FYS-2021: Machine learning

Candidate 20
UiT The Arctic University of Norway
Troms, Norway

Contents

PROBLEM 1

This exam question focuses on the binary classification of molecules based on their lipophilicity, specifically determining whether a molecule prefers fat. The task involves classifying chemical data using various methods learned in the course. The dataset includes molecular descriptors that vary in complexity, providing insights into the chemical properties of the molecules.

Multiple classification techniques from scikit-learn will be applied to improve predictive accuracy through data preprocessing, hyperparameter tuning, and cross-validation. This systematic exploration aims to identify key features influencing lipophilicity while rigorously documenting the methods and results, highlighting the importance of validation and optimization in effectively classifying chemical data.

Data Analysis and Preprocessing

The dataset contains 3 360 training samples and 840 test samples, each described by 228 chemical descriptors. The target variable, lipophilicity, shows a significant class imbalance, as shown in Figure 6, with 71.88% of samples classified as non-lipophilic and 28.12% as lipophilic. This imbalance impacts model selection and training, affecting the choice of evaluation metrics and the use of class weights.

Feature analysis, illustrated in Figure 7, reveals that while some features, like MaxEStateIndex and MinEStateIndex, are tightly clustered near zero, others, such as MolWt and HeavyAtomMolWt, range up to 1600 and contain many outliers. These scale differences influenced our preprocessing strategy, particularly for algorithms sensitive to feature scales, like logistic regression.

The correlation matrix (Figure 5) shows the interactions between features, highlighting distinct clusters of correlated chemical descriptors. For instance, molecular weight-related features tend to be positively correlated, while certain functional group indicators show negative correlations. Understanding these relationships helped guide our feature selection and suggested that tree-based methods might be effective in handling these correlations. Overall, this enhanced explanation better connects the figures to their implications for our analysis and modeling decisions.

Classification Methods

The task involves binary classification of molecules based on their lipophilicity, determining whether a molecule "likes fat" (class 1) or not (class 0). Based on the dataset analysis, we face several challenges including class imbalance (71.88% vs 28.12%) and features with varying scales and correlations. To address these challenges and explore different modeling paradigms, we selected three distinct classification methods:

Logistic Regression

Logistic regression [15] is a linear classification method that predicts the probability of lipophilicity by combining features in a straightforward way. It is preferred for its ease of interpretation and effectiveness with large feature sets. Additionally, it provides insights into feature importance through its coefficients, which show how much each feature affects the prediction.

To implement this classifier, I used my own implementation from Mandatory Assignment 1 of this course. A few adaptations had to be made to better fit the dataset and the rest of the code necessary to solve this problem. The code for this classifier can be found in Appendix D.

Preprocessing

For the logistic regression classifier, we tested several scaling methods to address the varying feature ranges in our dataset, specifically StandardScaler, MinMaxScaler, and RobustScaler. The StandardScaler standardizes features by removing the mean and scaling to unit variance, resulting in a mean μ of 0 and a standard deviation δ of 1, using the formula

$$z = \frac{(x - \mu)}{\delta}$$

This method is effective when features have different scales and approximately normal distributions. The MinMaxScaler scales features to a fixed range of [0,1] with the formula

$$z = \frac{(x - \min(x))}{(\max(x) - \min(x))}$$

While it preserves zero values and works well with non-normally distributed data, it struggles with outliers since it relies on the minimum and maximum values. The RobustScaler addresses this issue by using the median and interquartile range (IQR) to scale features, following the formula

$$z = \frac{x - \text{median}(x)}{IQR(x)}$$

This makes it particularly suitable for our dataset with significant outliers.

Comparing these scaling methods was essential due to the varying ranges of our molecular descriptors and the presence of outliers in features like molecular weight. The choice of scaling method can greatly influence logistic regression performance, as the algorithm is sensitive to the scale of input features.

The tuning process revealed that the $StandardScaler$ achieved a Kaggle score of 0.545, while $MinMaxScaler$ scored 0.538, indicating strong performance. In contrast, $RobustScaler$ scored only 0.306. The success of $StandardScaler$ and $MinMaxScaler$ can be attributed to their ability to handle the continuous nature of our molecular descriptor data effectively. $StandardScaler$ is particularly effective because it standardizes features by centering them around zero and scaling based on standard deviation, making it ideal for datasets with normally distributed features. $MinMaxScaler$, on the other hand, scales all features to a fixed range of [0, 1], which preserves the relationships between features and is beneficial when dealing with features that are not normally distributed.

In contrast, $RobustScaler$'s poor performance can be explained by its reliance on the median and interquartile range, which makes it less sensitive to the distribution of the data. While this method is beneficial for datasets with significant outliers, the relatively clean nature of our dataset meant that the advantages of $RobustScaler$ did not materialize, resulting in a much lower score. This demonstrates that the choice of scaling method can greatly influence model performance, particularly in relation to the specific characteristics of the dataset.

Hyperparameter Selection

The hyperparameters being tuned through grid search are the learning rate and the number of epochs. The learning rate, which determines the step size for gradient descent, is evaluated using values of 0.001, 0.01, and 0.1. For the number of epochs, values of 10,000, 50,000, and 100,000 are tested. Additional values like a learning rate of 0.0001 and epochs up to 500,000 were considered but did not improve results, so they were excluded.

Tuning the number of epochs was particularly challenging. Initially, the values tested were 50, 100, and 200, but the best performance came from using between 50,000 and 100,000 epochs. This highlights the need for careful parameter selection to optimize model performance.

The key improvement from tuning was finding the right number of epochs, which stabilized the accuracy score at around 80%. However, this figure should be viewed with caution. First, the 80% accuracy is based on a part of the training data with known true lipophilicity values, aiding in model optimization but not reflecting final performance. Second, we cannot assess accuracy for the test set predictions since we don't know the true labels until evaluated by Kaggle. Thus, while local accuracy metrics are helpful for tuning, they do not represent true performance on the test set, highlighting the difference between training optimization and actual testing results.

Decision Trees

A decision tree [7] is a non-linear method that uses a flowchart-like structure to make decisions. It is chosen for its capability to handle both numerical and categorical features without the need for scaling. This method can capture complex interactions between features and provides interpretable rules, making it easy to understand how decisions are made.

The implementation utilized Scikit-learn's functions for Decision Trees [3]. The classifier was evaluated both with and without Standard Scaling, but the results showed no significant differences in accuracy or Kaggle scores. Consequently, testing continued without applying scaling.

Hyperparameter Selection

The decision tree classifier's performance was improved by tuning key hyperparameters. Max_depth was tested at [3, 5, 7, 11, 13] to balance complexity and overfitting; lower values reduce overfitting, while higher values capture more details. $Min\_samples\_split$ was set to [2, 5, 10, 15] to control how many samples are needed to split a node, affecting rule specificity. N_folds was fixed at 5 for reliable cross-validation. Lastly, $max\_features$ was set to 'sqrt', allowing about 13 of the 187 features at each split, which helps prevent overfitting. These parameters work together to optimize the model's learning from the data.

These hyperparameters were tuned using cross-validation. Cross-validation is essential for decision trees because they are prone to overfitting. We used 5-fold cross-validation, dividing our 3,360 training samples into five parts. In each round, four parts are used for training and one for validation, ensuring each sample is validated once. This approach provides a reliable performance estimate and helps select effective hyperparameters by testing across different data subsets. Its particularly important for our imbalanced dataset, where only 28.12% of the molecules are lipophilic. Overall, 5-fold cross-validation allows us to make better use of our data than a simple train-test split.

Results

The impact of hyperparameter tuning on the decision tree's performance was significant. We started with a relatively narrow range of parameters:

```
hyperparameters = {
    'max_depth': [3, 5, 7],              # Limited depth options
    'min_samples_split':[5, 10],         # Conservative splitting criteria
    'n_folds': 3,                        # Basic cross-validation
    'max_features': 'sqrt'               # Standard feature selection
}
```

This initial configuration yielded a modest Kaggle score of 0.33. However, by expanding the hyperparameter search space to allow for more complex trees and finer-tuned splitting criteria:

```
hyperparameters = {
    'max_depth': [3, 5, 7, 11, 13],          # Broader range of tree depths
    'min_samples_split':[2, 5, 10, 15],      # More flexible splitting options
    'n_folds': 5,                            # More robust cross-validation
    'max_features': 'sqrt'                   # Maintained feature selection
}
```

The model's performance improved substantially, achieving a Kaggle score of 0.43. This 30% improvement in F1 score demonstrates how critical comprehensive hyperparameter tuning is for decision trees. By allowing the model to explore a wider range of tree structures while maintaining safeguards against overfitting, we found a better balance between model complexity and generalization ability.

Random Forest

The implementation utilized Scikit-learn's functions for Random Forests [2].

Random forest [11] is an ensemble method that combines multiple decision trees to enhance robustness. It is selected to take advantage of the strengths of decision trees while minimizing the risk of overfitting. This approach is particularly effective for managing high-dimensional feature spaces and capturing potential interactions between features.

Random forests work well with our molecular dataset due to their scale invariance, handling a mix of features naturallyranging from large values like molecular weight to small integers and binary indicators. Since tree-based models dont require scaled features, we can directly use our raw data, which maintains the interpretability and relationships among chemical descriptors. This compatibility makes random forests particularly effective for our data without extra preprocessing, unlike other models that need feature scaling.

Hyperparameter Selection

To handle hyperparameter tuning in the Random Forest implementation, the code allows for two approaches: using cross-validation ($main\_rf\_cross$) and without cross-validation ($main\_rf\_no\_cross$). When not using cross-validation, the model directly applies fixed hyperparameter values:

```
rf_model = {
      n_trees = 100,                # Number of trees in the forest
      max_depth=5,                  # Maximum tree depth
      min_samples_split=5,          # Minimum samples for a split
      max_feature='sqrt',           # Features per split
      class_weights='balanced'      # Adjust for class imbalance
    }
```

In this simpler approach, the model is initially trained on a validation split and then retrained on the entire dataset with the same fixed parameters. This method is efficient but may miss the optimal parameter set.

In the cross-validation approach ($main\_rf\_cross$), key parameters n_trees, max_depth, $min\_samples\_split$, and $max\_features$ are selected through systematic tuning:

```
rf_model = {
    n_trees=best_params['n_trees'],
    max_depth=best_params['depth'],
    min_samples_split=best_params['min_samples'],
    max_features=best_params['max_features'],
    class_weights='balanced'
}
```

The $perform\_rf\_cross\_validation$ function uses grid search with 5-fold cross-validation to evaluate combinations and select parameters that maximize performance metrics, addressing class imbalance by maintaining balanced class weights. Parameters were iteratively tuned through values in this hyperparameter grid:

```
hyperparameters = {
    'n_trees': [100, 200, 300],
    'max_depth': [7, 10, 15],
    'min_sample_split': [2, 5, 10],
    'n_folds': 5
}
```

This cross-validated approach systematically tests parameter combinations, providing a robust selection method compared to using fixed values in $main\_rf\_no\_cross()$.

Results

The Random Forest implementation highlights the importance of cross-validation and hyperparameter tuning. Initially, with fixed parameters and no cross-validation, the model scored only 0.035 on Kaggle. Adding k-fold cross-validation improved this to 0.31, showing the value of proper validation. Finally, tuning key hyperparameterslike the number of trees, max depth, and minimum samples per splitincreased the score to 0.58, demonstrating that even robust models benefit greatly from optimized parameters.

Performance Comparison

Random Forest achieved the best performance with a Kaggle score of 0.58 after tuning. Its initial score was very low at 0.035 without cross-validation, but it improved to 0.31 with k-fold cross-validation. Further hyperparameter optimization raised the score to 0.58.

Logistic Regression performed well with scores of 0.545 using StandardScaler, 0.538 with MinMaxScaler, and 0.306 with RobustScaler. It reached around 80% accuracy on training data, indicating strong predictive capability, especially when scaled correctly.

Decision Trees started with a Kaggle score of 0.33 using basic hyperparameters, which improved to 0.43 after extensive tuning, showing a 30% increase in F1 score. However, this was still the lowest performance among the three methods.

Best Performing Method

Random Forest stood out as the best method with a score of 0.58 due to several factors. Its ensemble nature helps reduce overfitting while capturing complex relationships between features. It handles the diverse features in the molecular dataset without needing scaling, which maintains the natural relationships between descriptors. The improvement from 0.035 to 0.58 shows its effectiveness with proper tuning.

Logistic Regression also performed well with a score of 0.545, suggesting that there are strong linear relationships in the data, although its performance depended heavily on the scaling method used.

In contrast, Decision Trees had the lowest score at 0.43, likely due to a higher risk of overfitting compared to the Random Forest and the regularization in Logistic Regression.

Overall, these results highlight that while model choice matters, proper implementationlike cross-validation and hyperparameter tuningis key to achieving the best performance.

Conclusion

This assignment provided valuable insights into binary classification for determining molecular lipophilicity. By employing Logistic Regression, Decision Trees, and Random Forest classifiers, the importance of model selection, preprocessing, and hyperparameter tuning was clearly demonstrated.

The Random Forest emerged as the most effective model, achieving a Kaggle score of 0.58 due to its ability to handle complex data without requiring feature scaling. Logistic Regression followed with a score of 0.545, benefiting from appropriate scaling techniques, while Decision Trees scored 0.43, facing challenges related to overfitting.

This experience underscored the significance of preprocessing methods, particularly for imbalanced datasets, and highlighted the critical role of cross-validation in ensuring robust model validation. Overall, the analysis emphasized the complexities of chemical data classification and the necessity of meticulous model implementation for optimal performance.

PROBLEM 2

Problem 2 explores unsupervised learning techniques, specifically k-means clustering, applied to a dataset of gray scale face images.

A: Clustering and Classification

Clustering is a form of unsupervised learning, which is a way to gain insights from unlabelled data points. Clustering is a way of grouping together data points based on how similar they are with each other [6]. Classification, on the other hand, is a form of supervised learning, where there is an input variable x and an output variable y. The variables gets mapped together using a mapping function, where the goal is for the mapping function to predict the outputs of new inputs [9].

The main difference between classification and clustering is the presence of labels. In classification, each data point is associated with a predefined label, whereas in clustering, the data points have no assigned labels. As a result, classification algorithms require both a training dataset to learn from and a testing dataset to evaluate the model's performance. In contrast, clustering algorithms do not rely on labeled data and therefore do not need separate training and testing sets for model verification [10].

Clustering is a powerful tool for data analysis across various fields [4]. For instance, businesses use clustering to segment their customers and create targeted marketing strategies. In healthcare, doctors apply clustering techniques to identify disease-affected areas in diagnostic images such as X-rays and CT scans. A notable example in astronomy is the Hertzsprung-Russell diagram (Figure 1), which clusters stars based on their temperature and luminosity, highlighting relationships between different types of stars.

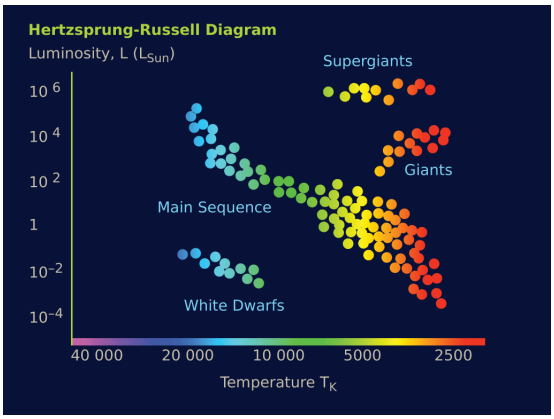

Fig. 1: Hertzsprung-Russel Diagram [13] is a result of clustering

B: K-means Information

The K-means algorithm is a widely used unsupervised learning technique for clustering data points [14]. It works by iteratively dividing the data into K clusters, where K is the predefined number of clusters. The algorithm minimizes the variance within each cluster, grouping similar data points together to optimize the overall clustering structure.

The centroid represents the "average" position of all data points in a given cluster and serves as the center of that cluster. In the K-means algorithm, centroids are initialized by randomly selecting K data points, where K is the desired number of clusters [5]. Once the centroids are selected, each data point is assigned to the cluster with the nearest centroid, typically determined using Euclidean distance. After this assignment, the centroids are updated to reflect the new average position of the data points of the data points within each cluster.

This process of reassigning data points and updating the centroids is repeated iteratively until the data points remain stable, meaning they no longer change clusters after the centroids are recalculated. This convergence marks the stopping criterion for the algorithm.

To apply the K-means algorithm to 2D image data, the data must be reshaped into a format that K-means can process, typically a 1D vector for each image [12]. For gray-scale images, each pixel is treated as a single feature, resulting in a 1D feature vector with on vector per image. K-means is then executed on these feature vectors, where each cluster represents a group of similar pixel intensities. After clustering, the images can be reconstructed into their original 2D format, with each pixel assigned the value of its corresponding cluster centroid. Figure 2 illustrates the before and after effects of applying K-means to an image.

In the case of colored images, which contain multiple color channels, the feature vectors are organized in 3D. Each pixel is represented by its values across the different color channels, allowing K-means to cluster based on color information.

Fig. 2: Original image and the image after K-means with K = 3 [12]

C: Dataset Familiarization

Figure 3 shows nine randomly selected samples from the dataset, all of which are grayscale images depicting the same person with various facial expressions. The images are well-aligned, with the faces centered and consistently scaled across all samples, making them uniform in appearance.

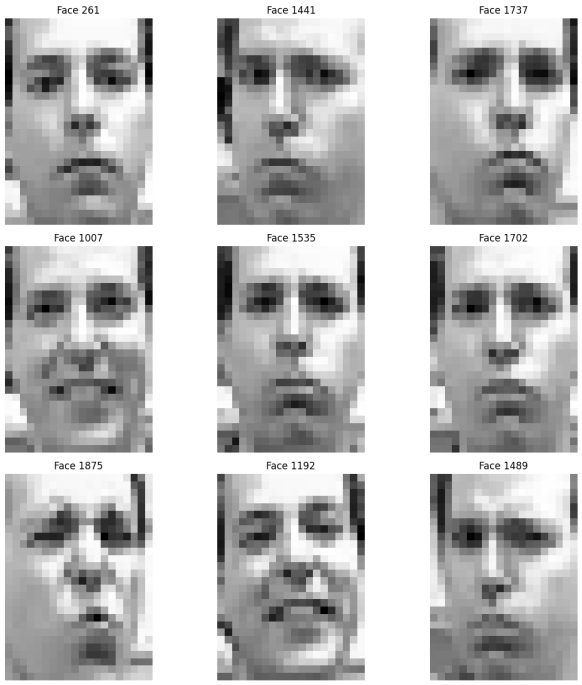

Fig. 3: A display of various samples from the dataset

Figure 4 shows a histogram that illustrates the distribution of pixel intensities within the dataset, with grayscale values ranging from 0 (black) to 255 (white), which is typical for 8-bit images. The x-axis represents pixel intensities, while the y-axis shows the frequency of pixels at each intensity level. The distribution is bimodal, with the first peak around intensity 150-160 and the second peak around 220. This indicates two dominant brightness levels that may correspond to different regions in the images, in which the first peak could represent the facial features, while the second peak likely represents the lighter background or skin tones. The lack of very dark pixels suggests well-lit images without deep shadows, while the lack of white pixels suggest no overexposure.

Based on these images, I believe these clusters will either form on the color information, like Figure 2, or the different facial expressions presented by Figure 3. If the clusters cluster on color information, I imagine about three clusters will be enough to get a fairly precise accuracy, and if the algorithm creates clusters based on facial expressions, I imagine more clusters is better than few clusters. This is because the span of grayscale colors presented in Figure 3 hints at about three clusters, black/dark, gray/middle, and white/light. However, if we look at the different facial expression presented by Figure 3, we see that they are smiling, frowning, showing discomfort and being neutral. Since nine images present at least four distinct emotions, it is likely more among the 1965 images, and more clusters will be needed to classify the images as best as possible.

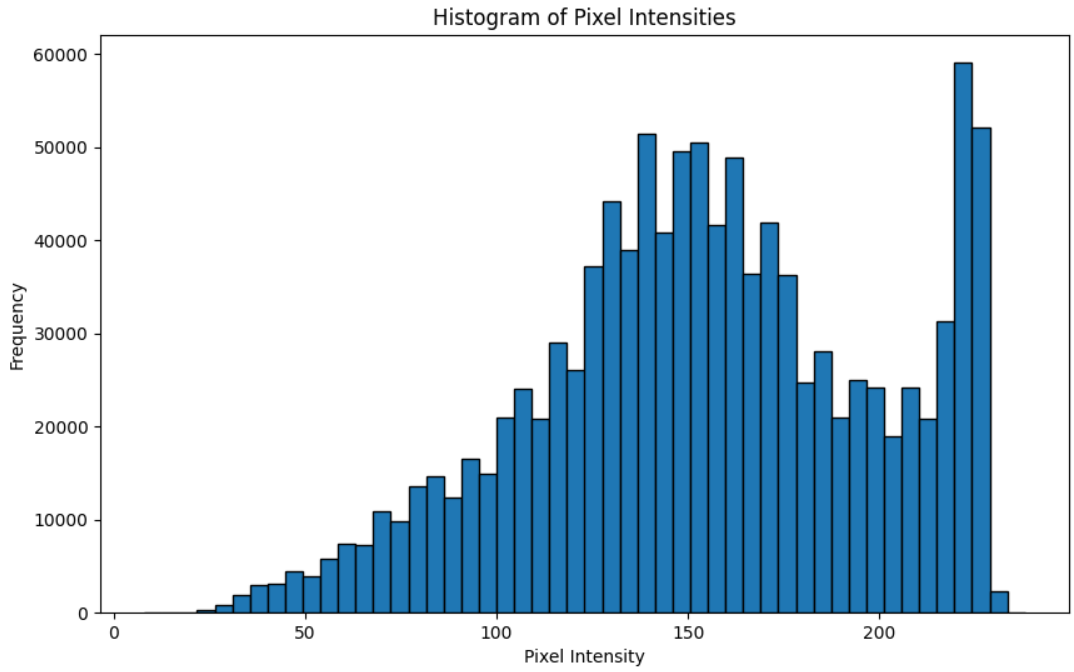

Fig. 4: Histogram showing the Intensities of the pixels

D: Implementing K-Means

The complete code can be found in Appendix E and is implemented using only NumPy, without any additional libraries. Below, the code will be divided into sections and explained in detail.

The Helper functions

To implement the K-means algorithm, several helper functions were created to improve clarity. These functions were separated from the main function to make the code easier to read and understand.

Firstly, a function was created to initialize a centroid.

```
def centroid_init(data, K):
    random_image = np.random.randint(0, data.shape[0], size=K)
    centroids = data[random_image]

    return np.array(centroids)
```

The `centroid_init` function initializes K centroids for the K-means algorithm by randomly selecting K images from the dataset. It takes two parameters: `data`, representing the dataset where each row is an image, and `K`, the number of clusters. The function generates K random indices and uses them to pick images from the dataset, returning these images as centroids. This random initialization is essential as it provides starting points for the K-means clustering process.

Euclidean distance [8] between two points in an n-dimensional space is given by

$$d = \sqrt{\sum_{i=1}^{n} (x_{2i} - x_{1i})^2}$$

where d is the distance between points $(x_{11}, x_{12}, \ldots, x_{1n})$ and $(x_{21}, x_{22}, \ldots, x_{2n})$. This formula generalizes the familiar two-dimensional distance calculation to n-dimensional space, which is particularly relevant in our case since each image in the dataset is represented as a high-dimensional vector. By explicitly defining the summation range from $i = 1$ to n, using precise subscript formatting, and defining each of the points in this space, this formula directly addresses the complexity of comparing images represented as vectors.

The code for calculating Euclidean Distance is provided below

```
def euclidian_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2)**2))
```

The $euclidian\_distance$ function computes the Euclidean distance between two points represented as vectors. It takes two inputs, "point1" and "point2", subtracts them element-wise, squares the differences, sums these squared values, and takes the square root of the result. This efficient implementation using numpy handles vectors of any length, making it well-suited for our dataset of high-dimensional face images, where each image is represented as a 560-dimensional vector (2028 pixels).

Another function was created to update the centroid positions:

```
    def update_centroids(data, clusters, K):
        new_centroids = np.zeros((K, data.shape[1]))

        for i in range(K):
            data_points_in_cluster = data[clusters == 1]
            new_centroids[i] = np.mean(data_points_in_cluster)

        return new_centroids
```

The $update\_centroids$ function is an important part of the K-means algorithm that recalculates each centroids position based on current cluster assignments. It initializes a new array for centroids and calculates the mean position for the data points in each cluster. The function takes three parameters: data, which contains the dataset; clusters, an array with cluster assignments; and K, the number of clusters. For each cluster, it identifies the assigned data points, computes their average position, and updates the centroid to this new mean. In our face dataset, each new centroid represents the average face of the images in its cluster, helping the algorithm refine cluster assignments until it converges.

The K-Means Algorithm

The $k\_means\_algorithm$ function manages the entire K-means clustering process, starting with its initialization phase.

```
def k_means_algorithm(data, K, max_iters=100, threshold=0.001):
    old_centroids = centroid_init(data, K)
    converged = False
    iters = 0
    visual_centroids = np.zeros_like(old_centroids)
```

The $k\_means\_algorithm$ function manages the entire K-means clustering process, starting with its initialization phase. It takes four parameters: data, the dataset of face images represented as vectors; K, the number of clusters; max_iters, which sets the maximum iterations (default is 100); and threshold, the convergence threshold (default is 0.001). During initialization, the function randomly selects K images as initial centroids using the $centroid\_init$ method, establishes variables to track convergence, and creates an array for visual representations of centroids. This setup is crucial as it prepares the algorithm for the main iterative phase, where cluster assignments are refined until either convergence is achieved or the maximum iteration limit is reached.

Moving on to the main loop of the K-means clustering algorithm, which will continue until the algorithm reaches convergence.

```
    while not converged and iters < iters_max:
        old_clusters = assign_data_to_cluster(data, old_centroids)
        new_centroids = update_centroids(data, old_clusters, K)
```

The K-means algorithm's core is an iterative loop that continues until the centroids stabilize or the maximum number of iterations is reached. Each iteration consists of two main steps: the Assignment Step and the Update Step. In the Assignment Step, each data point is assigned to the nearest centroid based on Euclidean distance, forming clusters. In the Update Step, the algorithm calculates the average position of points in each cluster and moves the centroid to this new mean. This process of assigning points and updating centroids allows K-means to improve groupings over time. For our face image dataset, it means iteratively grouping similar faces and refining the "average face" for each group until the clusters stabilize.

The next section creates a visual representation of the centroids (average faces) for each cluster by iterating through each cluster with a loop.

```
        for i in range(K):
            cluster_images = data[old_clusters == i]
            visual_centroids[i] = np.mean(cluster_images, axis=0)
```

For each cluster, it selects all images assigned to that cluster using a boolean mask, which filters the data to include only those images. It then calculates the average of these images, resulting in a single "average face" for each cluster, which is stored as the visual centroid. In our face dataset, each visual centroid represents a typical face in that cluster, helping to visualize the features and expressions characteristic of each group. Unlike the mathematical centroids used in calculations, these visual centroids are designed to be viewed as images.

The next step in the algorithm is to calculate the average movement of the centroids between iterations, and to check if the algorithm has reached convergence.

```
        distance = np.mean([euclidian_distance(old_centroids[i], new_centroids[i] for i in range(K)]
                                                                                    )
        if distance < threshold:
            converged = True
```

By looping through each centroid, the distance between its old and new positions is computed using the $euclidian\_distance$ function. It generates a list of distances for all centroids and then uses $np.mean$ to compute the average of these distances, resulting in a single value that represents the average centroid movement. This measurement is crucial as it indicates how much the centroids are still shifting, helping to determine if the algorithm has converged. If the average distance falls below a specified threshold, we consider the algorithm to have converged, suggesting that the clusters are stabilizing. At that point the algorithm will stop, and exit the while loop. If the distance is above the threshold, the loop will continue by updating the centroid position for the next iteration, as well as increasing the iteration counter.

```
        old_centroids = new_centroids.copy()
        iters += 1

    return new_centroids, old_clusters, visual_centroids
```

At last, the return statement from the K-means algorithm outputs three key components: $new\_centroids$, $old\_clusters$, and $visual\_centroids$. $New\_centroids$ are the final mathematical centers of each cluster used for calculations, while $old\_clusters$ indicate the final assignments of each image to its respective cluster. $Visual\_centroids$ provide average faces for each cluster, helping to visualize typical features. Together, these elements allow us to understand the clustering results, identify image groupings, and interpret the data effectively.

Results

Inertia [1] is a key metric in evaluating K-means clustering. It measures the compactness of clusters by calculating the sum of squared distances from each data point to its centroid; lower inertia indicates more compact clusters, but increasing clusters (K) will naturally reduce inertia, potentially leading to overfitting. The find_best_clustering function minimizes the randomness in initialization by running K-means multiple times and selecting the clustering with the lowest inertia, calculated by calculate_inertia.

For visualizing the clustering results, a $k \times 6$ grid layout was used where each row represents a cluster. The first column shows the centroid (average face) for each cluster, with the next five columns displaying the five closest faces to the centroid, selected based on Euclidean distance. This visualization was created for $k = 3$, $k = 6$, and $k = 10$, as shown in Figures 8, 9, and 10, respectively. This layout effectively highlights the shared features within each cluster. Due to visibility of the images, they are all placed at the bottom of this report.

E: Cluster Representation

The six clusters (Figure 9) represent distinct combinations of facial features and expression intensities, primarily based on variations in mouth position (neutral, slight frown, pronounced frown), eyebrow configuration, and the overall intensity of expression. While a three-cluster (Figure 8) solution was too broadmerging 1,261 diverse imagesand ten clusters (Figure 10) over-segmented the data by creating artificial distinctions, six clusters provided the best balance, capturing natural groupings without unnecessary segmentation.

This outcome aligns partially with my initial intuition about grouping by expression; however, I hadnt anticipated that certain facial features, like mouth position and eyebrows, would be weighted so heavily in the clustering process. Overall, the six-cluster solution effectively represents meaningful patterns in facial expressions.

F: Slightly Shifted Image Centers

Since k-means clusters based on overall patterns of facial expressions, slight shifts in face position likely wont impact the results, as long as the entire face remains within the frame. This is because the relationships between facial featuressuch as the positions of eyes, mouth, and eyebrows relative to each otherare preserved, allowing the algorithm to still identify similar expressions. However, if clustering were based solely on exact pixel positions, even slight shifts would lead to different pixel values at each position, potentially causing similar expressions to end up in separate clusters.

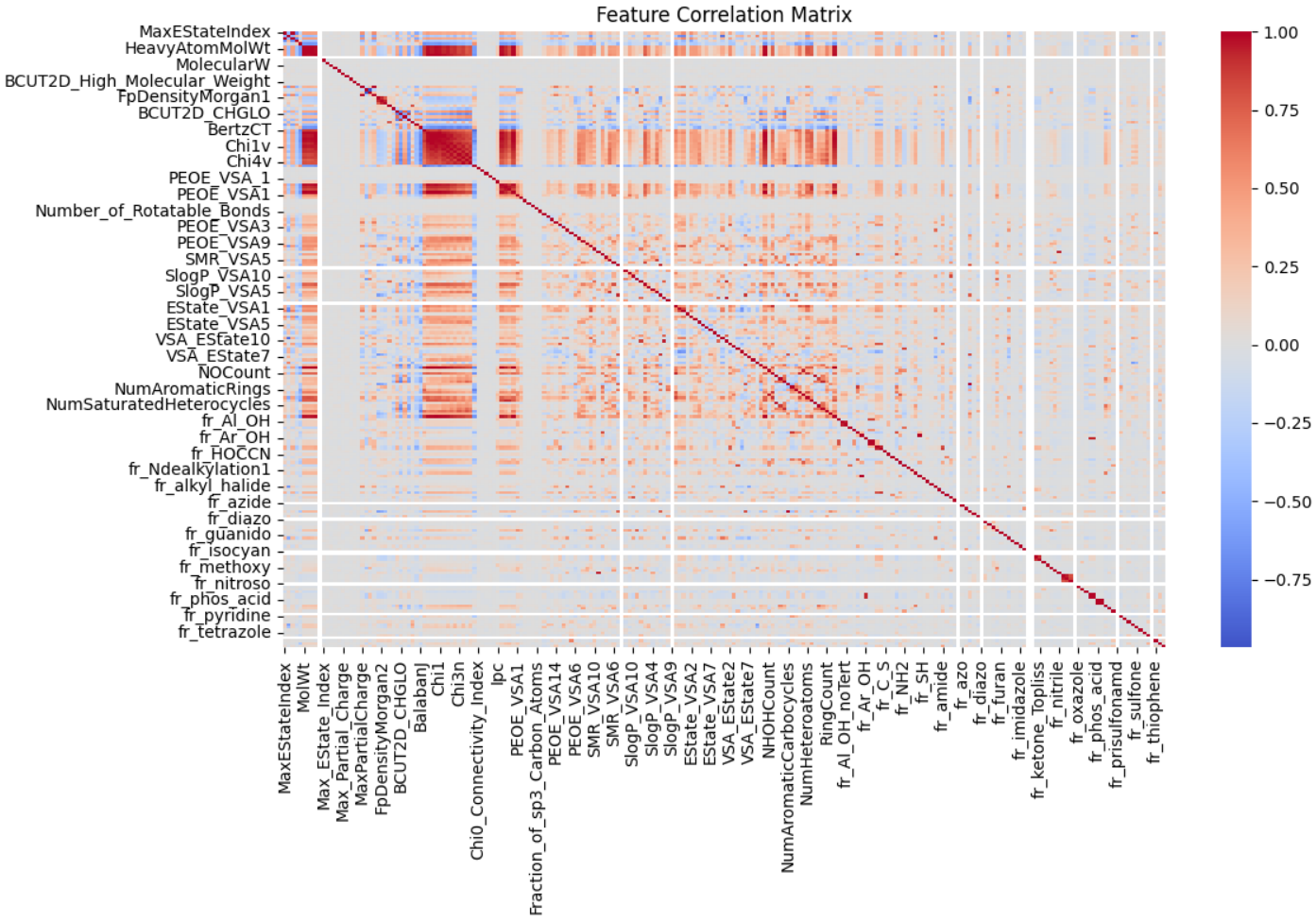

Fig. 5: Pairwise correlation matrix between molecular descriptors highlighting clusters of related chemical properties.

APPENDIX

A. Logistic Regression Implementation

```
import numpy as np
from sklearn.model_selection import train_test_split

from preprocessing import load_and_preprocess_data
from utils import create_submission


class LogisticClassifier ():
    def __init__(self, lr=0.01, epochs=100, bias=None):
        self.lr = lr                    # learning rate
        self.epochs = epochs            # epochs = 100
        self.weights = None
        self.bias = None

    def sigmoid(self, lin_reg):
        lin_reg = np.clip(lin_reg, -500, 500)       # Clipping the values below and over 500, to
        z = 1/(1+np.exp(-lin_reg))                  # Calculating the sigmoid function
        return z

    def loss_func(self, pred_y, true_y):
        epsilon = 1e-15
```

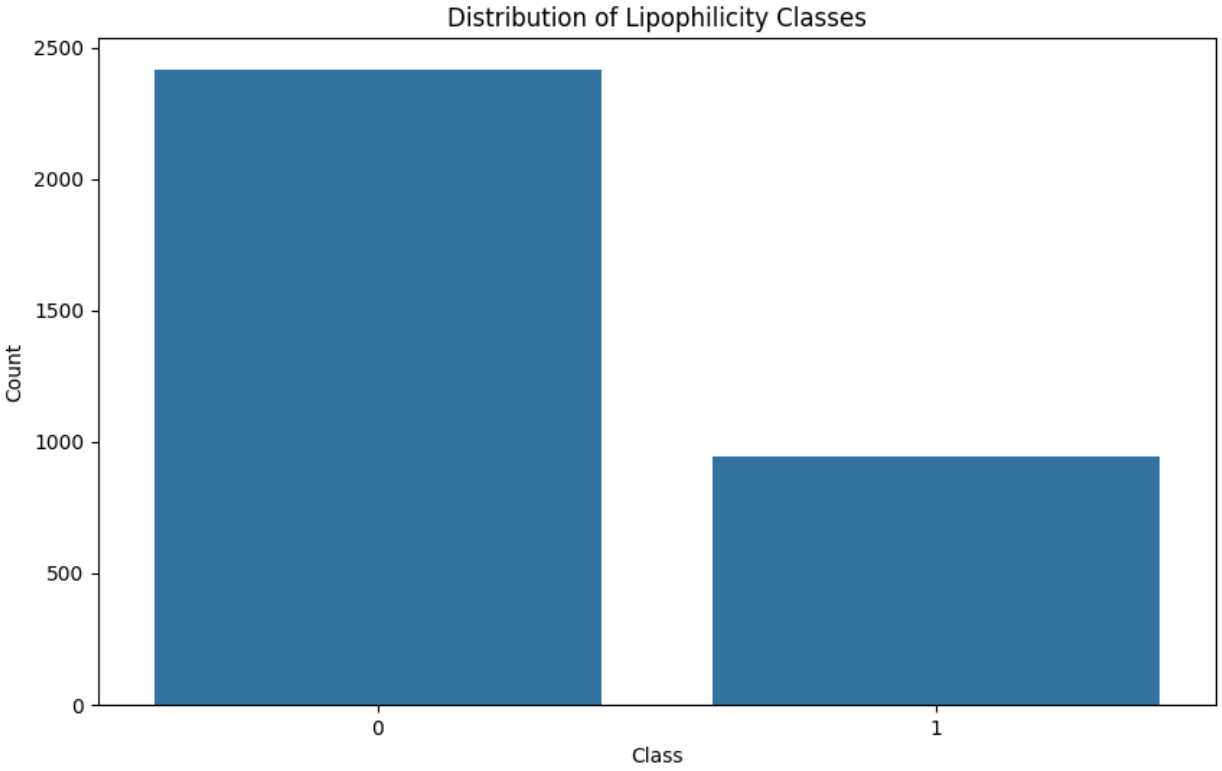

Fig. 6: Class distribution showing significant imbalance with 71.88% non-lipophilic (class 0) versus 28.12% lipophilic (class 1) samples.

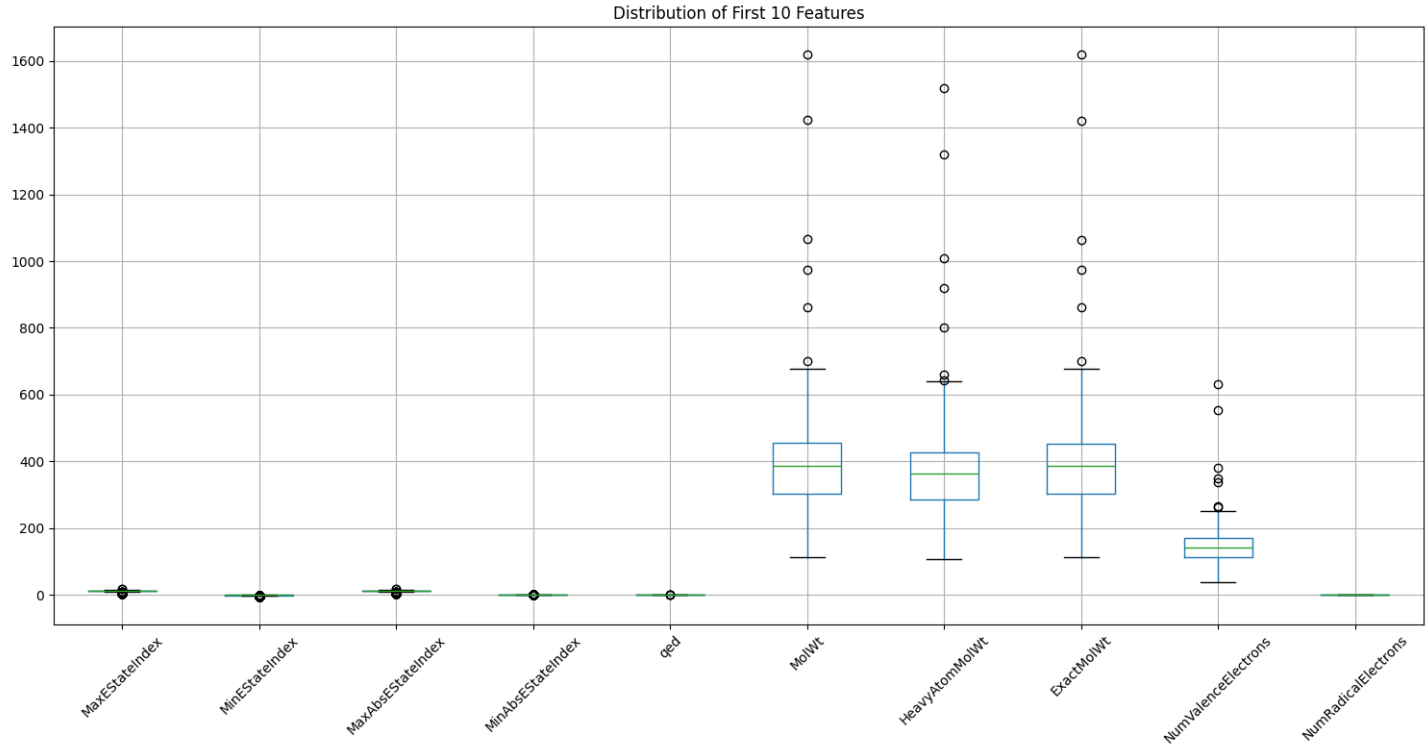

Fig. 7: Distribution of the first 10 molecular descriptors showing significant scale differences between features.

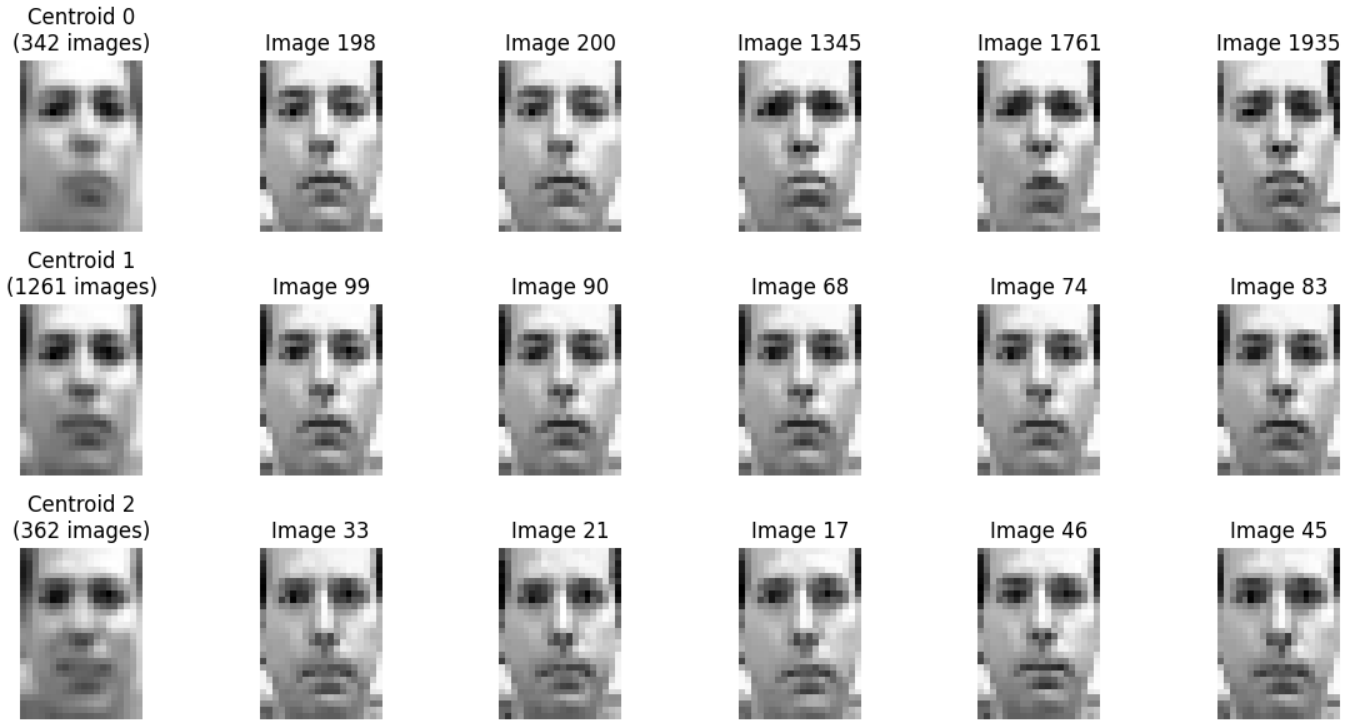

Fig. 8: Results with 3 clusters

```
    pred_y = np.clip(pred_y, epsilon, 1 - epsilon)
    L = true_y*np.log(pred_y) + (1-true_y)*np.log(1-pred_y)
    return L.mean()

def SGD(self, X_train, y_train):
    training_errors = []
    self.bias = 0                                   # Initializing the bias as 0
    samples, features = X_train.shape               # Initializing the number of samples and featu
    self.weights = np.zeros(features)               # Initializing the weights as 0 (as many as t

    for epoch in range(self.epochs):
        lin_pred = np.dot(X_train, self.weights) + self.bias        # Calculating the predict
        predictions = self.sigmoid(lin_pred)                        # Calculating the actual

        # Calculating the gradient
        dw = (1/samples) * np.dot(X_train.T, (predictions - y_train))
        db = (1/samples) * np.sum(predictions-y_train)

        # Calculating gradient, and updating the weights
        self.weights = self.weights - self.lr *dw
        self.bias = self.bias - self.lr*db

        loss = self.loss_func(predictions, lin_pred)
        training_errors.append(np.sum(loss) / samples)

# Predicting the values of y_train based on the training of X_train
def predict(self, X):
    linear_pred = np.dot(X, self.weights) + self.bias
    y_pred = self.sigmoid(linear_pred)

    class_pred = (y_pred >= 0.5).astype(int)
    return class_pred
```

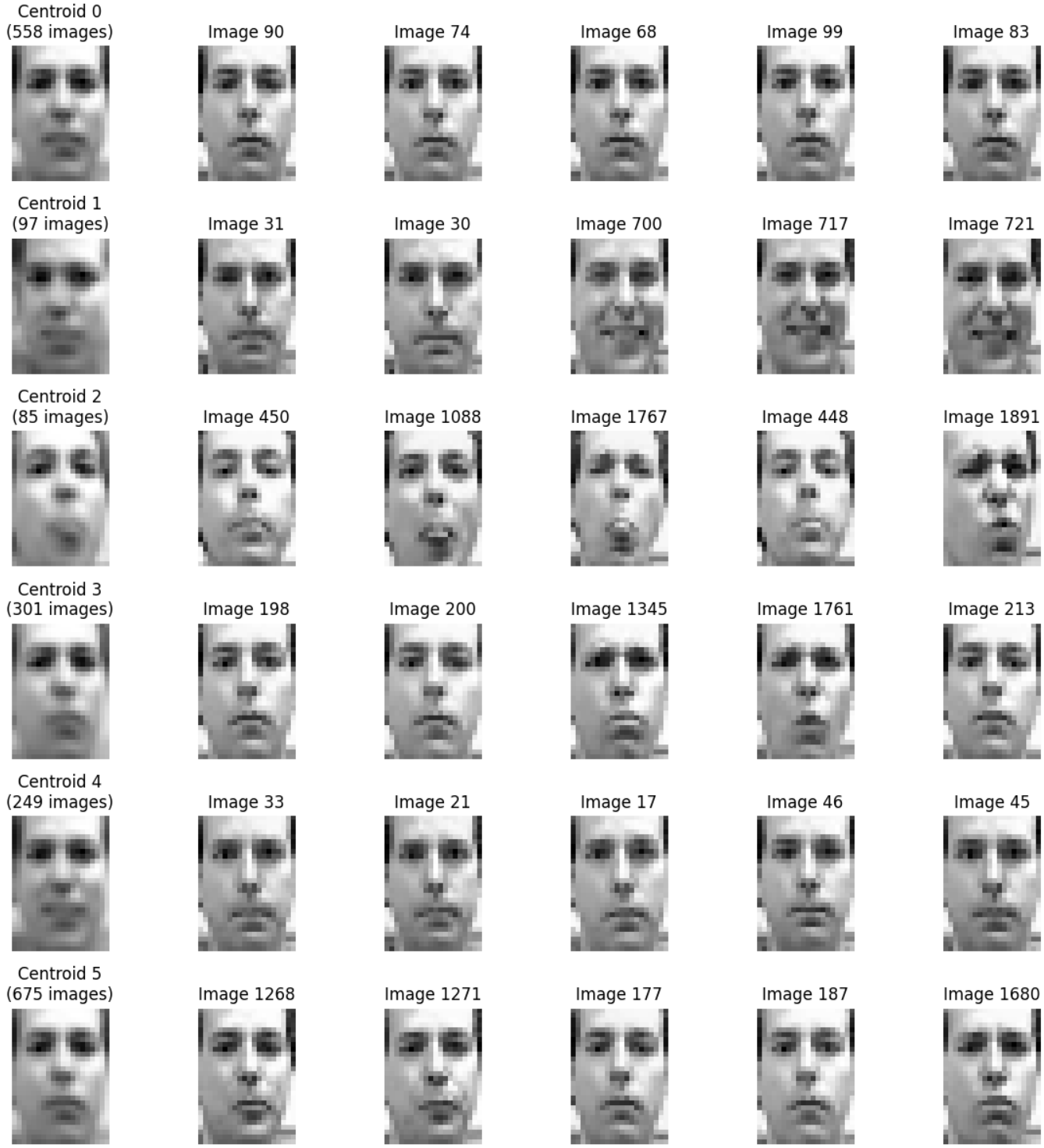

Fig. 9: Results with 6 clusters

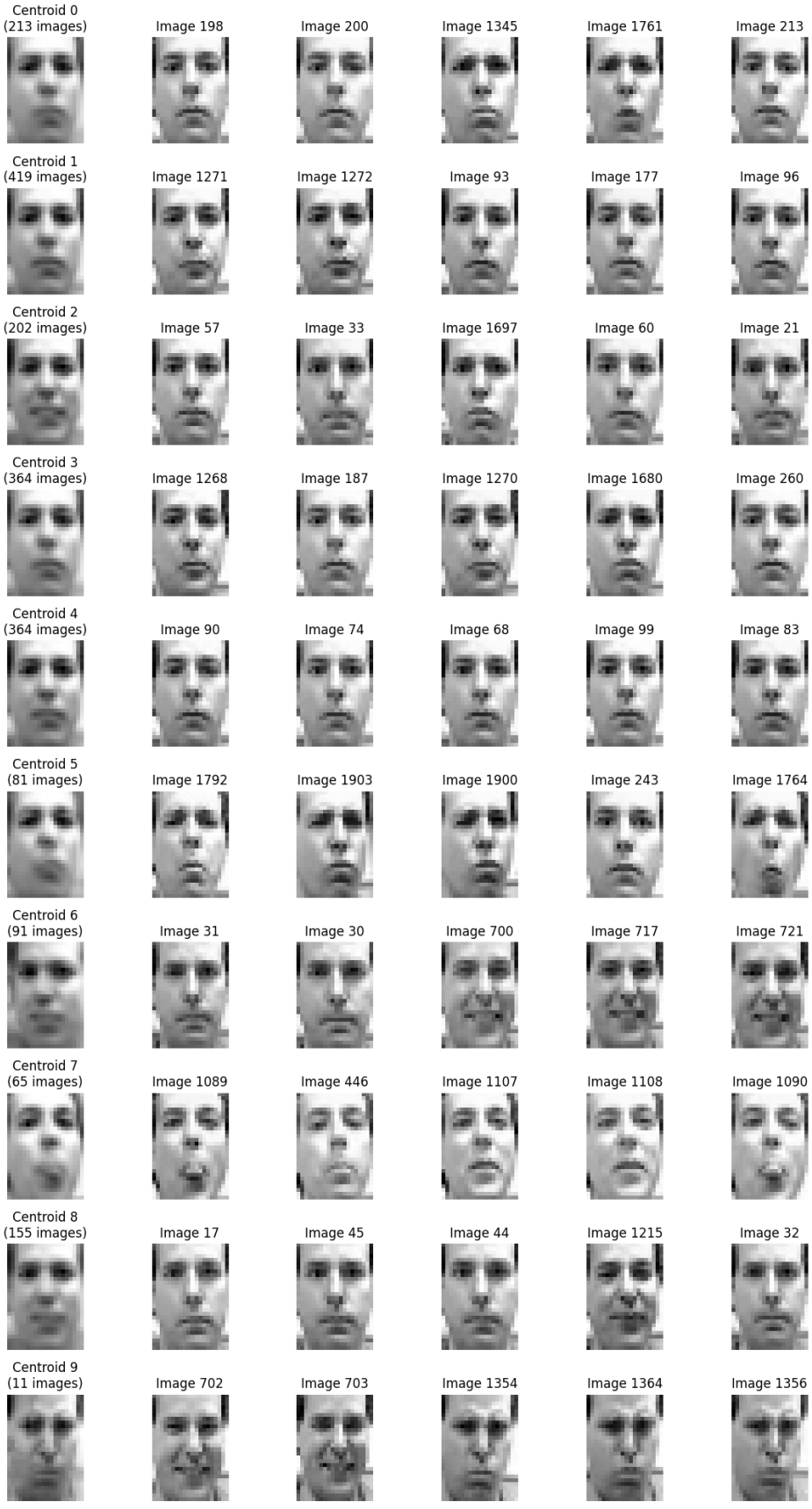

Fig. 10: Results with 10 clusters

```
def hyperparameter_tuning(X, y):
    print("Tuning the hyperparameters of Logistic Regression ...")

    learning_rates = [0.001, 0.01, 0.1]
    epochs = [10000, 50000, 100000]

    best_params = {
        'lr': None,
        'epochs': None,
        'score': -float('inf')
    }

    X_train, X_test, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

    for lr in learning_rates:
        for epoch in epochs:
            print(f"Testing lr={lr}, epochs={epoch}")

            model = LogisticClassifier(lr=lr, epochs=epoch)
            model.SGD(X_train, y_train)

            # Evaluate on validation set
            y_pred = model.predict(X_test)
            score = np.mean(y_pred == y_val)

            if score > best_params['score']:
                best_params['score'] = score
                best_params['lr'] = lr
                best_params['epochs'] = epoch

    print(f"\nBest parameters found:")
    print(f"Learning rate: {best_params['lr']}")
    print(f"Epochs: {best_params['epochs']}")
    print(f"Best accuracy score: {best_params['score']:.4f}")

    return best_params


def main_lr():
    X_scaled, y, X_test_scaled, test_ids = load_and_preprocess_data(None)

    best_params = hyperparameter_tuning(X_scaled, y)

    print("Training and evaluating model ...")
    X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=

    model = LogisticClassifier(
        lr=best_params['lr'],
        epochs=best_params['epochs']
    )
    model.SGD(X_train, y_train)

    y_val_pred = model.predict(X_test)

    print("\nTraining the final model ...")
    final_model = LogisticClassifier(
        lr=best_params['lr'],
        epochs=best_params['epochs']
    )
```

```
    final_model.SGD(X_scaled, y)

    test_predictions = final_model.predict(X_test_scaled)

    create_submission(test_predictions, test_ids)
```

B. K-means implementation

```
import numpy as np

""" Problem 2D """


"""
Randomly initializing centroids by selecting K data points
"""
def centroid_init(data, K):
    random_image = np.random.randint(0, data.shape[0], size=K)          # Choosing a random image
    centroids = data[random_image]                                      # Identyfying the centroid

    return np.array(centroids)                                          # Returning the centroids

"""
Calculating the euclidian distance between two data points
"""
def euclidian_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2)**2))        # Returning the distance between two data

"""
Assigning data to the cluster with the closest centroid
"""
def assign_data_to_cluster(data, centroids):
    """
    Parameters:
        data: input data points
        centroids: array of shape (K, n_features) containing current centroid positions

    """
    # Get number of clusters from centroid array
    K = len(centroids)
    # Initializing an array to store cluster assingments for each data point
    clusters = np.zeros(data.shape[0], dtype=int)

    # Loop through each data point in the dataset
    for i in range(data.shape[0]):
        # Initializing a minimum distance as infinity so first real distance wil be smaller
        min_distance = float('inf')

        # For each data point, loop through all centroids to find the closest one
        for j in range(K):
            # Calculating the euclidian distance between the current data point and the current c
            distance = euclidian_distance(data[i], centroids[j])

            # If this distance is smaller then the current minimum, the assignment is updated
            if distance < min_distance:
                min_distance = distance
                clusters[i] = j             # Assigning data point i to cluster j

    return clusters


"""
```

```
Calculate new centroid positions
"""
def update_centroids(data, clusters, K):
    """
    Parameters:
        data: the input dataset (array with shape (n_samples, n_features))
        K: number of clusters
    """
    # Initializing an array to store new centroid positions
    new_centroids = np.zeros((K, data.shape[1]))                    # (shape[1] is co

    # Calculate new centroid posotion for each cluster
    for i in range(K):
        # Getting all data points currently assigned to cluster i
        data_points_in_cluster = data[clusters == i]

        # Calculating the mean position of all in a cluster to get a new centroid
        # The mean is calculated across all features for those points
        new_centroids[i] = np.mean(data_points_in_cluster)

    return new_centroids

"""
Orchestrate the whole process of the K-means clustering
"""
def k_means_algorithm(data, K, max_iters=100, threshold=0.001):
    """
    Parameters:
        data: input dataset
        K: number of clusters
        max_iters: maximum number of iterations allowed
        threshold: convergence threshold for centroid movement
    """

    # Initializing centroid positions randomly from the data
    old_centroids = centroid_init(data, K)

    # Initializing the convergence flag and an iteration counter
    converged = False
    iters = 0

    # Creating an array to store visually interpretable centroids (the average faces)
    visual_centroids = np.zeros_like(old_centroids)

    # Main K-means loop - continue until convergence or max iterations reached
    while not converged and iters < max_iters:
        # print(f"This is iteration number {iters}")

        # Assigning each data point to the nearest centriod
        old_clusters = assign_data_to_cluster(data, old_centroids)

        # Calculating the new centroid positions based on cluster assignments
        new_centroids = update_centroids(data, old_clusters, K)

        # Calculating the visual centroids based for each cluster
        for i in range(K):
            # Getting all the images assigned to the current cluster
            cluster_images = data[old_clusters == i]
            # Calculating the mean image (average face) for the cluster
            visual_centroids[i] = np.mean(cluster_images, axis=0)
```

```
        # Calculating the average movement of centroids between iterations
        distance = np.mean([euclidian_distance(old_centroids[i], new_centroids[i]) for i in range
        # print(f"  Distance = {distance}\n")

        # Checking if the algorithm has converged (centroid movement is below threshold)
        if distance < threshold:
            # print(f"It is now converged!!\n")
            converged = True

        # Updating the centroid position for the next iteration
        old_centroids = new_centroids.copy()

        iters += 1

    print(f"Done with K-means after {iters} iterations!\n")
    return new_centroids, old_clusters, visual_centroids

""" Calculate the within-cluster sum of squares (inertia) """
def calculate_inertia(data, clusters, centroids):
    """
    Parameters:
    data: array of data points
    clusters: array of cluster assignments for each point
    centroids: array of centroid positions
    """

    # Initialize total distance counter
    total_distance = 0

    # Loop through each cluster
    for cluster_idx in range(len(centroids)):
        # Get all points assigned to current cluster
        cluster_points = data[clusters == cluster_idx]

        # Only process cluster if it contains points
        if len(cluster_points) > 0:
            # Get centroid for current cluster
            centroid = centroids[cluster_idx]
            # Calculate squared distances from points to centroid
            distances = np.sum((cluster_points - centroid)**2)
            # Add to total distance
            total_distance += distances

    return total_distance

""" Find the best clustering result by running k-means multiple times """
def find_best_clustering(data, normalized_data, K, n_tries=10):
    """
    Parameters:
    data: original (non-normalized) face data
    normalized_data: normalized face data for clustering
    K: number of clusters
    n_tries: number of clustering attempts
    """

    # Initialize best results with worst possible score
    best_inertia = float('inf')
    best_centroids = None
    best_clusters = None
```

```
    best_visual_centroids = None

    # Try k-means multiple times with different random initializations
    for i in range(n_tries):
        print(f"\nTry {i+1} of {n_tries}")

        # Run k-means clustering
        centroids, clusters, visual_centroids = k_means_algorithm(normalized_data, K)

        # Calculate quality score (inertia) for this attempt
        inertia = calculate_inertia(normalized_data, clusters, centroids)

        # If this attempt is better than previous best, update best results
        if inertia < best_inertia:
            best_inertia = inertia
            best_centroids = centroids
            best_clusters = clusters
            best_visual_centroids = visual_centroids
            print(f"New best inertia: {best_inertia}")

    # Return the results from the best clustering attempt
    return best_centroids, best_clusters, best_visual_centroids


if __name__ == "__main__":
    data = np.loadtxt('frey-faces.csv')          # Every row is an image with 20x28 pixels and the

    # Calculate normalization parameters
    data_mean = np.mean(data, axis=0)
    data_std = np.std(data, axis=0)

    # Normalize data for clustering
    normalized_data = (data - data_mean) / data_std

    K = 10  # or whatever number of clusters you want
    best_centroids, best_clusters, best_visual_centroids = find_best_clustering(data, normalized_
```

REFERENCES

[1] CodeAcademy. *Clustering: K-Means*. [Online, accessed 01-NOVEMBER-2024]. 2024. URL: https://www.codecademy.com/learn/dspath-unsupervised/modules/dspath-clustering/cheatsheet.

[2] CodeAcademy. *RandomForestClassifier*. [Online, accessed 01-NOVEMBER-2024]. 2024. URL: https://scikit-learn.org/1.5/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[3] CodeAcademy. *sklearn.tree.DecisionTreeClassifier*. [Online, accessed 01-NOVEMBER-2024]. 2024. URL: https://scikit-learn.org/0.15/modules/generated/sklearn.tree.DecisionTreeClassifier.html.

[4] Google for developers. *What is clustering?* [Online, accessed 21-0ctober-2024]. 2024. URL: https://developers.google.com/machine-learning/clustering/overview.

[5] EITCA. *How do we initialize the centroids in the custom k-means algorithm?* [Online, accessed 21-october-2024]. 2023. URL: https://eitca.org/artificial-intelligence/eitc-ai-mlp-machine-learning-with-python/clustering-k-means-and-mean-shift/custom-k-means/examination-review-custom-k-means/how-do-we-initialize-the-centroids-in-the-custom-k-means-algorithm/.

[6] GeeksForGeeks. *Clustering in Machine Learning*. [Online, accessed 18-0ctober-2024]. 2024. URL: https://www.geeksforgeeks.org/clustering-in-machine-learning/.

[7] GeeksForGeeks. *Descision Tree*. [Online, accessed 28-0ctober-2024]. 2024. URL: https://www.geeksforgeeks.org/decision-tree/.

[8] GeeksForGeeks. *Euclidian Distance*. [Online, accessed 23-0ctober-2024]. 2024. URL: https://www.geeksforgeeks.org/euclidean-distance/#euclidean-distance-formula.

[9] GeeksForGeeks. *Getting started with Classification*. [Online, accessed 18-0ctober-2024]. 2024. URL: https://www.geeksforgeeks.org/getting-started-with-classification/.

[10] GeeksForGeeks. *ML — Classification vs Clustering*. [Online, accessed 21-0ctober-2024]. 2024. URL: https://www.geeksforgeeks.org/ml-classification-vs-clustering/.

[11] GeeksForGeeks. *Random Forest Classifier using Scikit-learn*. [Online, accessed 29-0ctober-2024]. 2024. URL: https://www.geeksforgeeks.org/random-forest-classifier-using-scikit-learn/.

[12] Kaggle. *Image Segmentation with Kmeans*. [Online, accessed 21-0ctober-2024]. 2024. URL: https://www.kaggle.com/code/hal1001k/image-segmentation-with-kmeans.

[13] Paul M. Sutter. *Astronomy Jargon 101: Hertzsprung-Russel (HR) diagram*. [Online, accessed 21-0ctober-2024]. 2021. URL: https://www.universetoday.com/152202/astronomy-jargon-101-hertzsprung-russell-hr-diagram/.

[14] W3Schools. *K-means*. [Online, accessed 21-october-2024]. 2024. URL: https://www.w3schools.com/python/python_ml_k-means.asp.

[15] Wikipedia. *Logistic Regression*. [Online, accessed 01-NOVEMBER-2024]. 2024. URL: https://en.wikipedia.org/wiki/Logistic_regression.