	Group Members:  Nils Dunlop, e-mail: gusdunlni@student.gu.se  Francisco Aleiandro Frazo Piza, e-mail: guserafr@student.gu.se
n [1]:	Francisco Alejandro Erazo Piza, e-mail: guserafr@student.gu.se  Chukwudumebi Ubogu, e-mail: gusuboch@student.gu.se  Part 1: Implement the K-means algorithm  # Import libraries import numby as no
n [2]:	<pre>import numpy as np import matplotlib.pyplot as plt from PIL import Image import os from math import log2  # Import iris dataset from sklearn.datasets import load_iris iris = load_iris() X = iris.data</pre>
	<pre>def kmeans(x, K, n_init):     Implementation of the K-means clustering algorithm.     # Initialize centroids randomly     centroids = x[np.random.choice(len(x), K, replace=False)]     labels = np.zeros(len(x))</pre>
	<pre># Run until convergence while True:     prev_labels = labels.copy()  # Assign data points to the nearest centroid distances = np.linalg.norm(x[:, np.newaxis] - centroids, axis=2) labels = np.argmin(distances, axis=1)  # Update centroids</pre>
F47.	<pre>centroids = np.array([np.mean(x[labels == k], axis=0) if np.sum(labels == k) &gt; 0 else np.random.rand(x.shape[1]) for k in range(K)])  # Check for convergence if np.array_equal(prev_labels, labels):     break  return centroids, labels</pre>
	<pre>centroids, labels = kmeans(X, 3, 1000)  print("Centroids:") print(centroids) print() print("Labels:") print("Labels:") print(labels)  Centroids: [[6.85     3.07368421 5.74210526 2.07105263]</pre>
	[[6.85
	<pre># Visualize the results u_labels = np.unique(labels)  for i in u_labels:     plt.scatter(X[labels == i, 0], X[labels == i, 1], label=i) plt.scatter(centroids[:, 0], centroids[:, 1], s=80, color="black", label="Centroids", marker="x") plt.legend(bbox_to_anchor=(1.3, 1), borderaxespad=0) plt.show()</pre>
	4.5 -
	3.5 -
	2.5 -
	If you run this algorithm multiple times, do you get the same result every time? If so, why; if not, how do you determine which result is the best one?  If we run this algorithm multiple times, we will not get the same result every time. This is because the sensitivity to the initial centroids, so the algorithm will converge to different local minima, that is, different initilizations of the centroids will lead to a different local minima. To determine which result is the best one, we can use the elbow method to determine the optimal number of clusters. The elbow method involves plotting the sum of squared errors (SSE) for different values of K. The elbow in the curve indicates the optimal value of K.  How do you choose what K value to use?
	The Elbow Method involves plotting the sum of squared errors (SSE) for different values of K. The elbow in the curve indicates the optimal value of K. The optimal K value is the value that is at the inflection point on the curve where the SSE begins to decrease in a linear way. It consists of the following steps:  1. Run the K-means clustering algorithm for a range of K values (e.g. K=1 to 10)  2. For each value of K, calculate the sum of squared errors (SSE)  3. Plot SSE vs K (number of clusters)  4. Identify the elbow in the plot (the point where the SSE curve starts to bend) and select this as the best K value for the data.
	Another method is the Silhouette method. The silhouette value measures how similar a point is to its own cluster (cohesion) compared to other clusters (separation). The range of the Silhouette value is between +1 and -1. A high value is desirable and indicates that the point is placed in the correct cluster. If many points have a negative Silhouette value, it may indicate that we have created too many or too few clusters.  Implement both SSE and the Silhouette score; use them to address these questions.  # Implementing SSE
	<pre>def calculate_sse(x, centroids, labels):     """     Calculate the sum of squared errors (SSE) for a given clustering.     """     sse = np.sum((x - centroids[labels])**2)     return sse  # Calculate SSE sse = calculate_sse(X, centroids, labels)</pre>
[7]:	<pre>print(f'SSE: {sse}') SSE: 78.85144142614601  # Elbow Method def plot_elbow_method(data, max_k, n_init=10):     """     Plot the sum of squared errors (SSE) for different values of K.     """     sse_values = []</pre>
	<pre>for k in range(1, max_k + 1):     sse = 0  for _ in range(n_init):         centroids, labels = kmeans(data, k, n_init=10)         sse += calculate_sse(data, centroids, labels)  sse_values.append(sse / n_init)</pre>
[0]	<pre># Plot plt.plot(range(1, max_k + 1), sse_values, marker='0') plt.xlabel('Number of Clusters (K)') plt.ylabel('Sum of Squared Errors (SSE)') plt.title('Elbow Method') plt.show()</pre> <pre>plot_elbow_method(X, 15, n_init=50)</pre>
[8]:	Elbow Method  700  600  W 500
	Squared From 200 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 - 800 -
	100 - 2 4 6 8 10 12 14
	Number of Clusters (K)  On the x-axis, we have the number of clusters (k). On the y-axis, there's the sum of squared errors (SSE) for each case of K clusters, on the same dataset. This tells us how far away the points in a cluster are from the cluster center. A high SSE means the points are spread out not very close to the center, and smaller number indicated a tigher cluster. To find the "elbow" in the graph, we refer to the point where the SSE curve starts to bend. After this point, choosing more clusters make little difference to the tightneses of clusters. If we look at the graph we see a big drop from 1 to 2 clusters, and then from 2 to 3 clusters. After 3 clusters, the SSE improves at a slower rate. So, the 'elbow' is around 3 clusters. This means that 3 is a good number of clusters to use for our dataset, which in fact is correct, because we know that the iris dataset have 3 labels "setosa", "versicolor" and "virginica". With 3 clusters, each one is different enough from the others, but we also don't have too many clusters to manage.
	<pre># Implementing the Silhouette Score def calculate_silhouette_score(x, labels):     """     Calculate the silhouette score for a given clustering.     """     num_samples = len(x)     silhouette_values = np.zeros(num_samples)  for i in range(num_samples):     # Calculate the supersed distance from each comple to all other complex in the comp eluctor.</pre>
	# Calculate the average distance from each sample to all other samples in the same cluster intra_cluster_dist = np.mean(np.linalg.norm(x[labels == labels[i]] - x[i], axis=1))  # Calculate the average distance from each sample to all samples in the nearest other cluster other_clusters_dist = np.min([np.mean(np.linalg.norm(x[labels == k] - x[i], axis=1)) for k in set(labels) if k != labels[i]])  # Calculate the silhouette score for each sample silhouette_values[i] = (other_clusters_dist - intra_cluster_dist) / max(other_clusters_dist, intra_cluster_dist)
	<pre># Calculate the overall silhouette score silhouette = np.mean(silhouette_values)  return silhouette  silhouette = calculate_silhouette_score(X, labels) print(f'Silhouette Score: {silhouette}')  Silhouette Score: 0.5617085758870528</pre>
	A Silhouette Score of 0.56 is slightly closer to 1. This suggest that the clusters we got by using k=3 has done a good job in groupoing the data points. Most of the data in each cluster is of the same type.  Discuss the similarities and differences between 1) K-means, 2) Gaussian Mixture Models (GMM) and 3) the Gaussian naive Bayes classifier in terms of their assumptions and parameter estimation methods. In particular, write down the implications of each assumption (e.g., x and y are assumed to be independent, and therefore we have). You can choose to write your answers as Python code, pseudo code or mathematical equations.
	<ul> <li>K-means</li> <li>Assumptions and Implications:</li> <li>K-means assumes that clusters are spherical and equally sized with the centroid being the center of the sphere. This implies that clusters with non-spherical shapes can be poorly represented by centroids, also, if clusters have different sizes, K-means might not represent them accurately, preferring clusters with more points.</li> <li>Assumes that all clusters have equal variance (homodedasticity). If the scales of the variables are different, then variables with larger scales may dominate the clustering process, making the algorithm biased towards those features. This is why it's often recommended to standardize variables before applying K-means.</li> </ul>
	<ul> <li>Assigns each data point exclusively to one cluster (hard clustering), which can lead to misclassifications if the true underlying structure involves overlapping clusters.</li> <li>The number of clusters is assumed to be known in advance. In real world scenarios, the number of cluster is not known in advance. There is no one defined true number of cluster, it has to be determined from methods, data-based criteria and knowledge of the domain.</li> <li>The K-means algorithm implicitly assumes that the features within each cluster are independent. This assumption is reflected in the way the algorithm calculates distances between data points and cluster centroids. However, this assumption of feature independence is not always realistic in real-world data. Many datasets contain features that are correlated with each other.</li> </ul> Parameter Estimation:
	<ul> <li>Centroids: They are estimated by iteratively updating the mean of the data points assigned to each cluster. The algorithm starts with a random initialization of these centroids. Then it alternates between assigning data points to the nearest centroid and updating the centroid based on the current assignment of data points. Because the initial selection of centroids is random (it is sensitive to the initial selection of centroids), different runs of the K-means algorithm on the same data may produce different results. This is known as the problem of loop optima. In other words, the K-means algorithm may converge to different solutions depending on the initial selection of centroids.</li> <li>K-means uses distance-based metrics to identify similar points for clustering. This becomes a problem when we have numerous dimensions (curse of dimensionality). As the number of dimensions increases, the distances between different data points tend to be closer together.</li> <li>Labels: Assign each data point to the cluster with the nearest centroid, this process helps in forming the different clusters. Each cluster is a group of data points that are closer to one centroid than to any other.</li> </ul>
	Gaussian Mixture Models (GMM):  Assumptions and Implications:  Data points are generated from a mixture of of several Gaussian distributions with unknown parameters. This means each cluster is modeled as a Gaussian distribution, allowing for more flexibility compared to K-means.  GMM assumes that clusters can be elliptical and can have different sizes. This implies that GMM can capture more complex cluster structures and sizes, also overlapping clusters.  GMM model different covariance structures for each cluster, which is a key feature allowing GMM to adapt to various cluster shapes and orientations.
	<ul> <li>In GMM, a point can belong to different clusters with different levels of membership (probabilistic approach provides a nuanced view of the data structure, accommodating uncertainty in cluster membership.</li> <li>Parameter Estimation:</li> <li>Means, covariances, weights are estimated through the Expectation-Maximization (EM) algorithm. The algorithm iteratively updates the mean, covariance, and weight parameters to maximize the likelihood of the data. This iterative process involves alternating between assigning data points to clusters (expectation step) and updating the parameters based on these assignments (maximization step). This implies that GMM is generally more computationally expensive than K-means</li> <li>Gaussian Naive Bayes Classifier:</li> </ul>
	Assumptions and Implications:  • Features are assumed to be conditionally independent given the class label, which means that knowing the value of one feature doesn't change our belief about any other feature given the class. The independence assumption simplifies the computation and ignores potent interactions between features. However is often violated in real world scenarios.  • Features are assumed to be normally distributed (Gaussian), which is a strong assumption about the data structure. Gaussian distribution assumption may not hold for all features, which can negatively affect the model's performance if features have different distributions.  • The prior probabilities of classes are either uniform (assumed to be equal) or can be estimated from the training dataset.
	<ul> <li>The model tends to perform well even when this assumption is violated if the dataset is large enough or if the dependencies between features do not discriminate between classes.</li> <li>Parameter Estimation:</li> <li>Mean and variance for each feature in each class are estimated Maximum Likelihood Estimation.</li> <li>Part 2: Apply the K-means algorithm to compress images</li> </ul>
	<pre># Load multiple images def load_images(image_path):     """     Load multiple images from a list of file paths.     """     for path in image_path:         with Image.open(path) as image:         yield np.asarray(image)</pre>
	<pre>def save_images(images, titles, base_folder="images", sub_folder="compressed"):     """     Save images to a folder.     """     folder_path = os.path.join(base_folder, sub_folder)     os.makedirs(folder_path, exist_ok=True)     for i, image in enumerate(images):         path = os.path.join(folder_path, f'{titles[i]}.png')         Tmage_fromarray(image)_save(nath)</pre>
	<pre>Image.fromarray(image).save(path)  # Compress images utilizing our K-means implementation  def compress_images(images, K):     """      Compress images using K-means clustering.     """  # Initialize compressed images     compressed_images = []</pre>
	<pre>for image in images:     # Reshape the image to a 2D array     pixels = image.reshape(-1, 3)  # Run K-means clustering     centroids, labels = kmeans(pixels, K, n_init=10)  # Replace each pixel value with its corresponding centroid     compressed_image = centroids[labels].reshape(image.shape).astype(np.uint8)</pre>
	<pre>compressed_images.append(compressed_image) return compressed_images  # Plot original and compressed images side by side def plot_images(original_images, compressed_images, titles, K):</pre>
	<pre>for i, (original, compressed) in enumerate(zip(original_images, compressed_images)):     # Plotting original image     plt.subplot(len(original_images), 2, 2*i + 1)     plt.imshow(original)     plt.title(f'Original Image: {titles[i]}')     plt.axis('off')  # Plotting compressed image     plt.subplot(len(original_images), 2, 2*i + 2)</pre>
[14]:	<pre>plt.imshow(compressed)   plt.title(f'Compressed Image (K={K}): {titles[i]}')   plt.axis('off')  plt.tight_layout()   plt.show()  def compute_compression_ratio(original_images, compressed_images, K):   """Compute the compression ratio for each image"""</pre>
	ratios = []  for original, compressed in zip(original_images, compressed_images):  # Calculate the number of unique colors in the original image.  original_unique_colors = len(np.unique(original.reshape(-1, original.shape[2]), axis=0))  # Calculate the total number of pixels in the image  number_of_pixels = original.shape[0] * original.shape[1]  # Estimate the compressed image size
	<pre>compressed_size = K * 24 + number_of_pixels * log2(K)  # Estimate the original image size original_size = original_unique_colors * 24  # Calculate the compression ratio ratio = original_size / compressed_size ratios.append(ratio) return ratios</pre>
	<pre># Load images image_paths = ['images/original/img1.png', 'images/original/img2.png', 'images/original/img3.png'] original_images = list(load_images(image_paths)) compressed_images = compress_images(original_images, K=12)  # Plot images plot_images(original_images, compressed_images, ['Image 1', 'Image 2', 'Image 3'], K=12)</pre>
	# Save images save_images(compressed_images, ['Image1', 'Image2', 'Image3'])  Original Image: Image 1  Compressed Image (K=12): Image 1
	Original Image: Image 2  Compressed Image (K=12): Image 2
	Original Image: Image 3  Compressed Image (K=12): Image 3
	A Marian of Mari
	<pre># Calculate compression ratio ratios = compute_compression_ratio(original_images, compressed_images, K=12) print(f'Compression Ratios: {ratios}') # Calculate original image sizes</pre>
	<pre>original_image_sizes = [os.path.getsize(path) for path in image_paths] compressed_image_sizes = [os.path.getsize(f'images/compressed/{title}.png') for title in ['Image1', 'Image2', 'Image3']]  # Convert to Megabytes original_image_sizes_mb = [size / (1024 * 1024) for size in original_image_sizes] compressed_image_sizes_mb = [size / (1024 * 1024) for size in compressed_image_sizes] print(f'Sorted Original Image Sizes: {[f"{size:.2f} MB" for size in original_image_sizes_mb]}') print(f'Sorted Compressed Image Sizes: {[f"{size:.2f} MB" for size in compressed_image_sizes_mb]}')</pre>
	Compression Ratios: [0.4926535405512305, 0.27171666869431893, 0.5363781428581245] Sorted Original Image Sizes: ['2.65 MB', '1.68 MB', '2.45 MB'] Sorted Compressed Image Sizes: ['0.60 MB', '0.47 MB', '0.50 MB']  Image Compression Results  For the chosen value of K=12, the following observations were made:
	<ul> <li>Compression Ratios: The calculated compression ratios for the images were [0.49, 0.27, 0.54]. These ratios indicate the proportion of the original image size to the compressed image size meaning a significant reduction in size.</li> <li>File Sizes: The sizes of the original images were 2.65 MB, 1.68 MB, and 2.45 MB. After compression and saving as PNG files, the sizes were reduced to 0.60 MB, 0.47 MB, and 0.50 MB.</li> <li>Image Compression Discussion</li> <li>The compressed images are considerably smaller in size compared to the original images, which confirms the effectiveness of the K-means compression technique.</li> <li>The observed file size reduction is consistent with the calculated compression ratios, indicating a successful application of the algorithm.</li> </ul>
	<ul> <li>The choice of K=12 appears to be effective for this dataset, as it significantly reduces file size while presumably maintaining a reasonable level of detail in the images.</li> <li>Saving the compressed images as PNGs provided an accurate representation of the compression's impact on file size without additional file size that might have been introduced by a JPEG.</li> <li>This experiment demonstrates the potential of K-means for practical image compression and especially in scenarios where reducing storage space is crucial.</li> </ul> Part 3 Use AIC and BIC to choose K for Gaussian Mixture Models # Import libraries
	<pre>from sklearn.mixture import GaussianMixture from sklearn.datasets import load_breast_cancer  # Load dataset data = load_breast_cancer().data  def compute_gmm_aic_bic(data, max_k):     """</pre>
	Compute the AIC and BIC for a range of K values.  """  aic_values = [] bic_values = [] ks = range(1, max_k + 1)  for k in ks:  # Fit Gaussian Mixture Model gmm = GaussianMixture(n_components=k, random_state=0).fit(data)
	<pre>aic_values.append(gmm.aic(data)) bic_values.append(gmm.bic(data))  return aic_values, bic_values, ks  # Compute AIC and BIC for a range of K values aics, bics, ks = compute_gmm_aic_bic(data, max_k=15)</pre>
	<pre># Plot AIC and BIC plt.figure(figsize=(12, 5)) plt.plot(ks, aics, label='AIC') plt.plot(ks, bics, label='BIC') plt.xlabel('Number of Components (K)') plt.ylabel('Information Criterion') plt.title('AIC and BIC for Different Numbers of GMM Components') plt.legend() plt.show()</pre>
	AIC and BIC for Different Numbers of GMM Components  -15000 - AIC -20000 - BIC
	Up -2500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000350003500035000 -
	-40000 - -45000 -
	2 4 6 8 10 12 14  Number of Components (K)  Choose the best K on the breast_cancer data set using AIC and BIC and discuss your observations  The optimal number of components for the breast cancer dataset appears to be K=2, as indicated by the lowest AIC and BIC values at this point. This suggests that a model with two Gaussian components best captures the data's structure without unnecessary complexity. Bey
	K=2, the increasing AIC and BIC values imply that additional components are not justified, favoring a simpler model that sufficiently explains the data.
	References:  Silhouette (clustering) (2023). Available at: https://en.wikipedia.org/wiki/Silhouette_%28clustering%29 (Accessed: 10 December 2023).  Selecting the number of clusters with silhouette analysis on KMeans clustering (2023). Available at: https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html (Accessed: 10 December 2023).  Elbow method (clustering) (2023). Available at: https://en.wikipedia.org/wiki/Elbow_method_%28clustering%29 (Accessed: 10 December 2023).
	References:  • Silhouette (clustering) (2023). Available at: https://en.wikipedia.org/wiki/Silhouette_%28clustering%29 (Accessed: 10 December 2023).  • Selecting the number of clusters with silhouette analysis on KMeans clustering (2023). Available at: https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html (Accessed: 10 December 2023).