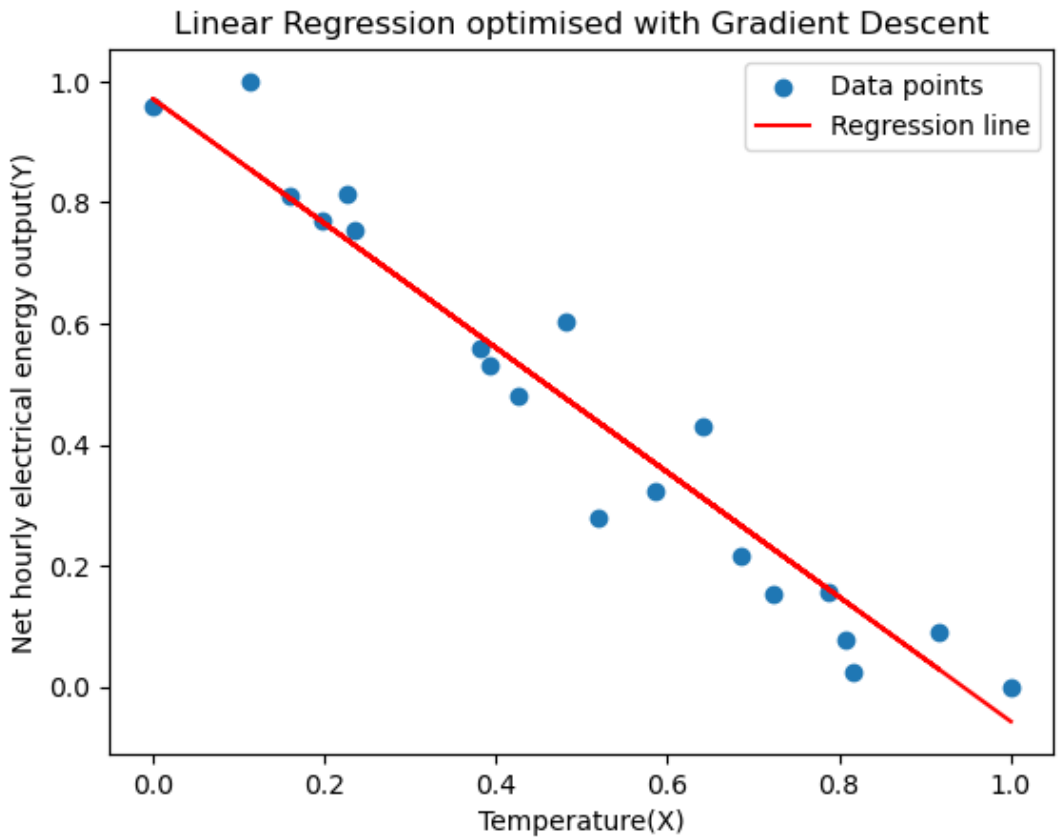


Part 8: It takes 550 more iterations until the regression line fits the data well. If we increase the number of iterations, the plots don't improve, so we can deduce that the gradient descent has converged for 553 (500 + previous 3) iterations. We have:

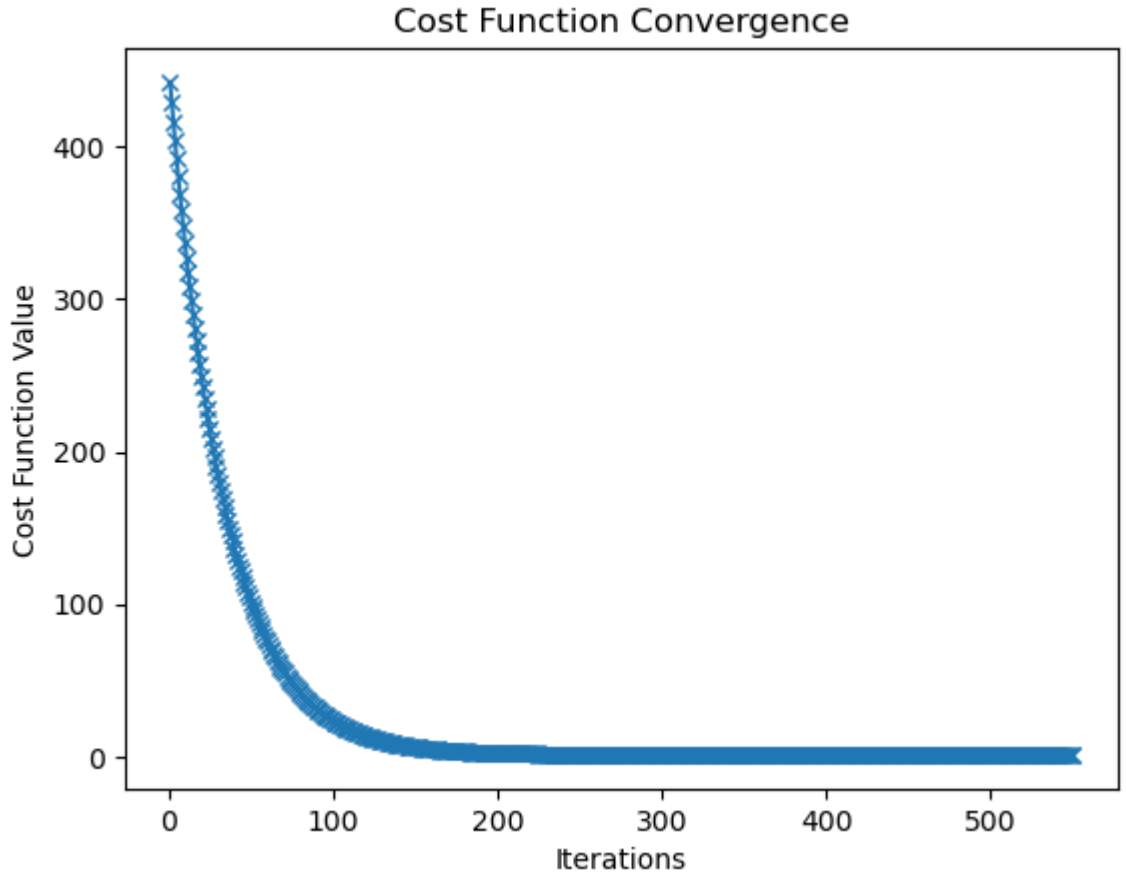
Updated theta0:-1.0285681082557068

Updated theta1:0.9714318917442913

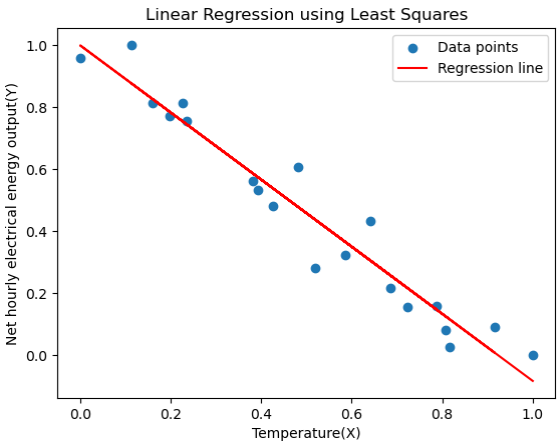
Plotting the regression line and the data points:



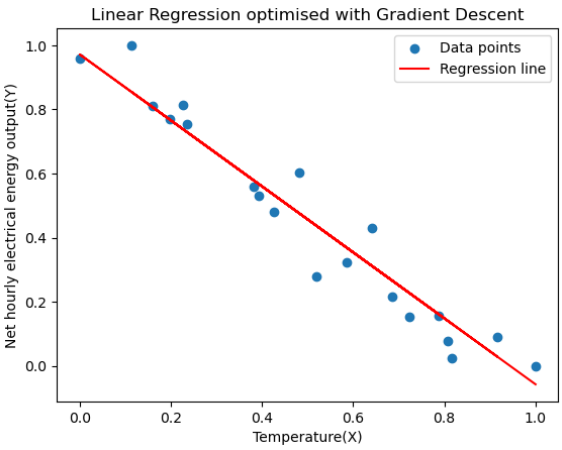
Plotting the values of the cost function for all iterations:



Part 9: Looking at the two plots, we can see that the regression line of the Linear Regression optimized with Gradient Descent fits the data better than the regression line of Least Squares. This means that the Gradient Descent found better parameters that fit the data compared to Least Squares.
Looking at the obtained final values for the parameter theta (θ), we can see that they are different. This could indicate that the Gradient Descent is more sensitive to initialization compared to Least Squares. However, Gradient Descent allows for flexibility of initialization of initial parameters and even learning rate, so one can customize the method based on different datasets and models.
Further, based on the initial parameters and the learning rate, Gradient Descent could require a bigger number of iterations to converge. As a result, this could make this method more computationally expensive than Least Squares, which directly computes the optimal parameters by using the closed-form solution.



Theta:[-1.08267452 0.99839588]



Theta:[-1.0285681082557068 0.9714318917442913]

A.II. Polynomial regression

We are using the data from A.I.2, i.e., the Min-Max normalized data. As explained in part A.I.2, Min-Max normalization preserves the relationships between the values, while being able to scale them such that they are all on a similar scale. This way, each value contributes proportionally, and comparisons and analysis between features is made easier. Furthermore, the gradient descent should be able to converge quicker thanks to this normalization.
We have given J as the cost function:

$J(\theta) = \frac{1}{4n}\sum_{i=1}^{n}(y^{(i)} - h_\theta(x^{(i)}))^4$, where i iterates over all data points n, $x(i)$ represents the input of a data point (i), $y(i)$ represents the true output of a data point (i).

Part 1: The partial derivatives of the cost function with respect to each of the model parameters are computed as follows:

$\frac{\delta J}{\delta \theta_2} = -\frac{1}{n}\sum_{i=1}^{n} x^{(i)2}(y^{(i)} - h_\theta(x^{(i)}))$; and

$\frac{\delta J}{\delta \theta_1} = -\frac{1}{n}\sum_{i=1}^{n} x^{(i)}(y^{(i)} - h_\theta(x^{(i)}))$; and

$\frac{\delta J}{\delta \theta_0} = -\frac{1}{n}\sum_{i=1}^{n} (y^{(i)} - h_\theta(x^{(i)}))$;

where i iterates over all data points n, $x^{(i)}$ represents the input of a data point (i), $y^{(i)}$ represents the true output of a data point (i).
In our code, we have implemented them as follows:

```
d_theta2 = np.sum(loss * (X[:, 0] ** 3)) / m
d_theta1 = np.sum(loss * X[:, 1]) / m
d_theta0 = np.sum(loss) / m
```

Part 2: The update rule for each parameter is computed as follows:

$\theta_0 = \theta_0 - \alpha\frac{\delta J}{\delta \theta_2}$,

$\theta_1 = \theta_1 - \alpha\frac{\delta J}{\delta \theta_1}$,

$\theta_2 = \theta_2 - \alpha\frac{\delta J}{\delta \theta_0}$, where α is the learning rate.

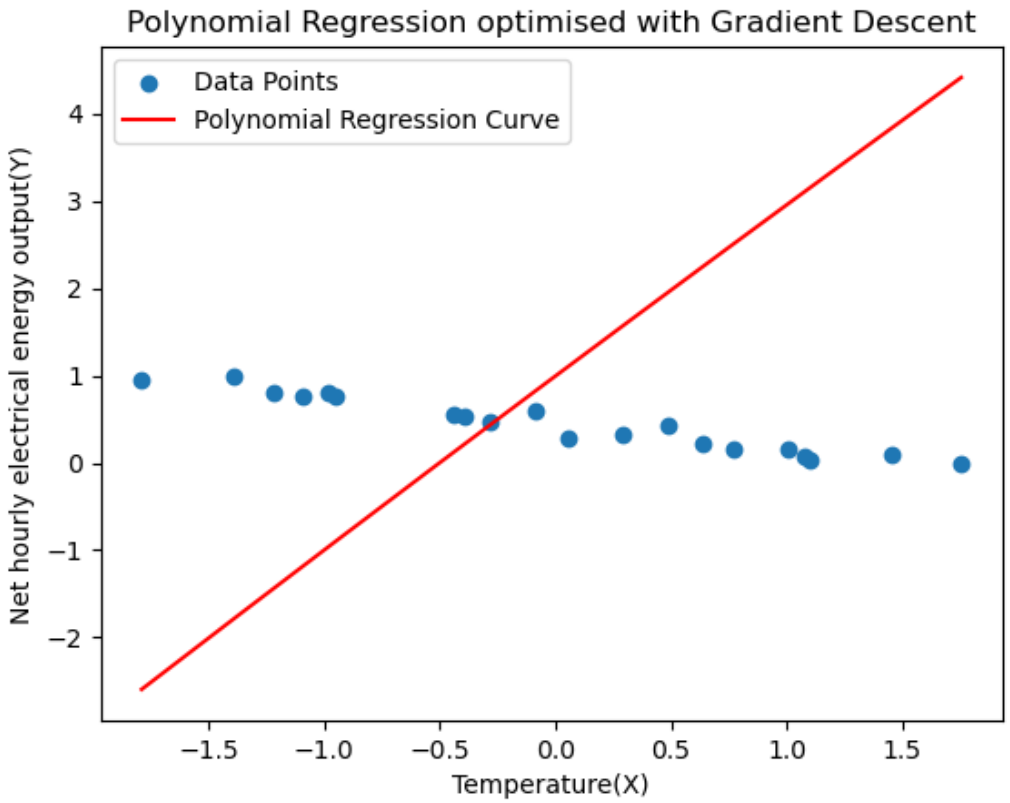
In our code, we have implemented the update rules as follows:

```
theta_poly[0] = theta_poly[0] - alpha_poly * d_theta2
theta_poly[1] = theta_poly[1] - alpha_poly * d_theta1
theta_poly[2] = theta_poly[2] - alpha_poly * d_theta0
```

Part 3: Our initial theta, θ, for polynomial regression: [0, 2, 1]. The chosen learning rate $\alpha = 0.01$.

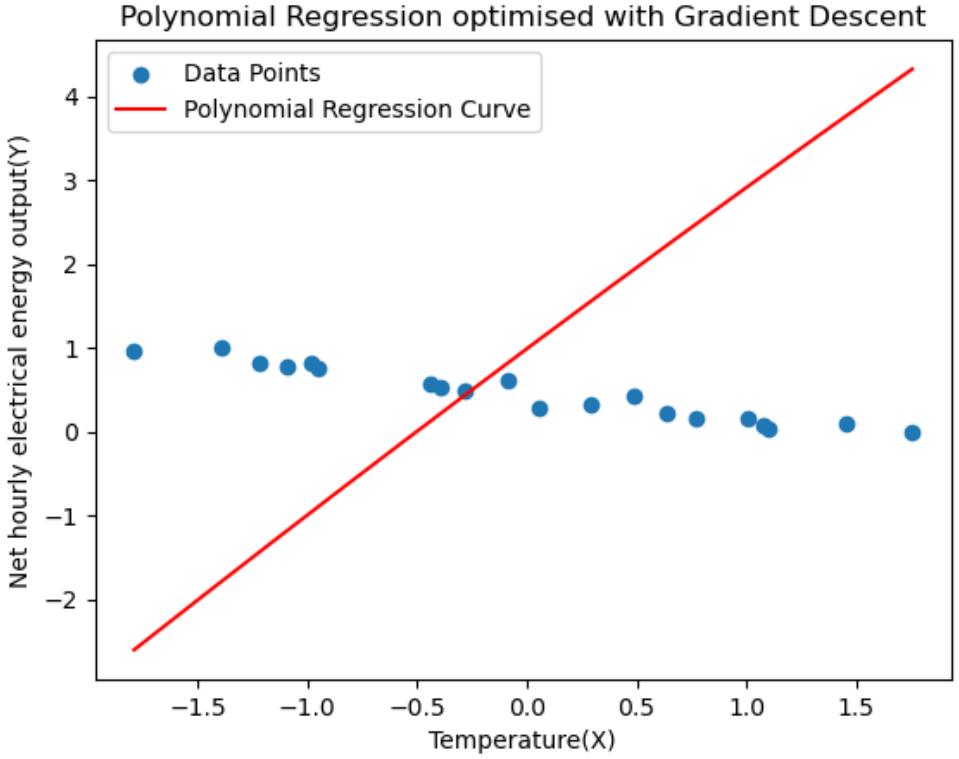
- After 1 iteration, we have:
  Updated theta:[-0.01619719839065062, 1.9769442704730824, 0.9945177386129549]
  Cost:[14.363477901516427]

Plotting the regression line and the data points:



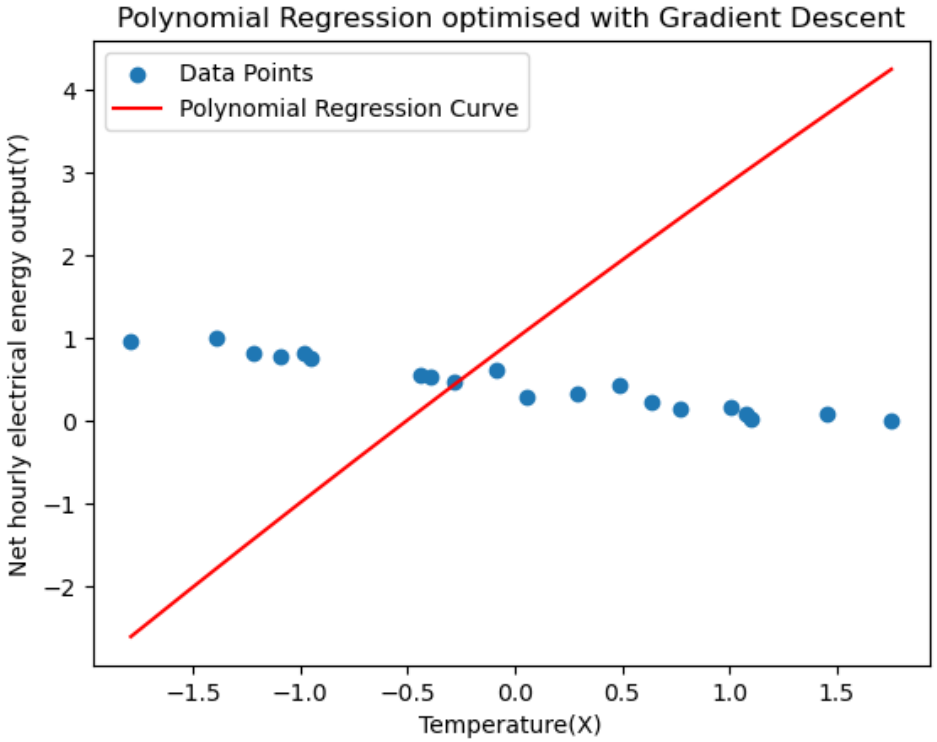
- After a second iteration, we have:
  Updated theta:[-0.030259293636409043, 1.954114349752158, 0.9892522718236868]
  Cost:[13.624074018191372]

Plotting the regression line and the data points:



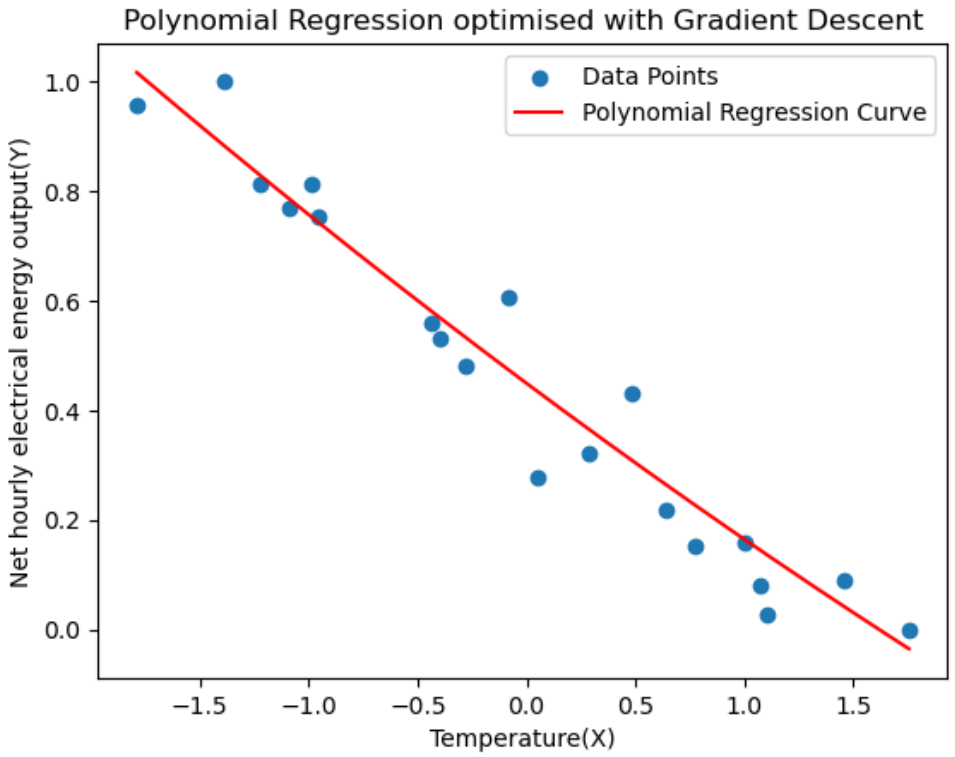
- After a third iteration, we have:
  Updated theta:[-0.04245373185504232, 1.931508605691638, 0.9841800806547689]
  Cost:[12.951172878291727]

Plotting the regression line and the data points:

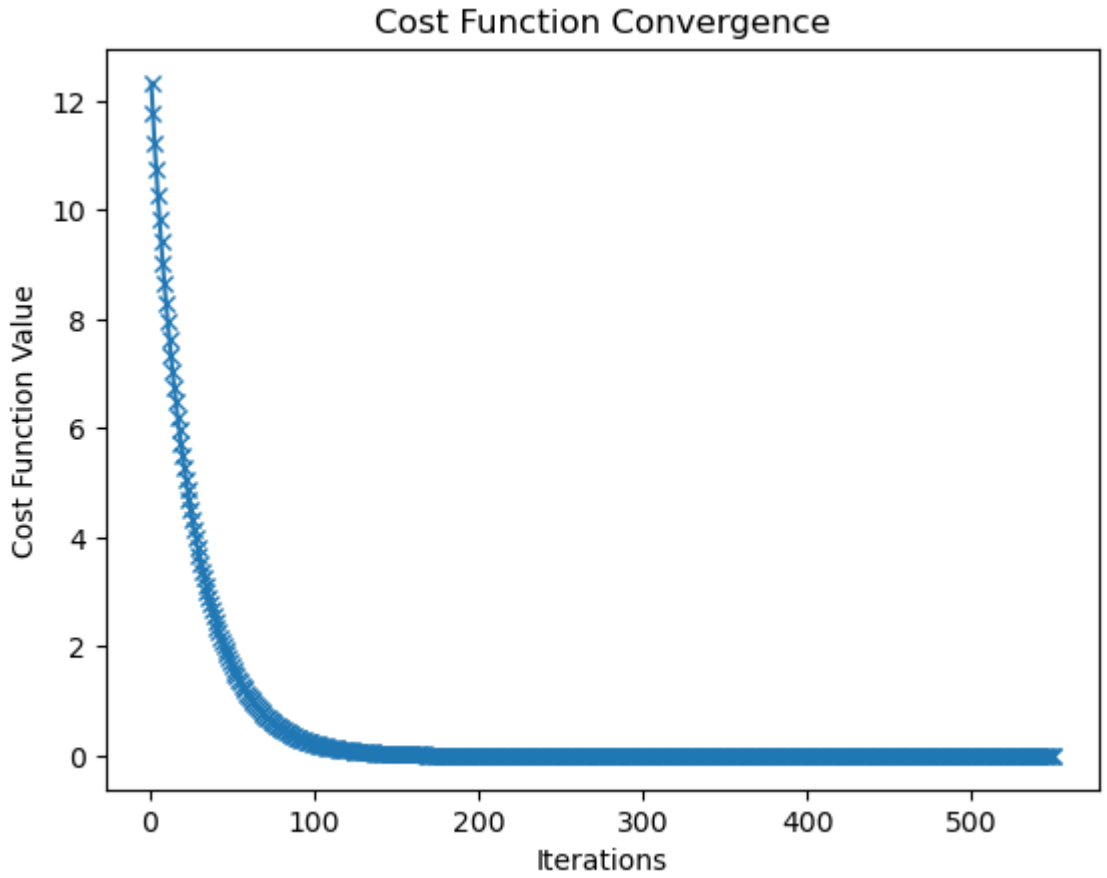


- It takes 550 more iterations until the regression line fits the data well. If we increase the number of iterations, the plots don't improve, so we can deduce that the gradient descent has converged for 553 (500 + previous 3) iterations. We have:
  Updated theta:[0.011590873651061167, -0.2965364595046681, 0.44898909201763565]

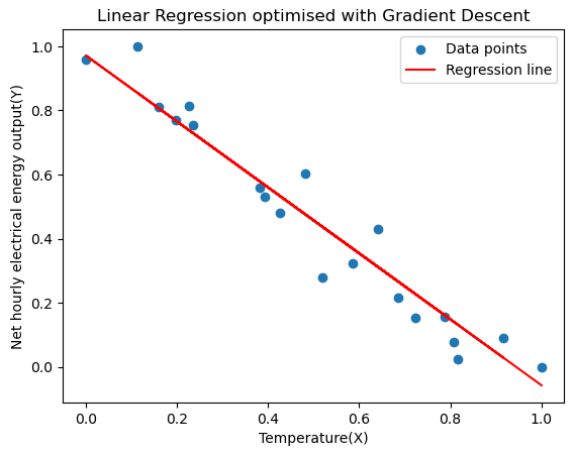
Plotting the regression line and the data points:



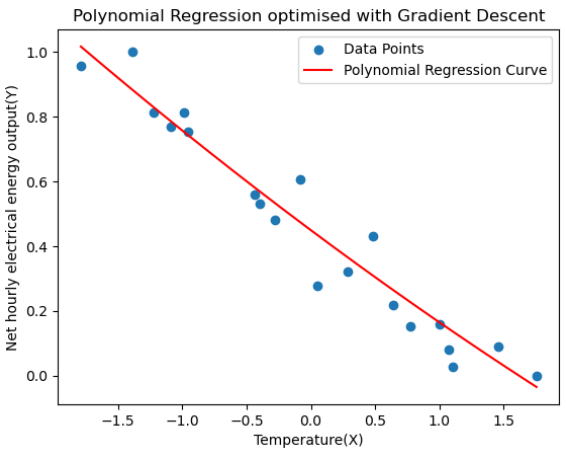
Plotting the values of the cost function for all iterations:



Part 4: Both linear and polynomial regressions model the relationship between the input variable(s) and the output variable. However, linear regression does so as a linear function, which means that we have a straight regression line as seen in the plot on the left. While polynomial regression does so as a polynomial function, which means we can model curvature in data, as seen in the plot on the right.
The degree of the polynomial determines the complexity of the model. As seen from the formulas in this whole part A, linear regression can be viewed as a polynomial of degree 1, while polynomial regression has higher degrees.
Further, from the plots we can see that both linear and polynomial regression fit the data well in 553 iterations.



Initial theta: [0, 2]
Updated theta:[-1.0285681082557068
0.9714318917442913]

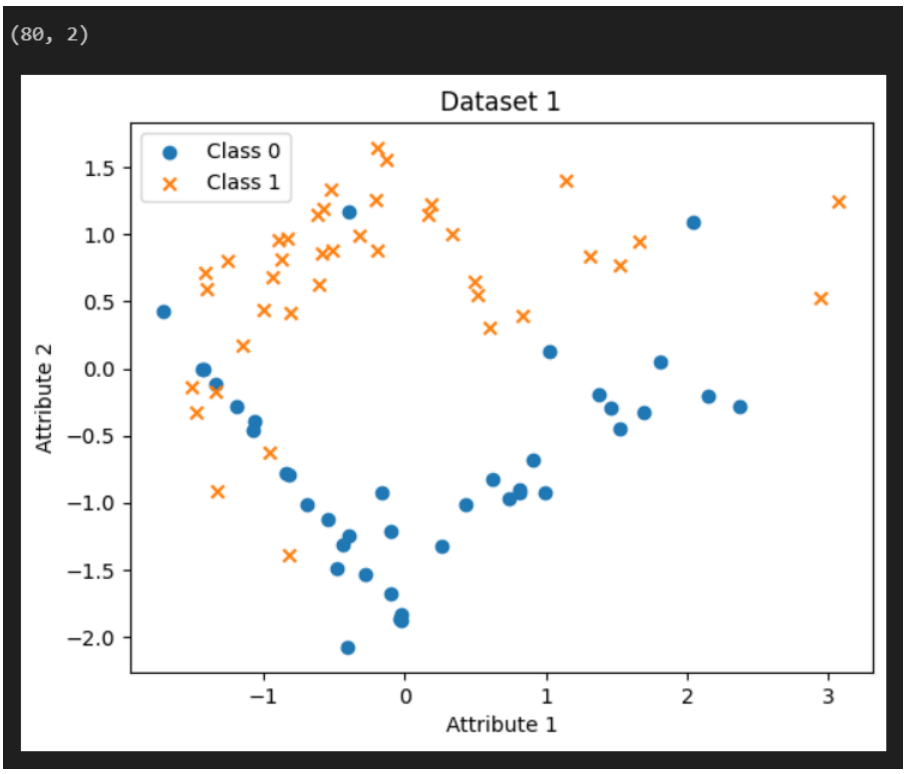


Initial theta: [0, 2, 1]
Updated theta:[0.011590873651061167,
-0.2965364595046681,
0.44898909201763565]

B. Supervised Learning (SL) - Classification

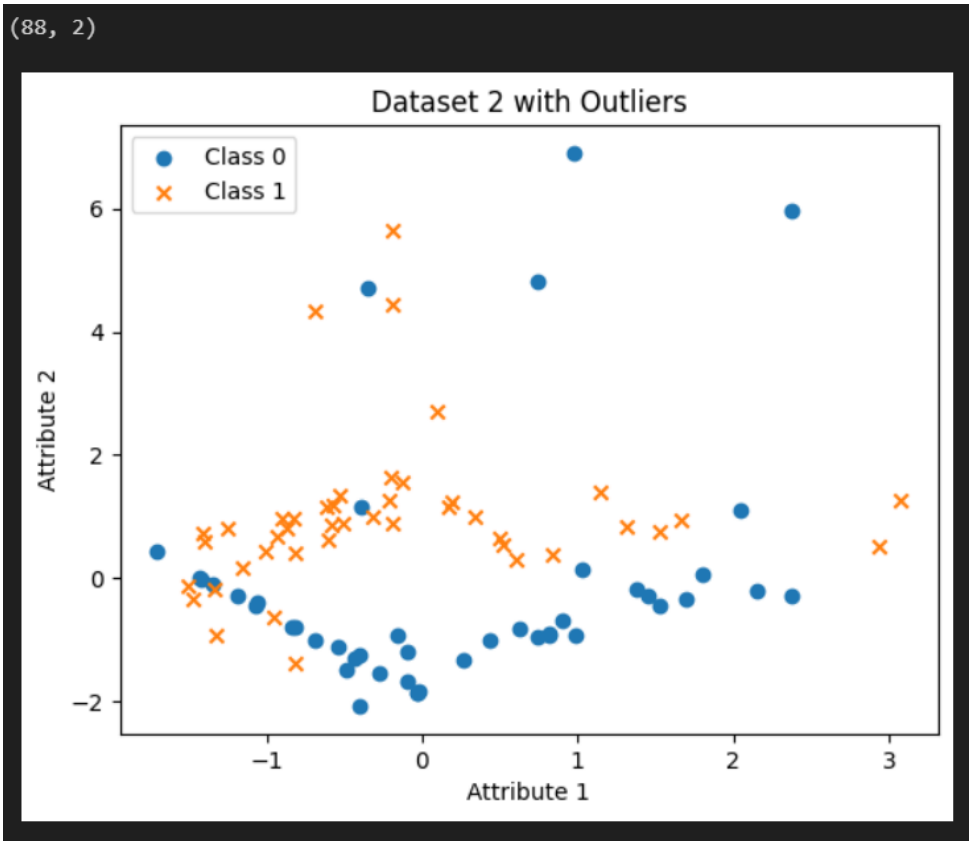
Part 1:

In order to generate a binary classification problem with two continuous real-number attributes, we used scikit-learn's make_classification function. A total of 80 data points are generated, evenly divided into two classes of 40 data points each. To control the dataset's characteristics, key parameters such as the number of informative and redundant features, clusters per class, noise level, and class separation are carefully configured. The random seed is chosen to be reproducible. The resulting dataset clearly distinguishes between the two classes, as shown in a scatter plot with Class 0 and Class 1 denoted by 'o' and 'x,' respectively. This method makes use of scikit-learn's utility functions to quickly generate a synthetic dataset tailored to the given binary classification problem.



Part 2:

When generating Dataset 2 from Dataset 1, four outliers are generated at random for each class using the NumPy np.random.normal function. Outliers for Class 0 are drawn from a normal distribution with a mean of [0, 5] and a standard deviation of [1.5, 1.25]. Similarly, outliers in Class 1 are sampled using a mean of [0, 5] and a standard deviation of [0.4, 1.2]. The resulting Dataset 2 is formed by vertically stacking the original Dataset 1 (D1X and D1y) with these generated outliers. The shape of D2X confirms the addition of eight outliers, and the scatter plot depicts the distribution of the data points, with 'o' representing Class 0, 'x' representing Class 1, and the added outliers contributing to Dataset 2's overall structure.



Part 3:

Both Dataset 1 and Dataset 2 are correctly divided into training and testing data. With a test size of 20%, the train_test_split function ensures that 80% of the data is used for training and 20% is reserved for testing. The use of a random seed (random_state=14) ensures reproducibility, which means that the same split will be obtained if the code is run again. This is critical for evaluating machine learning models consistently.

The printed output summarizes the split data by reporting the shape of the training and test data for both datasets:

The training data for Dataset 1 has shape (64, 2), and the test data has shape (16, 2).
The training data in Dataset 2 has the shape (80, 2), and the test data has the shape (20, 2).
These splits are suitable for training machine learning models on the majority of the data and assessing their performance on a separate, unseen portion. The 80-20 split is a popular choice for balancing a large enough training set for model learning with a meaningful test set for evaluation.

```
Dataset 1: Training data: (64, 2), Test data: (16, 2)
Dataset 2: Training data: (70, 2), Test data: (18, 2)
```

Part 4: For Dataset 1

k-NN (k-Nearest Neighbors):

Training Accuracy: 90.63%

Testing Accuracy: 87.50%

Confusion Matrix: [[8 1]
[1 6]]

The k-NN model performs well on both training and testing datasets, with a relatively high accuracy. According to the confusion matrix, the model misclassified one instance in each class.

Naive Bayes:

Training Accuracy: 85.94%

Testing Accuracy: 87.50%

Confusion Matrix: [[7 2]
[0 7]]

Naive Bayes also performs well, with a high level of testing accuracy. The confusion matrix reveals that the model misclassified two items, both in the first class.

Decision Trees:

Training Accuracy: 98.44%

Testing Accuracy: 81.25%

Confusion Matrix: [[7 2]
[1 6]]

The Decision Trees model has a high training accuracy but a lower testing accuracy, indicating potential overfitting. According to the confusion matrix, the model struggled with one instance in the second class during testing.

Random Forrest:

Training Accuracy: 95.31%

Testing Accuracy: 87.50%

Confusion Matrix: [[8 1]
[1 6]]

Random Forests, like k-NN, perform well on both training and testing datasets, with only one misclassification in each class based on the confusion matrix.

In summary, k-NN and Random Forests perform well, while Naive Bayes performs reasonably well. Despite their high training accuracy, Decision Trees appear to struggle with generalization to testing data, possibly indicating overfitting.

```
k-NN - Training Accuracy: 0.90625, Testing Accuracy: 0.875
Confusion Matrix:
[[8 1]
 [1 6]]
Naive Bayes - Training Accuracy: 0.859375, Testing Accuracy: 0.875
Confusion Matrix:
[[7 2]
 [0 7]]
Decision Trees - Training Accuracy: 0.984375, Testing Accuracy: 0.8125
Confusion Matrix:
[[7 2]
 [1 6]]
Random Forests - Training Accuracy: 0.953125, Testing Accuracy: 0.875
Confusion Matrix:
[[8 1]
 [1 6]]
```

For Dataset 2:

k-NN (k-Nearest Neighbors):

Training Accuracy: 90.00%

Testing Accuracy: 77.78%

Confusion Matrix: [[6 0]
[4 8]]

The training accuracy of the k-NN model is high, but the testing accuracy is slightly lower. According to the confusion matrix, the model correctly classified all instances in the first class but struggled with misclassifications in the second.

Naive Bayes:

Training Accuracy: 75.71%

Testing Accuracy: 77.78%

Confusion Matrix: [[4 2]

[2 10]]

Naive Bayes performs reasonably well, with comparable accuracy on both training and testing datasets. The confusion matrix reveals that the model misclassified some items in both classes.

Decision Trees:

Training Accuracy: 98.57%

Testing Accuracy: 77.78%

Confusion Matrix: [[6 0]
[4 8]]

Decision Trees, like the k-NN model, have a high training accuracy but a lower testing accuracy. The confusion matrix indicates that correctly classifying instances in the second class during testing may be difficult.

Random Forrest:

Training Accuracy: 97.14%

Testing Accuracy: 77.78%

Confusion Matrix: [[6 0]
[4 8]]

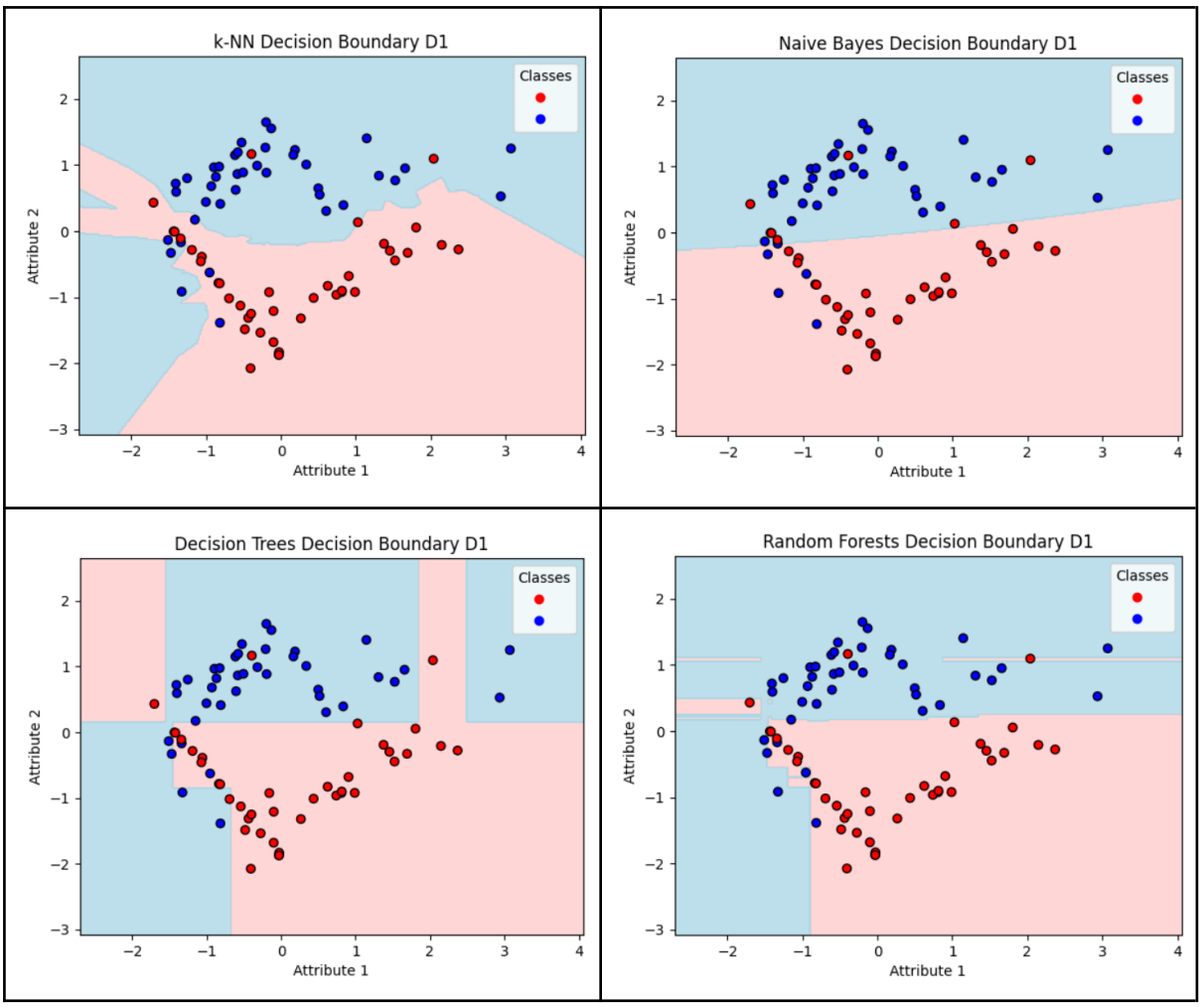
Random Forests perform well on training data but have slightly lower accuracy on testing data. The confusion matrix reveals some errors in the second class.

In summary, the second set of models performs well in training but struggles in generalizing to testing data, particularly correctly classifying instances in the second class for k-NN, Decision Trees, and Random Forests. On both training and testing datasets, Naive Bayes performs moderately.

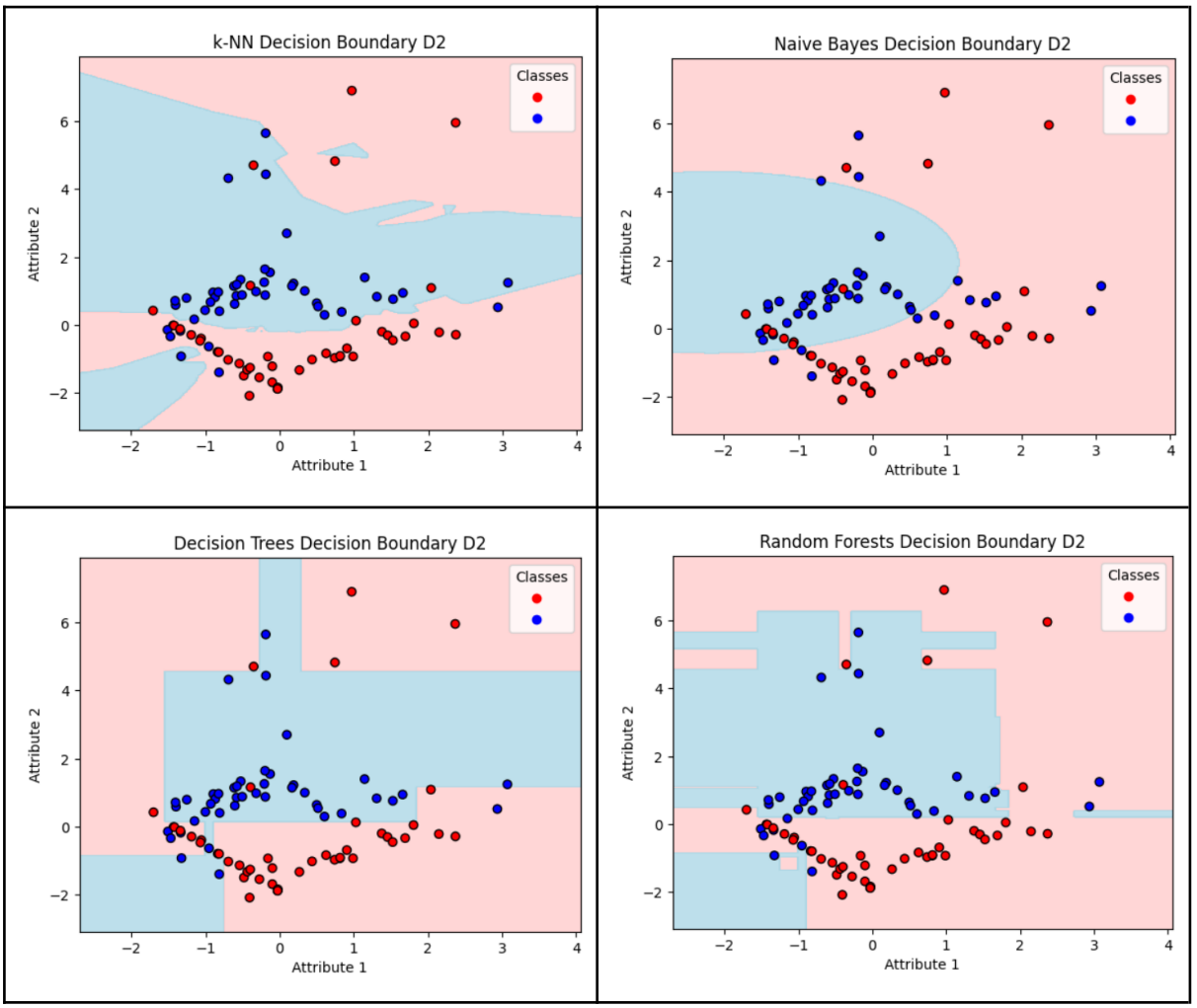
```
k-NN - Training Accuracy: 0.9, Testing Accuracy: 0.7777777777777778
Confusion Matrix:
[[6 0]
 [4 8]]
Naive Bayes - Training Accuracy: 0.7571428571428571, Testing Accuracy: 0.7777777777777778
Confusion Matrix:
[[ 4  2]
 [ 2 10]]
Decision Trees - Training Accuracy: 0.9857142857142858, Testing Accuracy: 0.7777777777777778
Confusion Matrix:
[[6 0]
 [4 8]]
Random Forests - Training Accuracy: 0.9714285714285714, Testing Accuracy: 0.7777777777777778
Confusion Matrix:
[[6 0]
 [4 8]]
```

Part 5:

Decision Boundaries For Dataset 1:



Decision Boundaries For Dataset 2:



Part 6:

There are several similarities and differences between the results obtained for Dataset 1 and Dataset 2 using different classification models (k-NN, Naive Bayes, Decision Trees, and Random Forests):

Similarities:

Training Accuracy: k-NN, Decision Trees, and Random Forests consistently achieved high training accuracies for both datasets, indicating that these models could fit the training data well.

Testing Accuracy: A common trend across models is to achieve relatively lower testing accuracy compared to training accuracy. This suggests that the models may have difficulty generalizing to new data.

Dissimilarities:

Testing Accuracy Variation: While the testing accuracies of k-NN, Naive Bayes, Decision Trees, and Random Forests are similar for Dataset 1, there is more variation in the testing

accuracies for Dataset 2. This suggests that the inclusion of outliers in Dataset 2 may have affected the models' generalization ability differently.

Boundaries of Decision:

k-NN: k-NN is based on local neighborhoods and has decision boundaries that closely follow the data distribution. Outliers may influence the decision boundary in Dataset 2, resulting in a less smooth boundary.

Naive Bayes: Naive Bayes assumes feature independence given the class. It has linear decision boundaries and is less affected by outliers.

Decision Trees: Decision Trees define decision boundaries in a hierarchical manner. Decision trees perform well in Dataset 1, where classes are well separated. However, in Dataset 2, outliers influence the decision boundaries, resulting in misclassifications.

Random Forests: Because they are an ensemble of Decision Trees, Random Forests are similar to Decision Trees. They improve robustness by combining predictions from multiple trees. However, they are still sensitive to outliers in Dataset 2.

In conclusion, the differences in model testing accuracies and the impact of outliers on decision boundaries highlight the importance of taking into account the characteristics of the dataset as well as the assumptions of each model. Fine-tuning hyperparameters and addressing outliers could improve the overall performance of these models on Dataset 2.

Part C - Unsupervised Learning (UL)

Clustering as an Unsupervised Learning Task:

- Task Chosen: Clustering, specifically the K-Means algorithm.
- Why Clustering?: It's a foundational method in machine learning for finding patterns and groupings in data without pre-labeled classes or examples. its wide range of applications and the interesting challenges it presents, such as determining the number of clusters, dealing with different shapes and sizes of clusters, and the sensitivity to the scale of the data.
- Applications: Market segmentation, social network analysis, organization of large computer clusters, bioinformatics, image segmentation, anomaly detection and more.

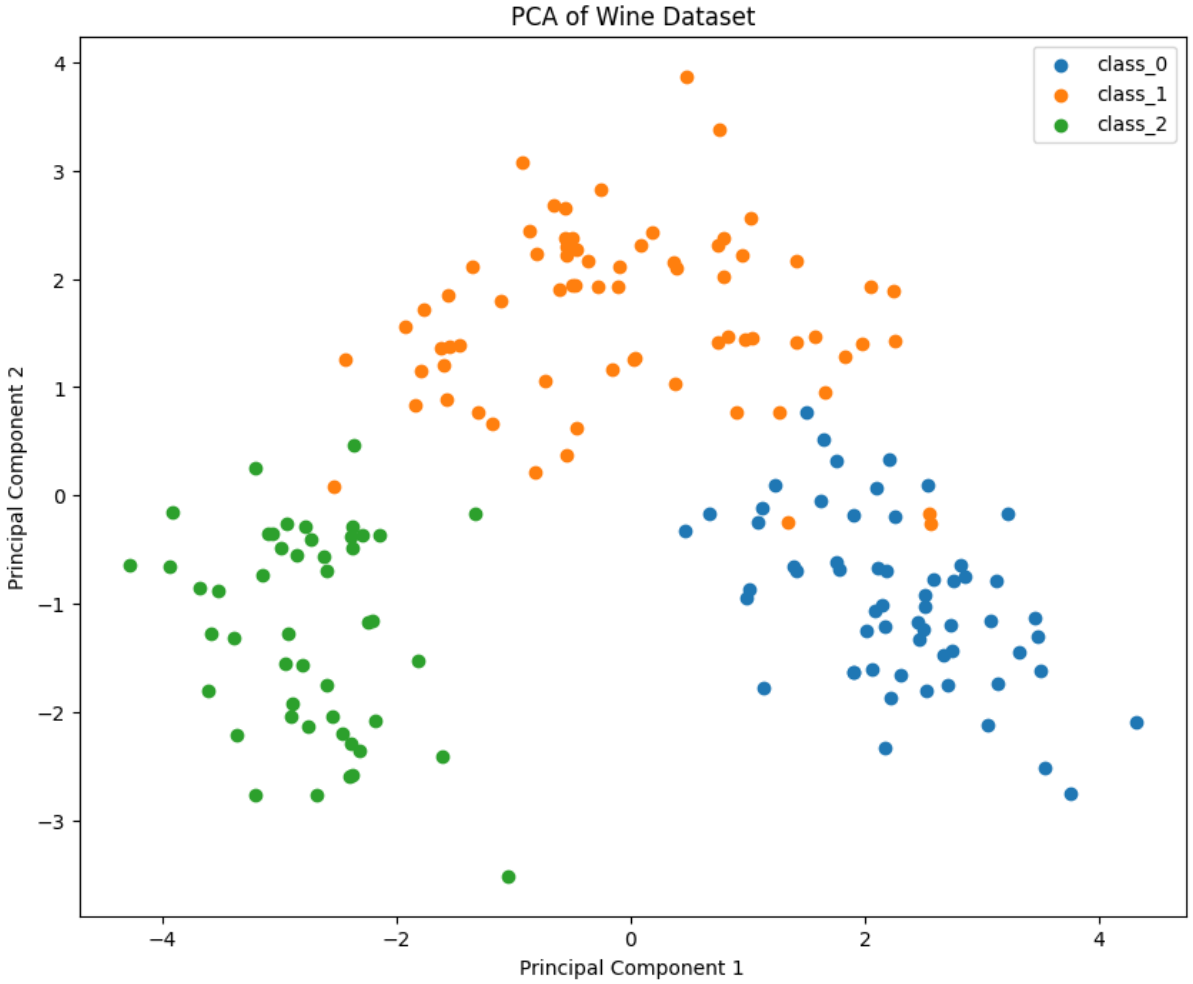
What do you need to know to perform clustering:

- Understanding the data.
- Preprocessing.
- Choosing the right algorithm.
- Determining number of Clusters.
- Algorithm Parameters
- Robustness and stability
- Visualization.

Evaluation Metrics involved in clustering:

- Silhouette Coefficient: Measures and compares the similarity of an object to its own cluster and other clusters. A high silhouette value indicates that the object is well-matched to its own cluster and poorly matched to neighboring clusters.
- Davies-Bouldin Index: The average 'similarity' between clusters, where similarity is a measure that compares the distance between clusters with the size of the clusters themselves. A lower Davies-Bouldin index indicates better clustering.
- Calinski-Harabasz Index: It is the ratio of the sum of between-clusters dispersion and of within-cluster dispersion for all clusters. The higher the score, the better the model is defined.
- Dunn Index: The ratio between the smallest distance between observations not in the same cluster to the largest intra-cluster distance. Higher values indicate better clustering.
- Adjusted Rand Index (ARI): Measures the similarity between two assignments, ignoring permutations and with chance normalization. A score close to 0.0 indicates random assignments, and a score close to 1 indicates perfect agreement between assignments.
- We load the dataset using load_wine() from sklearn.datasets.
- The dataset is then scaled using StandardScaler to ensure that each feature contributes equally to the analysis.
- The Wine Recognition dataset has been successfully visualized using PCA(Principal Component Analysis) for dimensionality reduction. In the scatter plot, you can see the data points reduced to two principal components, with different colors representing

different types of wine.



- The PCA process simplifies the data, making patterns more evident for clustering algorithms to detect.
- This visualization can help in understanding the separation and overlap between different types of wines based on their chemical characteristics.
- This transformation facilitates visualization in a 2D plot.
- A scatter plot is created to visualize the wines in the reduced space, with each point representing a wine and its color representing its class.
- The PCA plot suggests that there may be some natural clustering corresponding to the different types of wine.

```
      alcohol  malic_acid       ash  alcalinity_of_ash  magnesium  \
PC1  0.144329   -0.245188 -0.002051          -0.239320   0.141992
PC2 -0.483652   -0.224931 -0.316069           0.010591  -0.299634

     total_phenols  flavanoids  nonflavanoid_phenols  proanthocyanins  \
PC1       0.394661    0.422934             -0.298533         0.313429
PC2      -0.065040    0.003360             -0.028779        -0.039302

     color_intensity       hue  od280/od315_of_diluted_wines   proline
PC1        -0.088617  0.296715                      0.376167  0.286752
PC2        -0.529996  0.279235                      0.164496 -0.364903
```

Discussion on the Dataset and Operations Performed:

- The dataset was scaled using StandardScaler to normalize the features, ensuring each feature contributes equally to the analysis.
- PCA was used to reduce the dataset to two principal components for visualization, which helps us understand the inherent clustering of the data.

Motivation for Choice:

- The Wine dataset proves to be an optimal selection for the current undertaking. It boasts a moderate size and complexity, encompassing an array of numerical characteristics that represent diverse physicochemical properties of wine.
- It can be widely utilized for classification and clustering tasks, this dataset yields easily interpretable results and can be effectively visualized after dimensionality reduction.
- PC1 correlates positively with total_phenols, flavanoids, od280/od315_of_diluted_wines, and proline, which are all related to the phenolic compounds and the antioxidant capacities of the wines. It suggests that PC1 may be capturing variance associated with the biochemical properties of wine that are typically linked to quality.
- PC2 is negatively correlated with alcohol and color_intensity, which may indicate that this component captures variance associated with the physical properties of wine(such as its alcohol content and depth of color).

K-Means Clustering:

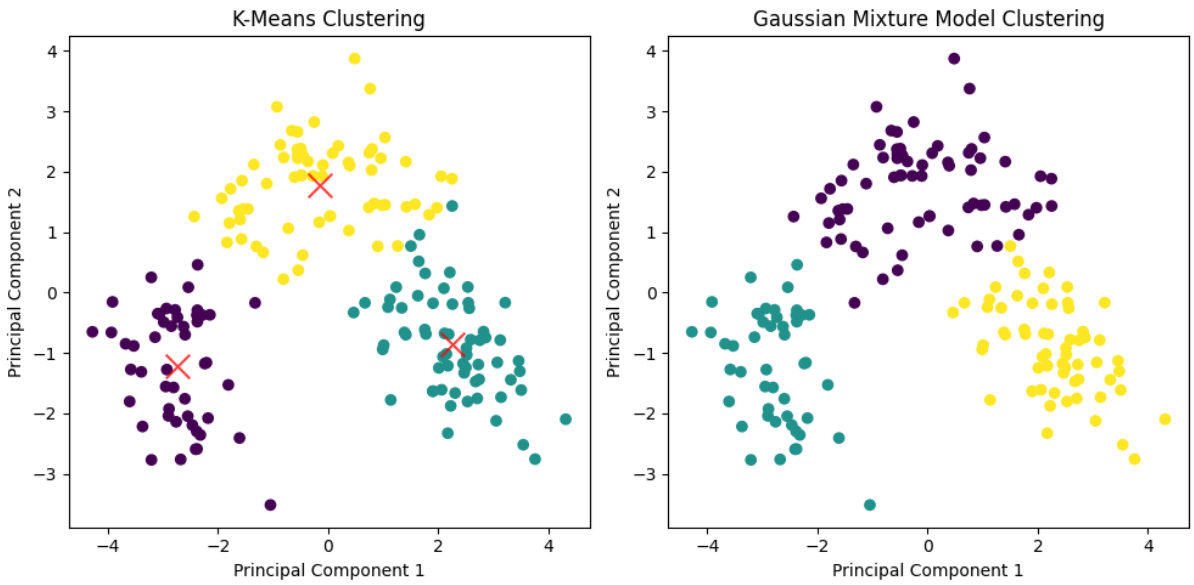
- Motivation : K-Means is a popular and straightforward clustering algorithm that partitions the dataset into K distinct, non-overlapping subgroups or clusters. It's a good starting point due to its simplicity and efficiency.
- Procedure : It partitions 'n' observations into 'k' clusters in which each observation belongs to the cluster with the nearest mean. This results in a partitioning of the data space into Voronoi cells.
- Optimization : The algorithm iteratively assigns points to the nearest cluster center and then computes the center of each cluster repeatedly.The 'k-means++' initialization method is used to speed up convergence and the algorithm repeats the process until the assignment of instances to clusters no longer changes.
- Implementation : The KMeans class is used to cluster data, and has some important parameters that can be set. The number of clusters is determined by the n_clusters parameter, and the init parameter sets the method used to initialize the algorithm. The default method, 'k-means++', is often a good choice. Two other key parameters are n_init and max_iter. The n_init parameter determines how many times the algorithm will be run with different centroid seeds, which can improve clustering quality. The

max_iter parameter sets the maximum number of iterations the algorithm can run for a single run.

- Considerations : The number of clusters K needs to be set beforehand, which can be determined using methods like the Elbow method or the Silhouette score. K-Means is sensitive to the scale of the data, hence preprocessing steps like normalization are important.

Gaussian Mixture Models (GMM):

- Motivation : GMM is a probabilistic model that assumes all the data points are generated from a mixture of several Gaussian distributions with unknown parameters. This is particularly useful for the Wine dataset if we believe that the boundaries between different types of wine are not clearly defined and can overlap.
- Procedure : GMM accommodates mixed membership. It has a key advantage in which it can incorporate the covariance between data points into its model. This allows for the identification of more complex clusters.
- Optimization : To estimate the mean and variance of each Gaussian cluster and maximize the likelihood of the data points, the GMM (Gaussian Mixture Model) uses the expectation-maximization (EM) algorithm. First, the algorithm calculates the probabilities that each data point belongs to each cluster during the expectation step. Then, during the maximization step, the model parameters are updated based on these probabilities.
- Implementation : In scikit-learn the GaussianMixture class have key parameters including n_components for the number of mixture components, covariance_type for the type of covariance to use (full, tied, diag, spherical), n_init for the number of initializations to perform, and max_iter for the maximum number of iterations.
- Considerations: GMM can fit more complex cluster shapes since each mixture component can freely define its covariance structure. However, it is computationally more intensive than K-Means and also requires the number of mixture components (clusters) to be specified a priori.



In the visualizations:

- The left plot shows the clusters as found by the K-Means algorithm, with the red 'x' marks representing the centroids of the clusters.
- The right plot illustrates the clusters according to the GMM. GMM does not explicitly provide cluster centers because it uses probability distributions to assign points to clusters.

Conclusion:

- K-Means is useful for clearly separating data into groups, while GMM is better suited to identifying overlapping, probabilistic-based groupings. The choice between which model to use should be based on the data distribution and specific clustering task requirements.
- Using the scikit-learn library in Python, both K-Means and GMM can be implemented and compared. Analyzing the results from both models provides insights into the structure of the Wine dataset and helps determine which model better captures the natural groupings within the data.
- Both algorithms identified clusters that did not exactly match the true wine varieties, but provided insights into how wines might group based on their chemical composition.

Limitations:

- The linear method of PCA was used may not capture all the dataset's nuances.
- Only a set of number of clusters was chosen to match the known wine classes, which may not reflect the natural groupings in the data.
- The interpretability of the clusters is limited due to the lack of domain expertise .

Future Improvements:

- Experimenting with non-linear dimensionality reduction techniques for potentially better clustering results.
- Collaborate with domain experts for deeper insights into the clustering results.
- Explore more advanced models and validate findings with additional external data.

Although the clustering provided some useful initial findings, it's worth noting that there is still room for a more nuanced approach that could lead to richer interpretations and more accurately defined clusters.

References:

Nicholls, Rowan C., (n.a.). [Data set]. GitHub.

https://rowannicholls.github.io/python/data/sklearn_datasets/wine.html

Patel, E., & Kushwaha, D. S. (2020). Clustering Cloud Workloads: K-Means vs Gaussian Mixture Model. *Procedia Computer Science*, *171*, 158–167. https://doi.org/10.1016/j.procs.2020.04.017

Wadibhasme, Bhavesh, (2021). [Data set]. Kaggle.

https://www.kaggle.com/code/bhavesh302/pca-on-wine-dataset