Logo similarity - Veridion

Toma Sabin-Sebastian

For a project, I had to identify similarities between certain logos of different companies on the market. I used **unsupervised learning**, capturing the most important features using different f**eature extraction methods**. After these features were extracted, a **clustering algorithm grouped** them based on **their numerical representations**. I applied unsupervised learning because the images had no labels and without these labels, a supervised learning algorithm could not have been trained.

I continued with a script that loads the **Parquet file** containing company domain names into a Pandas DataFrame, then generates **logo URLs** using the **Clearbit API** by appending each domain to the Clearbit logo service URL. It applies this function to the "domain" column, creating a new "logo\_url" column with the corresponding logo links. Furthermore, the modified DataFrame is **saved as a CSV file** (logos\_with\_urls.csv) for further use and a confirmation message is printed. This process automates the retrieval of company logos using their domain names.

After that, I made a script that automates the process of **downloading company logos** from **Clearbit** and **Google Favicon** and saves them locally while keeping track of their file paths. It first loads a CSV file (logos\_with\_urls.csv) that contains company domain names, then creates a ***"logos"*** folder to store the downloaded images. For each domain, it generates a filename by replacing dots with underscores and checks whether the logo already exists to avoid redundant downloads. If the logo is missing, the script first attempts to download it from **Clearbit**, if that fails, it falls back to **Google Favicon** as an alternative source. Once all logos are processed, the script updates the **CSV file** (logos\_with\_paths.csv) with the paths of successfully downloaded logos. It also prints status messages to indicate whether a logo was downloaded from Clearbit, Google Favicon or if both attempts failed. This ensures an efficient and structured way to retrieve, store, and track company logos for further use.

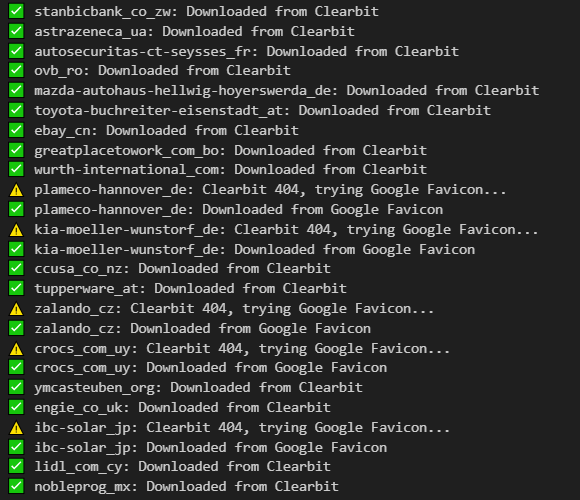
 Finally, I split the dataset into 80% training and 20% testing sets, resulting in 2,670 samples in the training set and 668 samples in the test set.

Figure 1. Status messages for every logo.

The first approach that I decided to serve as a baseline is **GMM-SIFT** method.

A **Gaussian Mixture model** is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters.

**SIFT** is a robust algorithm designed to identify and describe local features in images that are invariant to scale, rotation, and partially invariant to affine transformations and illumination changes. This means that SIFT can detect the same features in an image even if the image is resized, rotated, or viewed under different lighting conditions.

Key steps in the SIFT algorithm:

1. Scale-Space Extrema Detection: Identify key points invariant to scale by constructing a scale-space representation using Gaussian filters. Compute the Difference of Gaussians (DoG) and detect local extrema as potential key points.

2. Keypoint Localization: Refine key point positions by fitting a quadratic function. Discard low-contrast or poorly localized points for robustness.

3. Orientation Assignment: Assign orientations based on local image gradients, ensuring rotation invariance. The dominant gradient direction is determined using an orientation histogram.

4. Keypoint Descriptor: Construct a 128-dimensional descriptor by sampling gradient magnitudes and orientations around each key point. Normalize to enhance robustness to illumination changes.

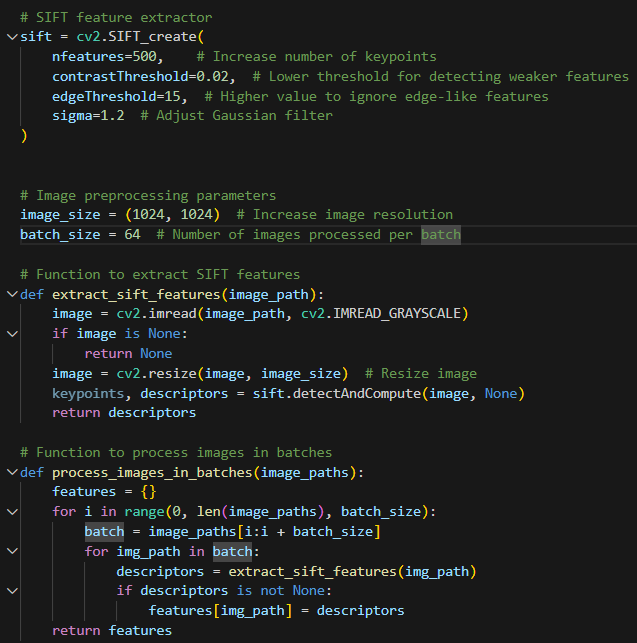


Figure 2. SIFT feature extraction method

The code defines a **SIFT-based feature extraction pipeline** for processing images in batches. It first initializes a **SIFT extractor** with customized parameters to detect and describe keypoints efficiently. The **extract\_sift\_features()** function reads an image in grayscale, resizes it and extracts **SIFT descriptors**, ensuring that only valid images are processed. To enhance efficiency, **process\_images\_in\_batches()** processes images in **batches**, iterating through the dataset and extracting descriptors for each image, which are stored in a dictionary mapped by image paths. This approach ensures scalable and optimized feature extraction, useful for downstream tasks like **image clustering or classification**.

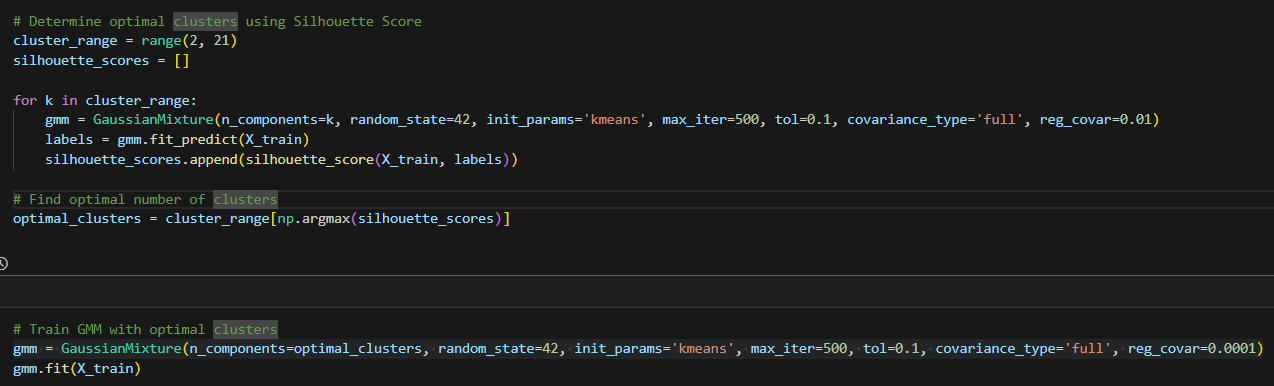


Figure 3. Gaussian mixture model

I trained the **first GMM model** to evaluate different cluster numbers (from 2 to 21) using the **Silhouette Score**, which helps determine how well-separated the clusters are. Since GMM requires the number of clusters (n\_components) to be specified in advance, I tested multiple values to find the optimal one. The **Silhouette Score** was computed for each model, and the value that maximized this score was chosen as the best cluster count.

Once the optimal number of clusters was identified, I trained the **second GMM model** using this specific number of clusters. The goal of this step was to fit the best possible model to the dataset and assign final cluster labels. The second training ensures that the clustering is done using the most suitable number of clusters, leading to better-defined and more meaningful groupings in the data.

* n\_components = optimal\_clusters: the **number of clusters (Gaussian components)** in the model(determined using Silhouette score)
* random\_state = 42: ensures **reproducibility** by setting a fixed seed for random number generation. Controls the random seed given to the method chosen to initialize the parameters(init\_params) and controls the generation of random samples from the fitted distribution.
* init\_params = k-means++: defines how the initial means of the Gaussians are initialized. Used the k-means++ algorithm to intelligently initialize the Gaussian means before running GMM. K-means++ ensures that initial cluster centers are well spread out, reducing the chances of poor convergence. It picks the centroid randomly and selects subsequent centroids far from the already chosen ones.

init\_params = k-means: uses vanilla k-means clustering to initialize the Gaussian means. This runs k-means clustering first, then sets the GMM means to the final k-means centroids. The downside is that k-means may still pick bad initial centroids, leading to slow convergence or suboptimal solutions.

init\_params = random: responsabilities are initialized randomly.

init\_params = random\_from\_data: initial means are randomly selected data points.

* max\_iter = 500: sets the maximum number of iterations.
* tol = 0.1: specifies the convergence threshold for stopping the iterations. Prevents unnecessary computation once the model has sufficiently converged. Higher values allow more iterations for model convergence, but excessive values can slow down training.
* covariance\_type = full: defines how the covariance matrices of the Gaussian components are modeled. ―Full‖ means that each component has its own general covariance matrix.

Full means that each component has its own general covariance matrix.

covariance\_type = tied: all components share the same general covariance matrix. covariance\_type = diag: each component has its own diagonal covariance matrix. covariance\_type = spherical: each component has its own single variance.

* reg\_covar=0.001: Adds a regularization term to the covariance matrices to prevent numerical instabilities. Helps when clusters have very small variances, avoiding division by near-zero values.

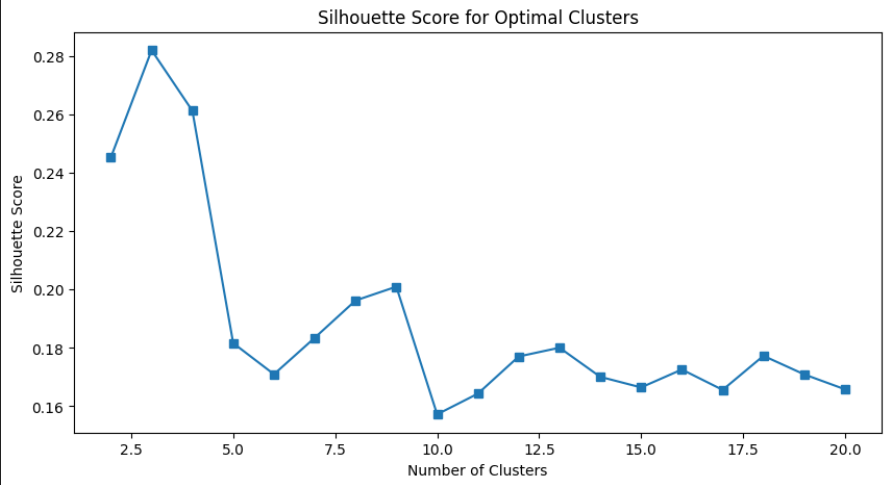


Figure 4. Silhouette score per number of clusters

Based on the silhouette score, **k = 3** is the best choice, as it provides the highest score, meaning the clusters are well-separated and compact.

As k increases, clusters may become too fine-grained, leading to poor separation (lower silhouette scores).

For very low k (k = 2), while the silhouette score is still relatively high, it may not capture sufficient variation in the data.

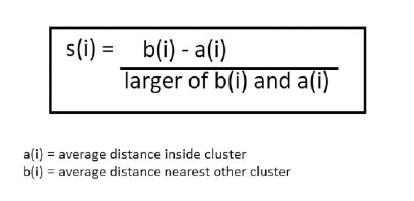


Figure 5. Silhouette score formula

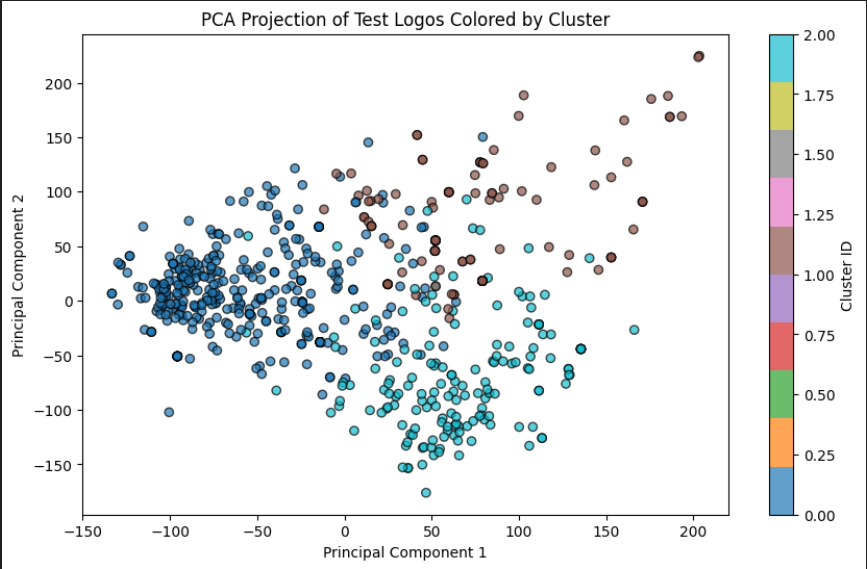
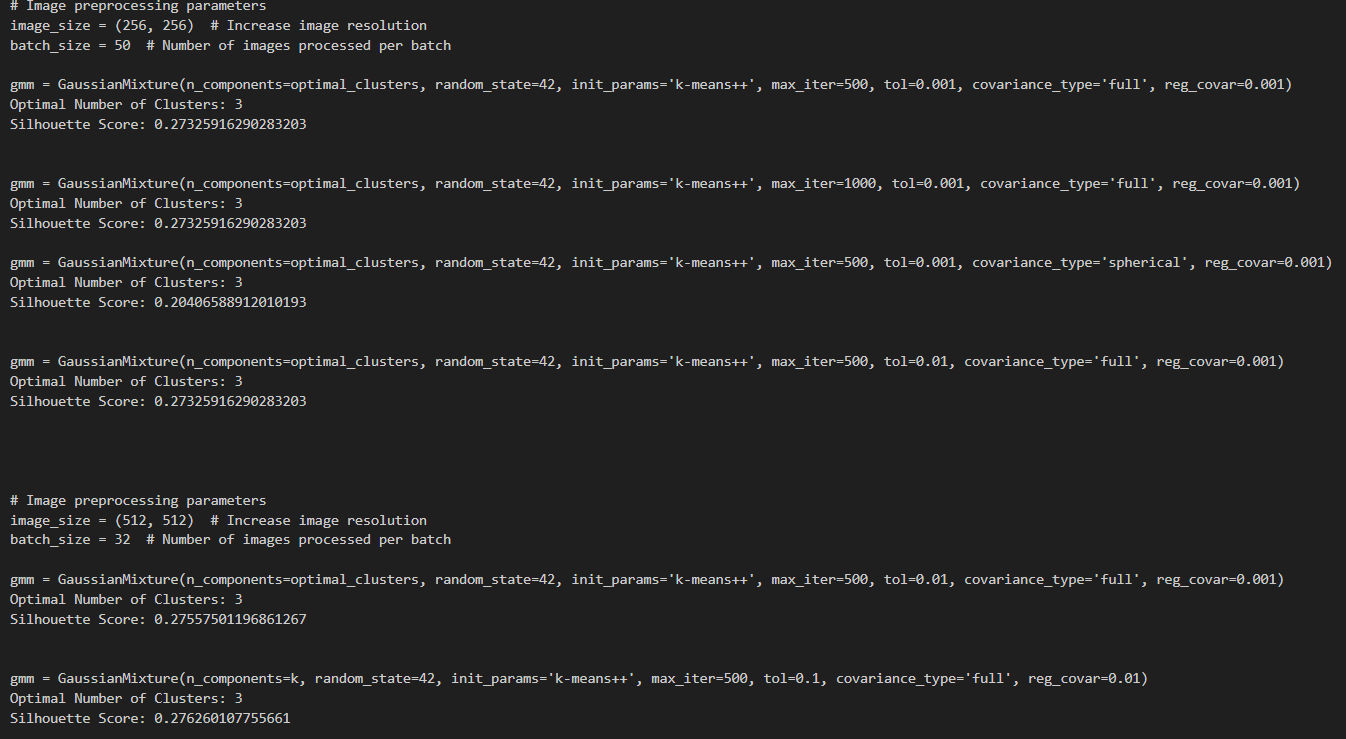


Figure 6. PCA projection

This **PCA Projection Plot** visualizes the clustering of test logo images in a **2D space** by reducing the high-dimensional SIFT feature space into two principal components. Each point represents a test image, and the colors indicate cluster assignments from the **Gaussian Mixture Model (GMM)**. The distribution suggests that some clusters are well-separated, while others overlap, indicating possible similarities between certain logos. The presence of distinct groups implies that the clustering captures meaningful structure in the data, but the overlap suggests that some clusters may need refinement or that more discriminative features could improve separation. The color bar on the right maps **Cluster IDs** to their respective colors, helping interpret the grouping visually.

Figure 7. Results(one of the results)



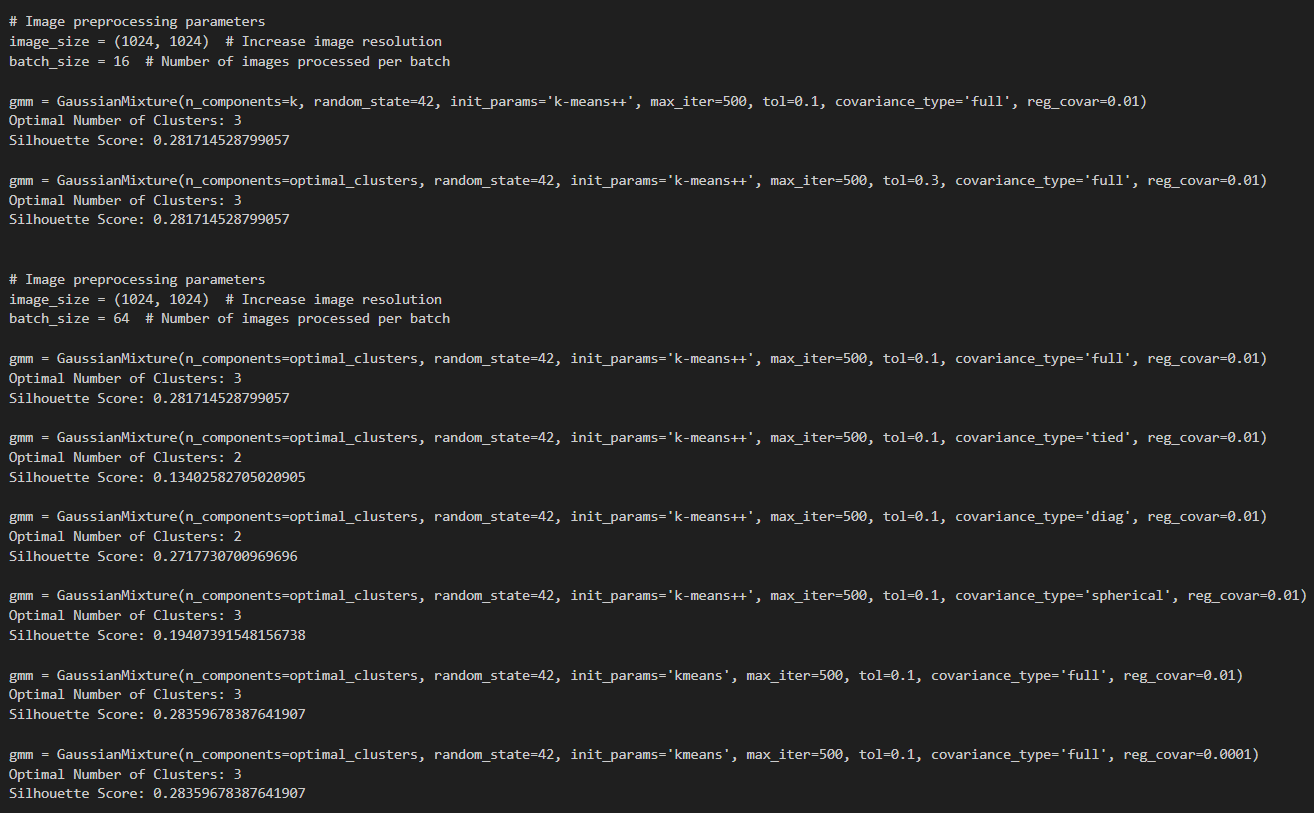
Figure 8. Hyperparameter tuning

Figure 9. Hyperparameter tuning

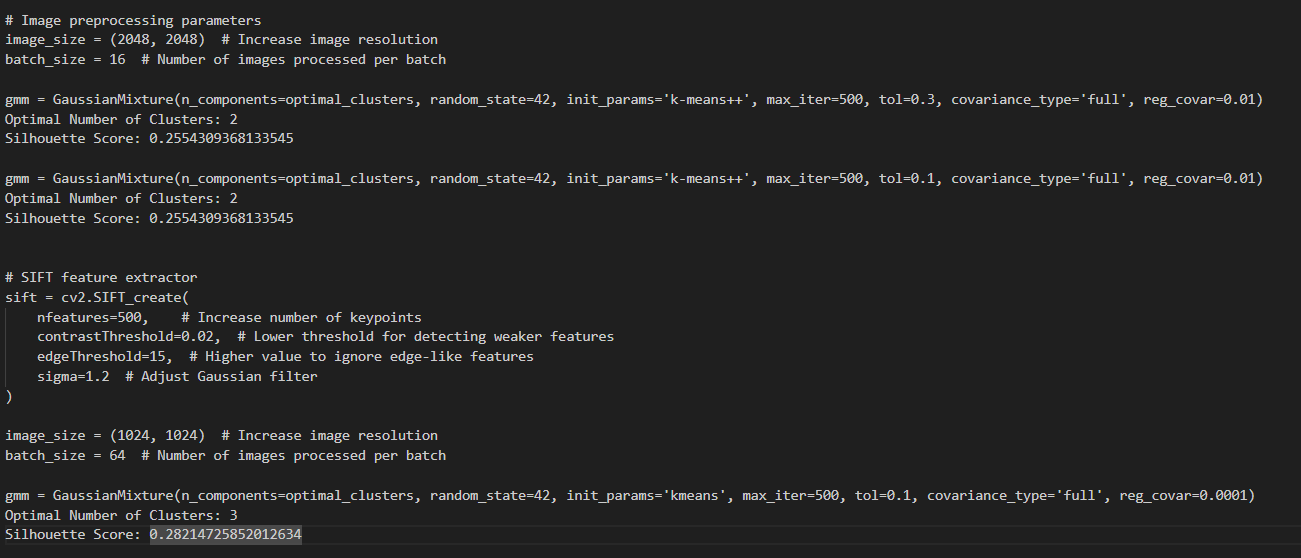


Figure 10. Hyperparameter tuning

In my second approach, I used the same model(GMM) and feature extraction method(SIFT), but I added another feature extracting method, **HOG**.

“In computer vision and image processing, the Histogram of Oriented Gradients (HOG) approach is used to explain the characteristics of a particular piece of data. It takes into account how frequently a particular area of a picture displays a gradient orientation.”

How HOG works:

1. **Gradient computation**: the image is converted to **grayscale** (if it is colored). Gradients (changes in intensity) are computed using **filters** in both horizontal and vertical directions that helps in capturing edge information.

2. **Cell and block division**: The image is divided into small regions called **cells** (eg. 8x8 pixels). A group of cells forms a **block** (eg. 2x2 cells) which is used for normalization.

3. For each cell, a histogram of gradient orientations (angles) is created. The histogram bins represent gradient angles (0°–180° or 0°–360°), and the values represent the magnitude of the gradients.

4. **Block normalization**: To handle illumination changes, the histogram values are normalized across larger overlapping blocks.

5. **Feature vector formation**: The final HOG descriptor is a concatenated vector of all normalized histograms from the image. This vector is then used as input for machine learning models like **Support Vector Machines (SVM)** for object detection.

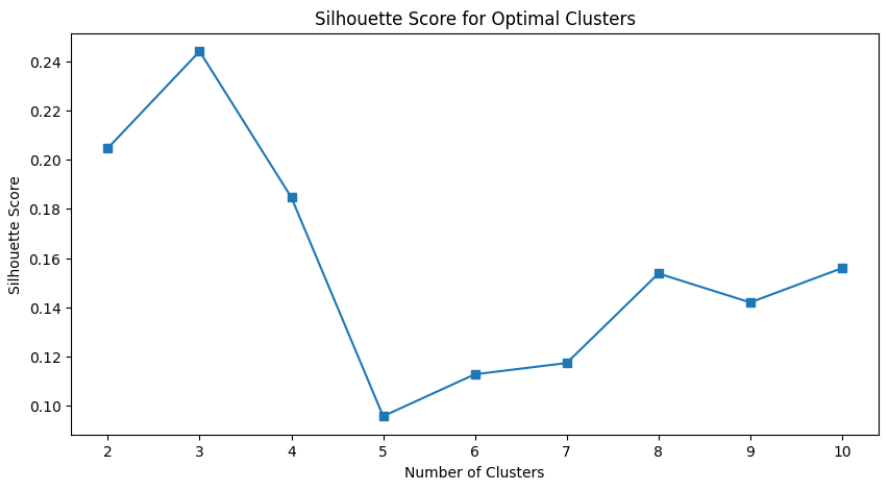


Figure 11. Silhouette score per number of clusters

**Peak at k = 3:** The highest silhouette score (~0.24) is achieved at **k = 3**, indicating that using **3 clusters** results in the most well-separated and compact clusters.

**Sharp Decline at k = 5:** There is a significant drop in the silhouette score at **k = 5**, suggesting that increasing the number of clusters beyond this point leads to less distinct separations.

**Gradual Improvement Beyond k = 6:** The silhouette score slowly increases again, but does not surpass the peak at k = 3.

**Fluctuations at Higher k Values:** The scores remain relatively low and fluctuate, indicating that adding more clusters does not necessarily improve separation quality.

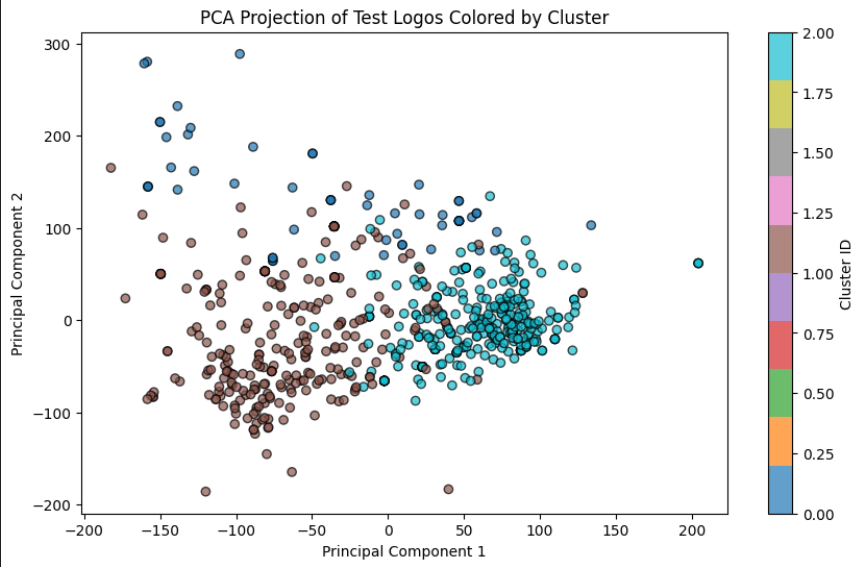


Figure 12. PCA projection

**Two Main Clusters:** The data points are primarily divided into two distinct groups, which align with the **k= 3 optimal clustering** result from the silhouette score analysis.

**Overlapping Regions:** While there is some separation, the clusters are not completely distinct, meaning that some images have similar feature representations and may not be easily distinguishable.

**Cluster Density:** One cluster (cyan/blue) appears more compact and densely packed, while the brown-colored cluster is more spread out, indicating variability in intra-cluster distances.

Figure 13. Results

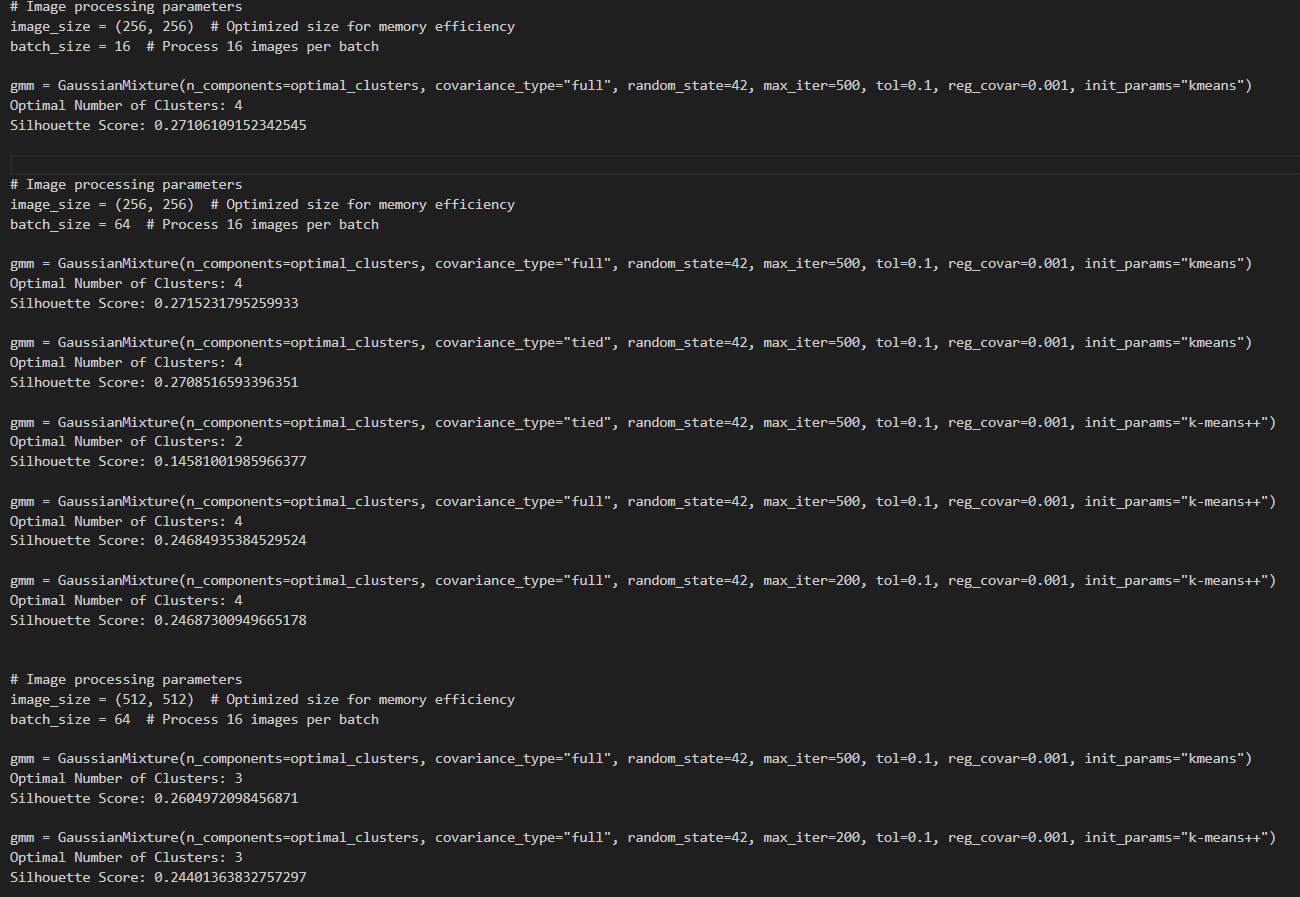


Figure 14. Hyperparameter tuning

In my third approach, I used the same feature extraction methods as before, but with another model, **DBSCAN**.

“DBSCAN is a **density-based clustering algorithm** that**groups data points that are closely packed together and marks outliers as noise** based on their density in the feature space. It  identifies clusters as dense regions in the data space, separated by areas of lower density.

Unlike K-Means or hierarchical clustering, which assume clusters are **compact and spherical**, DBSCAN excels in handling real-world data irregularities such as:

* **Arbitrary-Shaped Clusters**: Clusters can take any shape, not just circular or convex.
* **Noise and Outliers**: It effectively identifies and handles noise points without assigning them to any cluster.”
* eps = 2 : The maximum distance between two samples for one to be considered as in the neighborhood of the other. This is not a maximum bound on the distances of points within a cluster. This is the most important DBSCAN parameter to choose appropriately for your data set and distance function.
* min\_samples = 20 : The number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself. If min\_samples is set to a higher value, DBSCAN will find denser clusters, whereas if it is set to a lower value, the found clusters will be more sparse.
* metric = “Euclidean” : The metric to use when calculating distance between instances in a feature array. Other metrics: “Manhattan”, “Minkowski”, “Cosine Distance”.
* metric\_params = {} : Additional keyword arguments for the metric function.
* algorithm = “brute” : The algorithm to be used by the NearestNeighbors module to compute pointwise distances and find nearest neighbors. See NearestNeighbors module documentation for details. Other algorithms : “auto”, “ball\_tree”, “kd\_tree”.
* leaf\_size = 30 : This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.
* p = 2 : The power of the Minkowski metric to be used to calculate distance between points.
* n\_jobs = -1 : The number of parallel jobs to run. -1 means using all processors.

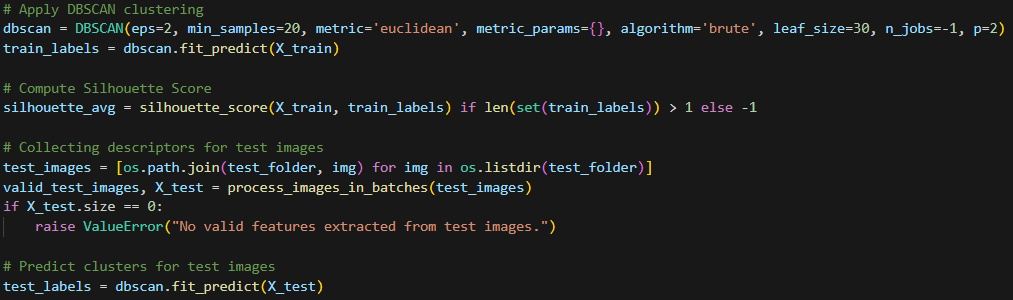


Figure 15. DBSCAN model

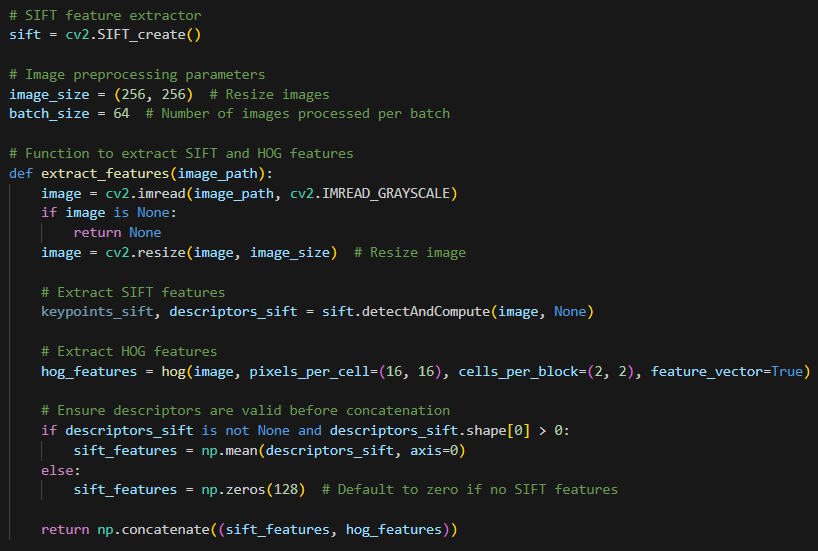


Figure 15. SIFT and HOG features

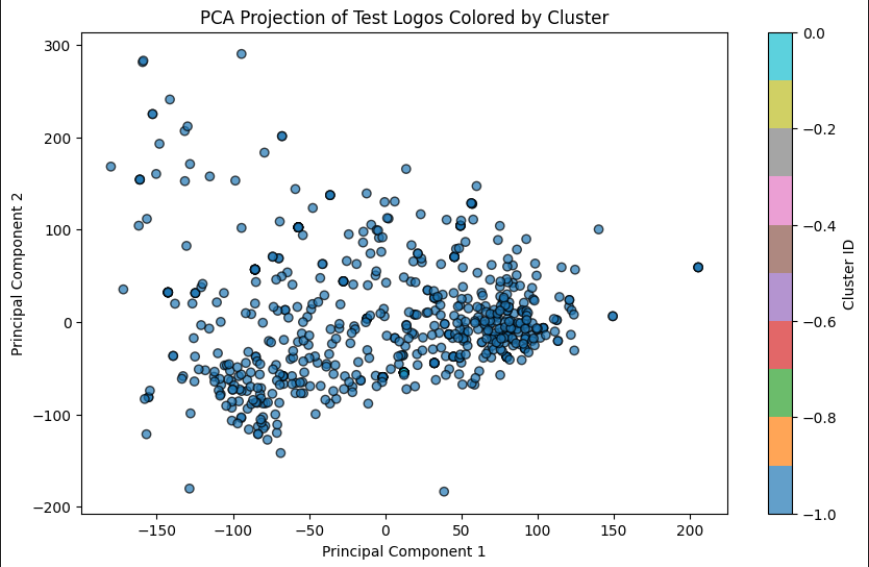


Figure 16. PCA projection

The clustering process seems to have classified most points as **noise (-1)**, leading to a misleading visualization.

A large portion of the data points are clustered together, indicating a **high-density region** where many logos share similar features.

Figure 17. Results

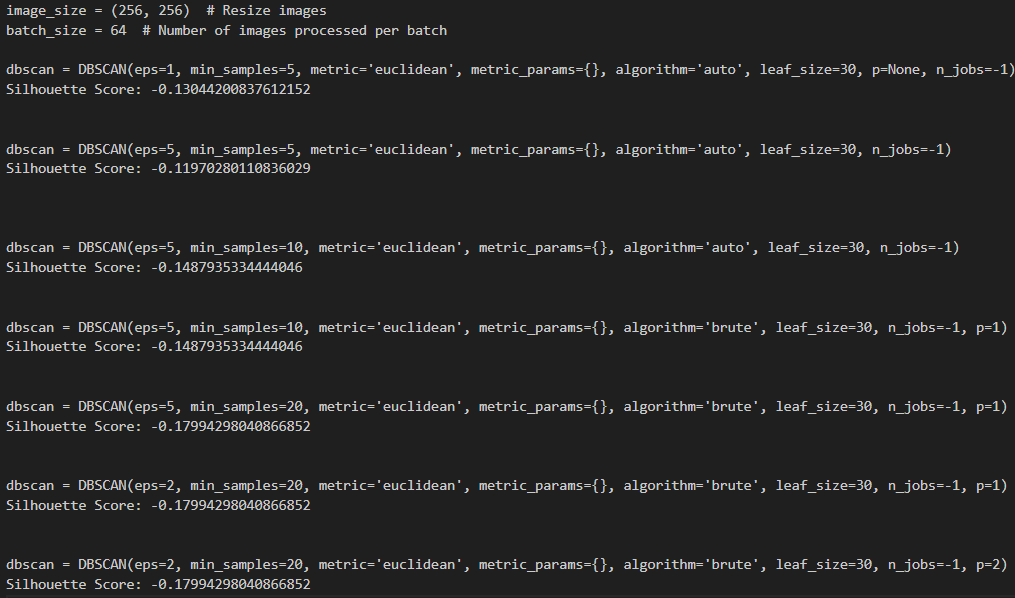


Figure 18. Hyperparameter tuning

For my final ML approach, I used KMeans for clustering and I extracted the features with pre-trained CNN.

“K-means clustering is a technique used to organize data into **groups based on their similarity**. The algorithm works by first randomly picking some central points called**centroids**and each data point is then assigned to the closest centroid forming a cluster. After all the points are assigned to a cluster the centroids are updated by finding the average position of the points in each cluster. This process repeats until the centroids stop changing forming clusters. The goal of clustering is to divide the data points into clusters so that similar data points belong to same group.”

* n\_clusters = k : The number of clusters to form as well as the number of centroids to generate.
* init = “k-means++” : selects initial cluster centroids using sampling based on an empirical probability distribution of the points’ contribution to the overall inertia. This technique speeds up convergence. The algorithm implemented is “greedy k-means++”. It differs from the vanilla k-means++ by making several trials at each sampling step and choosing the best centroid among them.

“random” : choose n\_clusters observations (rows) at random from data for the initial centroids.

* n\_init = 10/auto : Number of times the k-means algorithm is run with different centroid seeds. The final results is the best output of n\_init consecutive runs in terms of inertia. Several runs are recommended for sparse high-dimensional problems.
* max\_iter = 200 : Maximum number of iterations of the k-means algorithm for a single run.
* tol = 0.1 : Relative tolerance with regards to Frobenius norm of the difference in the cluster centers of two consecutive iterations to declare convergence.
* random\_state = 42 : Determines random number generation for centroid initialization. Use an int to make the randomness deterministic.
* algorithm = “lloys/elkan” : K-means algorithm to use. The classical EM-style algorithm is "lloyd". The "elkan" variation can be more efficient on some datasets with well-defined clusters, by using the triangle inequality.

**Residual Network:**In order to solve the problem of the vanishing/exploding gradient, this architecture introduced the concept called **Residual Blocks**. In this network, we use a technique called ***skip connections***. The skip connection connects activations of a  layer to further layers by skipping some layers in between. This forms a residual block. Resnets are made by stacking these residual blocks together.

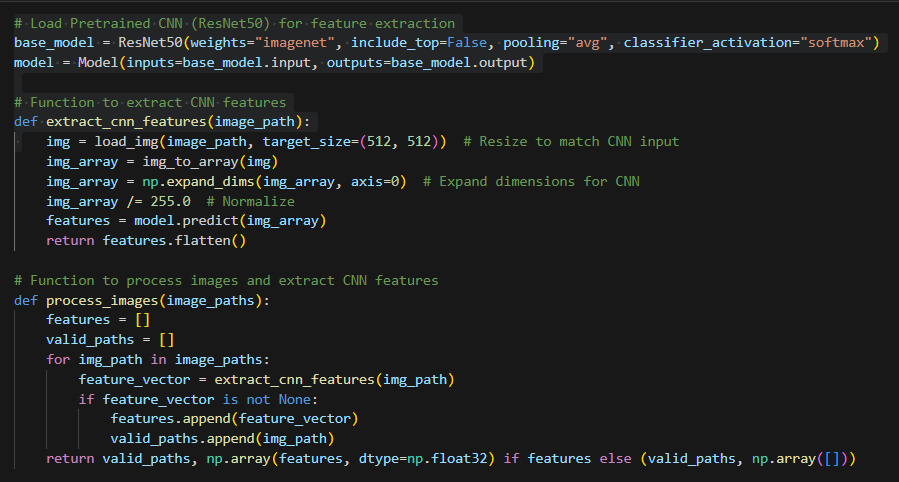


Figure 19. ResNet

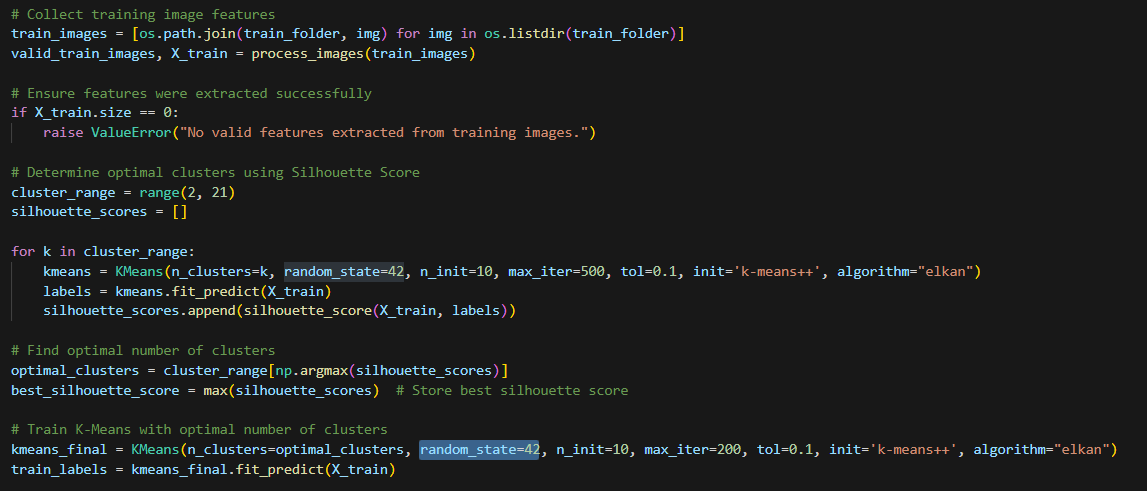


Figure 20. KMeans model

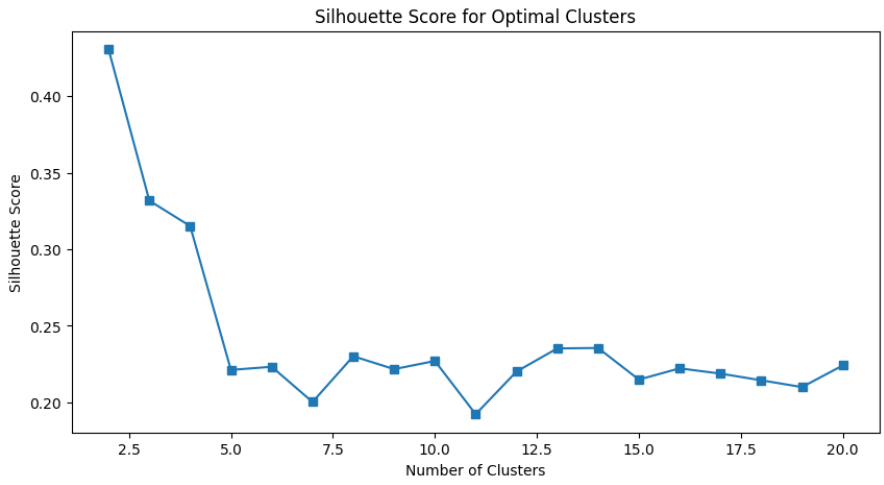


Figure 21. Silhouette score for optimal clusters

The highest Silhouette Score occurs at **k = 2**, meaning **two clusters provide the best separation**.

The score **drops sharply** as k increases, indicating that more clusters lead to worse-defined groups.

Beyond **k = 5**, the scores stabilize but remain low (around 0.2–0.25), suggesting that increasing k further does **not** significantly improve clustering.

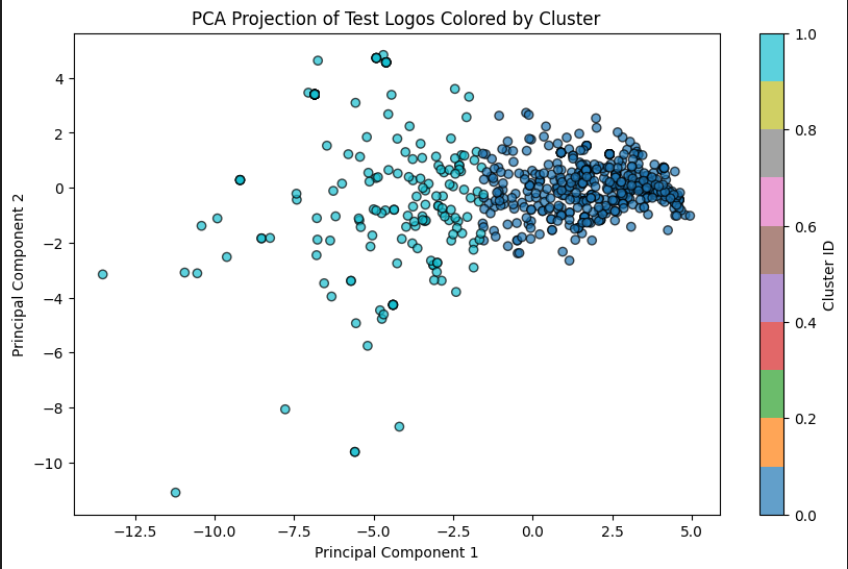


Figure 22. PCA projection

Clusters are not well-separated : The points are **spread out** rather than forming distinct, tight groups. Some clusters **overlap**, indicating that the features might not be strongly differentiating logos.

Densely packed regions : There's a dense concentration of points on the right side, suggesting **many images share similar features**. The left side has **outliers**, possibly unique logos or ones with very different features.

Possible sub-structure in clusters : Some colors appear to blend together, meaning **certain clusters may not be very distinct**. If the clusters were well-separated, we would see **clear, non-overlapping groups**.

Figure 23. Results

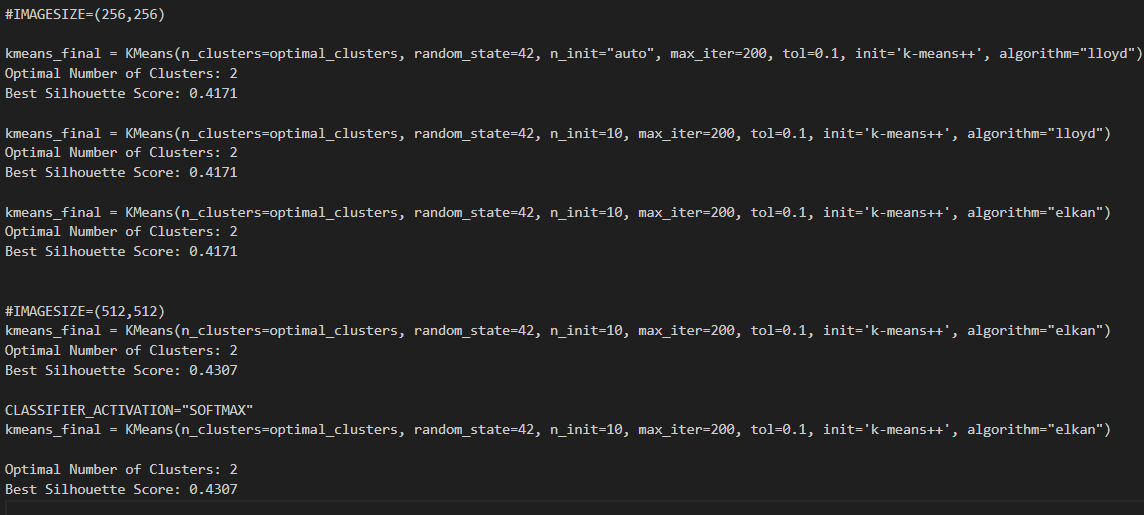


Figure 24. Hyperparameter tuning

Below, it is my full code for my approach without using ML.

Firstly, I resized all images to a fixed size(128x128) and then I extracted the features using SIFT descriptos, HOG features and HSV color histogram and then I concatenate all three in one feature vector.

Secondly, I computed cosine similarity between every pair of feature vectors. If similarity is greater than threshold, it connects the images as neighbors in graph. The resulting graph is an adjacency list where graph[i] lists indices of images similar to image i.

Finally, I treated the graph as an undirected network and used DFS(Depth-First Search) to find connected components and loaded all the images from the directory.

*import cv2*

*import numpy as np*

*from skimage.feature import hog*

*from sklearn.preprocessing import normalize*

*from itertools import combinations*

*from scipy.spatial.distance import cosine*

*import os*

*# Resize all images to fixed size to ensure consistent HOG dimensions*

*def resize\_image(img, size=(128, 128)):*

*return cv2.resize(img, size, interpolation=cv2.INTER\_LINEAR)*

*# Extract SIFT descriptor, average-pooled to fixed length*

*def extract\_sift(img):*

*gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)*

*sift = cv2.SIFT\_create()*

*kp, des = sift.detectAndCompute(gray, None)*

*if des is None:*

*return np.zeros(128)*

*return np.mean(des, axis=0)*

*# Extract fixed-length HOG descriptor*

*def extract\_hog(img):*

*gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)*

*hog\_feat = hog(gray, pixels\_per\_cell=(8, 8),*

*cells\_per\_block=(2, 2),*

*feature\_vector=True)*

*return hog\_feat*

*# Extract normalized HSV histogram*

*def extract\_hsv\_hist(img, bins=(8, 8, 8)):*

*hsv = cv2.cvtColor(img, cv2.COLOR\_BGR2HSV)*

*hist = cv2.calcHist([hsv], [0, 1, 2], None, bins,*

*[0, 180, 0, 256, 0, 256])*

*return cv2.normalize(hist, hist).flatten()*

*# Combined feature vector: SIFT + HOG + HSV*

*def extract\_combined\_features(img):*

*img = resize\_image(img)  # Force consistent size*

*sift\_feat = extract\_sift(img)*

*hog\_feat = extract\_hog(img)*

*hsv\_feat = extract\_hsv\_hist(img)*

*combined = np.concatenate([sift\_feat, hog\_feat, hsv\_feat])*

*return normalize(combined.reshape(1, -1))[0]*

*# Build graph of similar images using cosine similarity*

*def build\_similarity\_graph(feature\_vectors, threshold=0.9):*

*n = len(feature\_vectors)*

*graph = [[] for \_ in range(n)]*

*for i, j in combinations(range(n), 2):*

*sim = 1 - cosine(feature\_vectors[i], feature\_vectors[j])*

*if sim > threshold:*

*graph[i].append(j)*

*graph[j].append(i)*

*return graph*

*def connected\_components(graph):*

*visited = set()*

*clusters = []*

*def iterative\_dfs(start):*

*stack = [start]*

*cluster = []*

*while stack:*

*node = stack.pop()*

*if node not in visited:*

*visited.add(node)*

*cluster.append(node)*

*stack.extend(graph[node])*

*return cluster*

*for i in range(len(graph)):*

*if i not in visited:*

*cluster = iterative\_dfs(i)*

*clusters.append(cluster)*

*return clusters*

*# Load all images from a folder*

*def load\_images\_from\_folder(folder):*

*images = []*

*filenames = []*

*for filename in os.listdir(folder):*

*path = os.path.join(folder, filename)*

*img = cv2.imread(path)*

*if img is not None:*

*images.append(img)*

*filenames.append(filename)*

*return images, filenames*

*# === Main Execution ===*

*folder\_path = "dataset/logos"*

*images, names = load\_images\_from\_folder(folder\_path)*

*# Feature extraction*

*features = [extract\_combined\_features(img) for img in images]*

*# Optional: sanity check on vector lengths*

*vec\_lengths = [len(f) for f in features]*

*assert all(l == vec\_lengths[0] for l in vec\_lengths), "Inconsistent feature vector lengths!"*

*# Clustering*

*graph = build\_similarity\_graph(features, threshold=0.9)*

*clusters = connected\_components(graph)*

*# Output results*

*for idx, cluster in enumerate(clusters):*

*print(f"Cluster {idx + 1}: {[names[i] for i in cluster]}")*

I found 207 clusters, but only one larger cluster(with over 1000) logos, while the rest are mostly tiny or singleton clusters.

Bibliography:

1. [GaussianMixture — scikit-learn 1.6.1 documentation](https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html)
2. [Gaussian Mixture Model - GeeksforGeeks](https://www.geeksforgeeks.org/gaussian-mixture-model/)
3. [Describe the concept of scale-invariant feature transform (SIFT) - GeeksforGeeks](https://www.geeksforgeeks.org/describe-the-concept-of-scale-invariant-feature-transform-sift/)
4. [Introduction to Histogram of Oriented Gradients (HOG) – Pythonista Planet](https://pythonistaplanet.com/histogram-of-oriented-gradients/)
5. [DBSCAN — scikit-learn 1.6.1 documentation](https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html)
6. [DBSCAN Clustering in ML | Density based clustering - GeeksforGeeks](https://www.geeksforgeeks.org/dbscan-clustering-in-ml-density-based-clustering/)
7. [K means Clustering - Introduction - GeeksforGeeks](https://www.geeksforgeeks.org/k-means-clustering-introduction/)
8. [KMeans — scikit-learn 1.6.1 documentation](https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html)
9. [Residual Networks (ResNet) - Deep Learning - GeeksforGeeks](https://www.geeksforgeeks.org/residual-networks-resnet-deep-learning/)