Clustering is an unsupervised machine learning task in which the goal is to partition points within a dataset into distinct groups (or clusters) such that elements within the same group are similar to each other. K-means is an example of a hard-clustering method which means that each point can only be assigned to a single cluster and there are hard boundaries between clusters.

Though we told to use k=7 as value but if I was to justify or pick, this is what I will do…

A graph with a line

Description automatically generated

K-Means is sensitive to the initial placement of centroids, as demonstrated by running the algorithm multiple times with n\_init=1. The sum of squared distances (SSD) varied significantly across different random initializations. However, when n\_init=10 was used, the algorithm automatically selected the best initialization, resulting in the lowest SSD. This demonstrates the importance of using multiple initializations to ensure robust clustering.  
The effect of the n\_init parameter in K-Means clustering was evaluated by comparing the **Sum of Squared Distances (SSD)** across different values of n\_init. SSD measures the compactness of clusters by summing the squared Euclidean distances between data points and their respective cluster centroids. A lower SSD indicates better clustering, as data points are more tightly grouped within their clusters.

The results show that:

1. Using a single initialization (n\_init=1) results in a high SSD (~700,000), highlighting the sensitivity of K-Means to poor random centroid placement.
2. Increasing n\_init to 5 or 10 significantly reduces SSD, as the algorithm selects the best initialization from multiple random trials.
3. Beyond n\_init=10, the SSD stabilizes, suggesting diminishing returns for higher values. This balance highlights that n\_init=10 achieves robust clustering results efficiently.

K-Means clustering is sensitive to the initial placement of centroids, which are chosen randomly at the start of the algorithm. Poor initial centroids can lead to suboptimal clustering (local optima). To address this, the n\_init parameter allows the algorithm to run multiple times with different random initializations of centroids. For example, with n\_init=10, the algorithm performs 10 runs, each with a new initialization, and selects the clustering result with the lowest Sum of Squared Distances (SSD).

Using a fixed random\_state ensures that the sequence of random initializations is reproducible across runs, while still allowing the algorithm to test multiple centroid placements. This improves the robustness of K-Means and ensures that the best initialization is consistently chosen

A graph with a blue line

Description automatically generated

The default Euclidean distance metric was used in K-Means because it aligns with the algorithm's objective of minimizing the variance within clusters. This alignment ensures that clusters are compact and well-separated. Experiments comparing Euclidean distance with alternative metrics (e.g., Manhattan distance) show that K-Means achieves lower SSD and higher Silhouette Scores with Euclidean distance, validating its suitability for numerical datasets.

Euclidean Distance:

SSD: 7738681547.12, Silhouette Score: 0.33

Manhattan Distance:

SSD: 14893455.94, Silhouette Score: 0.28

The tolerance (tol) parameter in K-Means determines the precision of convergence by setting the threshold for acceptable changes in SSD between iterations. To evaluate its impact, experiments were conducted using tolerances ranging from 10−210^{-2}10−2 to 10−1010^{-10}10−10. The results show that:

1. **Higher Tolerances (e.g., 10−210^{-2}10−2)**: The algorithm converges quickly (fewer iterations), but at the cost of higher SSD, indicating less optimal clustering.
2. **Stricter Tolerances (e.g., 10−610^{-6}10−6 to 10−1010^{-10}10−10)**: The algorithm takes significantly more iterations to converge, but the improvement in SSD becomes negligible beyond 10−610^{-6}10−6.

The default tolerance of 10−410^{-4}10−4 achieves a good balance, allowing the algorithm to efficiently converge to compact and stable clusters without unnecessary computational cost.

A graph of a graph

Description automatically generated with medium confidence

Although the default max\_iter=300 is sufficient for most datasets, experiments demonstrate that the algorithm consistently converges within ~80 iterations for this dataset. Reducing max\_iter to 100 achieves the same clustering quality (SSD) while minimizing unnecessary computational overhead. Therefore, max\_iter=100 is an efficient and effective choice for this dataset.  
A graph of a graph

Description automatically generated with medium confidence

The init\_params parameter in GMM was set to random to allow flexible initialization of cluster parameters (means, covariances, and weights). Unlike the default kmeans initialization, which assumes spherical clusters, random initialization avoids potential biases and enables the model to explore diverse starting points. This choice is particularly useful for datasets with non-spherical or complex clusters, and combined with a fixed random\_state, it ensures reproducibility without limiting the variability of initial parameters

**How to Use This in the Report**

**Sample Explanation**

"Clustering errors were calculated by counting the number of pairs of datapoints with the same true class label but assigned to different clusters by the algorithm. The results show:

1. **K-Means**: Produced fewer errors compared to the random baseline, demonstrating its ability to form meaningful clusters.
2. **GMM**: Made fewer errors than K-Means, leveraging its probabilistic framework to capture more complex cluster structures.
3. **Random Baseline**: Expectedly performed the worst, as it assigns clusters randomly without considering the data structure."

The error counts highlight the strengths and weaknesses of each clustering algorithm:

1. **K-Means** made **15,430,954 errors**, which is fewer than the Random Baseline but more than GMM. This is because K-Means minimizes Euclidean distances and assumes spherical clusters. If the true clusters are non-spherical or overlapping, K-Means struggles to assign datapoints correctly, especially near cluster boundaries.
2. **GMM** performed the best, making **13,697,715 errors**, due to its ability to model elliptical cluster shapes using covariance and its probabilistic approach to soft clustering. This allows GMM to better handle overlapping clusters and datapoints near boundaries.
3. The **Random Baseline** made the most errors (**16,121,145**) because it assigns cluster labels without considering the structure of the data. The high error count reflects the lack of meaningful clustering.

These results demonstrate that structured clustering algorithms like K-Means and GMM significantly outperform random assignment. Furthermore, GMM's flexibility and probabilistic framework make it the most effective choice for datasets with complex cluster geometries.

K-Means Errors: 15430954 / 18807759

GMM Errors: 13697715 / 18807759

Random Baseline Errors: 16121145 / 18807759

K-Means Errors: 12792643 / 18807759

GMM Errors: 12794219 / 18807759

Random Baseline Errors: 16121145 / 18807759

K-Means Errors: 12792643 / 18807759

GMM Errors: 12794219 / 18807759

Random Baseline Errors: 16121145 / 18807759

**ample Explanation**

"A comparison was conducted between K-Means with k-means++ initialization and random initialization. The results showed:

* **K-Means with k-means++ Initialization** achieved a lower SSD and higher Silhouette Score compared to random initialization. This demonstrates that k-means++ improves centroid placement, leading to tighter and better-separated clusters.
* The random initialization method resulted in less compact clusters and lower clustering quality, as evidenced by higher SSD and lower Silhouette Score.

K-Means Errors: 12426423 / 18807759

The impact of the maximum number of iterations (max\_iter) on K-Means clustering was analyzed. Results showed that the algorithm converged within ~20 iterations for all tested values of max\_iter, provided that the tolerance (tol) was set to 1e-4. Increasing max\_iter beyond 20 had no effect on the number of iterations or clustering quality (as measured by SSD). This indicates that the algorithm's convergence is primarily governed by the tolerance parameter, making excessively large max\_iter values unnecessary.

Different covariance structures were tested to identify the best fit for the data. The 'full' covariance type provided the lowest BIC and was selected for clustering.

**Interpretation of Results**

1. **RMSE Similarity**:
   * The training and test RMSE values being nearly identical suggests that the model is generalizing well to unseen data.
   * This means the model is not just memorizing the training data but has captured the underlying patterns effectively.
2. **Model Performance**:
   * Depending on the domain and dataset, an RMSE of around 1.2251.2251.225 may or may not be considered good. You can compare this against domain-specific benchmarks or simpler baseline models to understand its adequacy.

 Significant **Spread**:

* A graph with blue dots and red line

  Description automatically generated
* There are many points far away from the diagonal line, indicating a large gap between actual and predicted values.
* This suggests the model is not fitting the data well and might be underfitting.

 Prediction **Trends**:

* The predictions seem to follow a general trend, but they fail to capture the variability in the data effectively.
* This could imply that the model is too simple (e.g., a linear model applied to non-linear data).

With training and testing accuracy both at **72%**, it indicates that the logistic regression model is generalizing well—there's no significant overfitting or underfitting based on these metrics alone. However, **72% accuracy** suggests there’s room for improvement. Here's a structured approach to investigate further and improv

**Observations:**

1. **Diagonal Values:**
   * The diagonal cells (top-left to bottom-right) represent the correctly classified instances for each class.
   * For example:
     + Class 1 has a large number of correctly classified instances (about 30,00030,00030,000).
     + Class 2 also has a significant count of correctly classified instances (about 45,24345,24345,243).
2. **Off-Diagonal Values:**
   * These represent misclassifications.
   * For instance:
     + A significant number of instances from Class 1 are being misclassified as Class 2 (about 11,83611,83611,836).
     + Similarly, many instances from Class 2 are being misclassified as Class 1 (about 10,00010,00010,000).
     + There’s a notable confusion between Classes 6 and 7.
3. **Rare Classes:**
   * Classes like 4 and 7 have smaller total counts, but they still have some correct classifications.
   * Misclassifications may disproportionately affect their performance metrics (e.g., precision and recall).

Logistic regression assumes linear relationships. If certain features are not contributing well to class separation, this could cause misclassification.

**Action:**

* Check feature importance using the logistic regression coefficients.

A screenshot of a graph

Description automatically generated

To ensure robust model evaluation during training, the dataset was split into a training set (80%) and a validation set (20%). The model's RMSE was tracked on both datasets across 3000 epochs. The inclusion of validation RMSE allowed us to monitor generalization performance and detect overfitting. Additionally, L2 regularization (α=0.001\alpha = 0.001α=0.001) was applied to prevent overfitting by penalizing large weights. The RMSE vs. Epochs plot indicates that the model stabilizes around 500–1500 epochs, suggesting an optimal number of training epochs

From the RMSE vs. Epochs plot with shuffled data, the validation RMSE stabilizes between 1500 and 2000 epochs, indicating model convergence. To strike a balance between convergence and computational efficiency, **1800 epochs** was chosen as the optimal training duration. This ensures the model is well-trained without overtraining or unnecessary computation

o ensure unbiased evaluation, the data was shuffled before splitting into training and validation sets. When the data was not shuffled, the validation RMSE exhibited significant fluctuations during early epochs, likely due to biased data splits where training and validation sets represented different regions of xxx. Shuffling resulted in smoother validation RMSE curves, closely aligning with training RMSE, and improved generalization. Therefore, shuffling was chosen for the final mode

**Choice of Output Activation Function**

In the final layer of the neural network, no activation function (i.e., a **linear transformation**) was applied. This choice was based on the nature of the problem:

1. **Continuous Target Values**:
   * The task is a regression problem where the target variable (yyy) is continuous and spans a wide range, from approximately −1000-1000−1000 to +1000+1000+1000.
   * A linear output ensures the model can predict any value within this range without constraints.
2. **Incompatibility of Sigmoid**:
   * Using a sigmoid activation in the output layer would restrict the model’s predictions to the range [0,1][0, 1][0,1]. While sigmoid is ideal for classification tasks (e.g., binary classification), it is not suitable for regression tasks where the output requires a much broader range.
3. **Avoiding Unnecessary Complexity**:
   * While it is theoretically possible to scale and normalize the yyy values to fit within a sigmoid’s range during training and reverse-scale them during inference, this would add complexity without meaningful benefit.
4. **Linear Output for Regression**:
   * A linear activation in the output layer is the standard practice for regression problems, as it allows the network to model the full range of the target variable natural

Given the small dataset size (100 examples), the chosen model architecture was designed to balance simplicity and performance. A single hidden layer with 64 neurons was selected to capture non-linear patterns while minimizing the risk of overfitting. Dropout and L2 regularization were employed to further control overfitting. Experiments with more complex architectures (e.g., additional layers or more neurons) did not yield significant improvements, confirming that the selected model was appropriate for the dataset size.

**What Happened Before**

1. **Original Features:**
   * Initially, your data had only one feature: xxx.
   * The neural network, while flexible, was tasked with learning a **non-linear relationship** directly from this single feature.
   * If the function f(x)f(x)f(x) generating yyy is inherently **non-linear** (e.g., f(x)=ax3+bx2+cx+df(x) = ax^3 + bx^2 + cx + df(x)=ax3+bx2+cx+d), the network must implicitly learn these higher-order relationships, which is more challenging with just xxx as input.
2. **Model Struggled with Complexity:**
   * Without polynomial features, the network had to discover these relationships purely through the weights and hidden layers.
   * This worked to some extent but likely required:
     + **More hidden layers or neurons** to approximate the higher-order terms.
     + **More data** to learn these patterns effectively.
   * This resulted in a model that performed poorly on unseen data (test data).

**Why Polynomial Features Helped Despite Using a Neural Network**

* Neural networks are theoretically capable of learning any continuous function (Universal Approximation Theorem), but:
  1. **They require sufficient data** to learn complex relationships. With only 100 training points, the network struggled.
  2. **Polynomial features act as a shortcut**, reducing the network's reliance on hidden layers to infer non-linear relationships.

By providing polynomial features, you effectively "preprocessed" the data to make it more aligned with the underlying f(x)f(x)f(x), improving the network's performance.

Using 5-fold cross-validation, polynomial degrees from 1 to 14 were evaluated by computing the average Root Mean Squared Error (RMSE) across validation folds. The optimal degree was selected based on the lowest average RMSE, ensuring a balance between underfitting and overfitting for better generalization.

Degree 3 was chosen as the optimal polynomial degree because the RMSE stabilized beyond this point, indicating minimal improvement in performance with higher degrees while avoiding overfitting  
Linear regression assumes that all features are on a similar scale. For example, if x values range from -10 to 10 and y values range from -1000 to 1000, the optimization might be less effective. Scaling the data can improve numerical stability.

**xample Outline for Linear Regression-Only Report**

1. **Introduction**:
   * Briefly describe linear regression and its assumptions.
   * Mention the synthetic nature of the dataset and the presence of noise.
2. **Methodology**:
   * Describe how the model was trained using ordinary least squares (OLS).
   * Explain any preprocessing steps, such as scaling or outlier removal.
3. **Results**:
   * Report training and test MSE.
   * Include figures:
     + Scatterplot with the regression line for training data.
     + Scatterplot with predictions for test data.
4. **Analysis**:
   * Discuss the limitations of the linear model given the non-linear pattern in the data.
   * Compare the original model to the scaled or filtered model (if applicable).
   * Provide confidence intervals for the slope and intercept.
5. **Conclusion**:
   * Summarize findings, emphasizing that linear regression provides a baseline but may not be the best fit for this dataset.

The distribution of training and validation data further highlights the model's need to handle **non-linear relationships effectively**, especially near the upper end of the range. Polynomial regression effectively improved the fit by mapping x to higher-degree features (x^2, x^3, etc.), suggesting that neural networks could benefit from similarly **enhanced input features**.  Based on these observations, I considered **explicitly mapping features** to higher degrees (e.g., x^2, x^3) before feeding them into the neural network.

 This allows the model to better approximate the underlying function without needing overly complex architectures.

 The decision is further supported by the fact that neural networks are capable of learning non-linear functions but may benefit from informative, preprocessed input data.

During training, it became evident that the neural network struggled to generalize well to the last 20% of the data range, where the validation points were concentrated. This prompted the realization that such performance issues would only worsen when dealing with even larger input ranges (e.g., x = 50, 60), as the network would struggle to extrapolate and approximate the non-linear patterns. To address this challenge, without overcomplicating the network architecture, we considered applying **feature transformations**.

By explicitly mapping features to higher degrees (e.g., x^2, x^3), the network can better approximate the underlying non-linear relationships. This preprocessing step simplifies the task for the model, allowing it to generalize effectively across the input range and reducing the risk of instability or overfitting. Additionally, this approach prepares the model for handling larger inputs outside the training range, minimizing performance degradation while maintaining a manageable architecture building on what I learnt with linear regression.

Initially, the neural network struggled to generalize, particularly for the last 20% of the data range, where non-linear patterns became more pronounced. To address this, we applied polynomial feature transformations (degree-3), allowing the model to better approximate complex relationships in the data. This preprocessing step improved validation performance, reducing RMSE and stabilizing convergence. The updated RMSE trends demonstrate that the transformed features enabled the network to achieve better generalization without increasing architectural complexity and needing less epochs  
  
Root Mean Squared Error (RMSE): 60.99973650892675

Mean Squared Error (MSE): 3720.9678541584913

Root Mean Squared Error (RMSE) - Test: 75.074214363339

Mean Squared Error (MSE) - Test: 5636.137662272576

In this experiment, both Adam and SGD optimizers were tested for minimizing RMSE on the given dataset. Adam showed faster and smoother convergence with a final RMSE of **42.5**, while SGD required more tuning and converged slower with a final RMSE of **45.4**. Based on these results, Adam is recommended for its reliability and minimal tuning requirements, though SGD remains a valid option for simpler tasks.