Northeastern University, Khoury College of Computer Science

CS 6220 Data Mining - Assignment 4

Name: Xinyue Han

Git Username: aiC0ld

Github repo link:
https://github.com/aiC0ld/CS6220-DataMining/tree/main

E-mail: han.xinyue@northeastern.edu

Parameter Estimation

It is well-known that light bulbs commonly go out according to a Poisson distribution, and are independent regardless of whether or not they're made in the same factory. An architect has outfitted a building with 32,000 of the same lightbulb.

Assuming the Poisson distribution has the form:

$$p(X|\lambda) = \frac{\exp^{-\lambda} \lambda^{x_i}}{x_i!} \tag{0.1}$$

1. derive the maximum likelihood estimate of the parameter λ.

$$p(x|\lambda) = \frac{exp^{-\lambda}\lambda^{x_i}}{x_i!}$$

Likelihood function : $L(\lambda) = \prod_{i=1}^{n} \frac{exp^{-\lambda}\lambda^{x_i}}{x_i!}$

Log-likelihood function : $\log L(\lambda) = \sum_{i=1}^{n} \log\left(\frac{exp^{-\lambda}\lambda^{x_i}}{x_i!}\right)$

$$= \sum_{i=1}^{n} \log(exp^{-\lambda}\lambda^{x_i}) - \log(x_i!)$$

$$= \sum_{i=1}^{n} \log(exp^{-\lambda}) + \log\lambda^{x_i} - \log(x_i!)$$

$$= \sum_{i=1}^{n} -\lambda + x_i \log\lambda - \log(x_i!)$$

$$= -n\lambda + \log\lambda \cdot \sum_{i=1}^{n} x_i - \log \sum_{i=1}^{n} x_i!$$

$$\log L(\lambda) = -n\lambda + \log\lambda \cdot \sum_{i=1}^{n} x_i - \log \sum_{i=1}^{n} x_i!$$

$$\frac{\partial L(\lambda)}{\partial \lambda} = -n + \frac{1}{\lambda}\sum_{i=1}^{n} x_i - 0 = 0$$

$$\lambda = \frac{\sum_{i=1}^{n} x_i}{n}$$

K-Means

The normalized automobile distributor timing speed and ignition coil gaps supplied are from production F-150 trucks over the years of 1996, 1999, 2006, 2015, and 2022. We have stripped

out the labels for the five years of data. Each sample in the dataset is two-dimensional, i.e. xi ∈ R^2, and there are N = 5000 instances in the data.

Vanilla k-Means

In this part of the homework, we'll take a look at how we can identify patterns in this data despite not having the labels. We'll start with the simplest approach, the k-Means unsupervised clustering algorithm.

2. Implement a simple k-means algorithm in Python on Colab with the following initialization:

$$\mathbf{x}_1 = \begin{pmatrix} 10 \\ 10 \end{pmatrix}, \mathbf{x}_2 = \begin{pmatrix} -10 \\ -10 \end{pmatrix}, \mathbf{x}_3 = \begin{pmatrix} 2 \\ 2 \end{pmatrix}, \mathbf{x}_4 = \begin{pmatrix} 3 \\ 3 \end{pmatrix}, \mathbf{x}_5 = \begin{pmatrix} -3 \\ -3 \end{pmatrix}, \tag{0.2}$$

You need only 100 iterations, maximum, and your algorithm should run very quickly to get the results.

2. Implement a simple k-means algorithm in Python on Colab with the following initialization:

```
def kmeans_cluster(data,
                   centroids = np.random.randn(2,5),
                   P = np.eye(2),
                   num_iterations = 100):
  classes = np.random.choice(5, len(data))

  # <YOUR-CODE-HERE>
  centroids = data.T[:, :5]
  centroids = centroids.T
  for i in range(num_iterations):
    for j in range(len(data)):
      min_distance = np.linalg.norm(data[j] - centroids[0])
      nearst_centroid_index = 0
      for idx in range(1, len(centroids)):
          distance = np.linalg.norm(data[j] - centroids[idx])
          if distance < min_distance:
              min_distance = distance
              nearst_centroid_index = idx
      classes[j] = nearst_centroid_index
    new_centroids = np.full_like(centroids, fill_value=0)
    for j in range(centroids.shape[0]):
        points_in_cluster = data[classes == j]
        if len(points_in_cluster) > 0:
            new_centroids[j] = np.mean(points_in_cluster, axis=0)
        else:
            new_centroids[j] = centroids[j]
    if np.array_equal(centroids, new_centroids):
      break
    centroids = new_centroids
  centroids = centroids.T

  return centroids, classes
```

3. Scatter the results in two dimensions with different clusters as different colors. You can use **matplotlib**'s **pyplot** functionality:

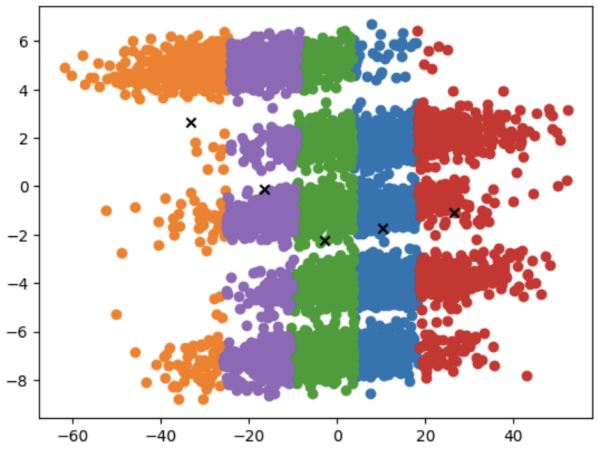
```
>> import matplotlib.pyplot as plt
>> plt.scatter(<YOUR CODE HERE>)
```

3. Scatter the results in two dimensions with different clusters as different colors.

```
[102] import matplotlib.pyplot as plt

      def plot_data(data, centroids, classes):
        #<YOUR-CODE-HERE>
        for i in range(len(centroids.T)):
            cluster_points = data[classes == i]
            plt.scatter(cluster_points[:, 0], cluster_points[:, 1])
        plt.scatter(centroids[0, :], centroids[1, :], c='black', marker='x')
        plt.show()
        return
```

```
use_P = True #@param
centroids_ = np.array([[10.0, 10.0], [-10.0, -10.0], [2.0, 2.0], [3.0, 3.0], [-3.0, -3.0]])

if not use_P:
  P_ = np.eye(2)
else:
  P_ = np.array([[10, 0.5], [-10, 0.25]])

centroids_, classes_ = kmeans_cluster( all_data, centroids_, P = P_ )
plot_data(all_data, centroids_, classes_)
```

use_P: True



4. You will notice that in the above, there are only five initialization clusters. Why is $k = 5$ a logical choice for this dataset? After plotting your resulting clusters and. What do you notice?

Q4 Answer: The dataset consists of normalized data representing the automotive distributor timing speed and ignition coil gap for F-150 trucks produced in five years: 1996, 1999, 2006, 2015, and 2022. Since there are five different years in the dataset, it's reasonable to assume that the data points for each year form natural groupings or clusters. Therefore, it is logical to

choose k = 5. We expect each cluster to roughly correspond to one of these five production years by clustering the data into five groups. After plotting the result clusters, I can see five distinct clusters. Each cluster corresponds to a different grouping of data points. This indicates that the K-Means algorithm has successfully partitioned the dataset into five clusters. I also noticed that the cluster plot is not perfectly spherical, but rather stretched horizontally. The cluster plot is stratified vertically, which suggests that one feature with a smaller range contributes less to the Euclidean distance calculation, while features with a larger range on the horizontal axis dominate the clusters. And using Euclidean distance emphasizes larger distances. We should ensure that both features contribute equally to the distance calculation, and normalizing the data may produce a more evenly shaped cluster plot.

With Production Information

Very often, it is possible to obtain additional information about the collected data. This sometimes allows us to define a new mathematical operators (including distances). In this part of the homework, we'll look at how to use this information to improve our modeling with an understanding of how two features in each sample are related.

A common distance metric is the *Mahalanobis Distance* with a specialized covariance.

$$d(\mathbf{x}, \mathbf{y}) = (\mathbf{x} - \mathbf{y})^T (P^T P)^{-1} (\mathbf{x} - \mathbf{y}) \tag{0.3}$$

where $\mathbf{x}$ and $\mathbf{y}$ are two points of dimensionality m (2 in this case), and $d(\mathbf{x}, \mathbf{y})$ is the distance between them. In the case of the F150 engine components, P is a known relationship through Ford's quality control analysis each year, where it is numerically shown as below:

$$P = \begin{pmatrix} 10 & 0.5 \\ -10 & 0.25 \end{pmatrix} \tag{0.4}$$

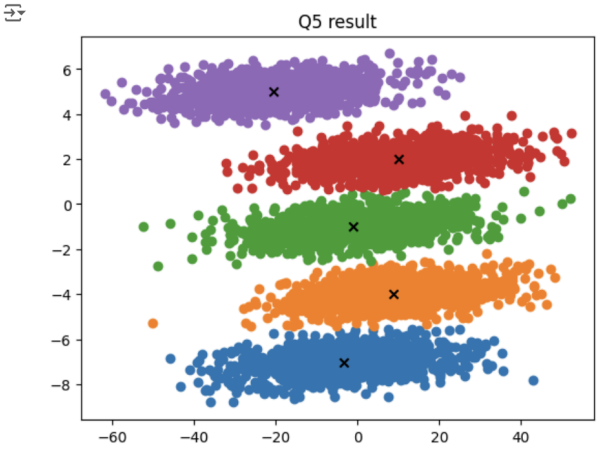
5. Implement a specialized k-means with the above Mahalanobis Distance. Scatter the results with the different clusters as different colors. What do you notice? You may want to pre-compute P^{-1} so that you aren't calculating an inverse every single loop of the the k-Means algorithm.

5. Implement a specialized k-means with the above Mahalanobis Distance. Scatter the results with the different clusters as different colors. What do you notice? You may want to precompute P−1 so that you aren't calculating an inverse every single loop of the the k-Means algorithm.

```python
def mahalanobis_distance(x, y, ptp):
    delta = x - y
    return np.sqrt(delta.T @ ptp @ delta)

def kmeans_mahalanobis(data,
                       centroids = np.random.randn(2,5),
                       P = np.eye(2),
                       num_iterations=100):
    centroids = data.T[:, :5]
    centroids = centroids.T
    classes = np.random.choice(5, len(data))
    ptp = np.linalg.inv(P.T @ P)
    for i in range(num_iterations):
      for j in range(len(data)):
        distances = []
        for centroid in centroids:
            distances.append(mahalanobis_distance(data[j], centroid, ptp))
        classes[j] = np.argmin(distances)
      new_centroids = []
      for i in range(len(centroids)):
          points_in_cluster = data[classes == i]
          if len(points_in_cluster) > 0:
              new_centroids.append(np.mean(points_in_cluster, axis=0))
          else:
              new_centroids.append(centroids[i])
      new_centroids = np.array(new_centroids)
      if np.array_equal(centroids, new_centroids):
        break
      centroids = new_centroids
    return centroids, classes
```

```python
def plot_clusters(data, centroids, classes):
    for i in range(len(centroids)):
      cluster_points = data[classes == i]
      plt.scatter(cluster_points[:, 0], cluster_points[:, 1])
    for i, centroid in enumerate(centroids):
      plt.scatter(centroid[0], centroid[1], c='black', marker='x')
    # plt.scatter(centroids[0, :], centroids[1, :], c='black', marker='x')
    plt.title("Q5 result")
    plt.show()
    return

P = np.array([[10, 0.5], [-10, 0.25]])
use_P = True #@param
initial_centroids = np.array([[10, 10], [-10, -10], [2, 2], [3, 3], [-3, -3]])
if not use_P:
  P_ = np.eye(2)
else:
  P_ = np.array([[10, 0.5], [-10, 0.25]])

final_centroids, final_classes = kmeans_mahalanobis(all_data, initial_centroids, P)

plot_clusters(all_data, final_centroids, final_classes)
```

use_P: True



I noticed five distinct clusters with clear vertical separation and significant horizontal expansion. The centroids are located in the center of their respective clusters, indicating that the specialized K-Means algorithm using the Mahalanobis distance has successfully grouped the data points. The vertical separation indicates that one feature plays a dominant role in distinguishing the clusters. Overall, the clustering results are well-defined, with each cluster clearly separated from the others.

6. Calculate and print out the *first* principle component of the aggregate data.

Q6 Answer: The first principle component of the aggregate data is `[[ 0.99838317 -0.05684225]]`

6. Calculate and print out the first principle component of the aggregate data.

```
[96] def get_first_principle_component(data, n_components = 1):
       cov_matrix = np.cov(data, rowvar=False)
       # Eigenvalue decomposition
       eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
       first_principal_component = eigenvectors[:, np.argmax(eigenvalues)]
       return first_principal_component

     first_principle_component = get_first_principle_component(all_data)
     print("Q6 Answer: The first principle component of the aggregate data:", first_principle_component)
```

```
Q6 Answer: The first principle component of the aggregate data: [[ 0.99838317 -0.05684225]]
```

7. Calculate and print out the *first* principle components of *each cluster*. Are they the same as the aggregate data? Are they the same as each other?

```
Q7 Answer
The first principle component of cluster 1 : [[0.99992533 0.01222027]]
The first principle component of cluster 2 : [[0.99989374 0.01457781]]
The first principle component of cluster 3 : [[0.99990986 0.01342629]]
The first principle component of cluster 4 : [[0.99993306 0.01157047]]
The first principle component of cluster 5 : [[0.99993527 0.01137789]]
```

They are not the same as the aggregate data and not the same as each other.

7. Calculate and print out the first principle components of each cluster. Are they the same as the aggregate data? Are they the same as each other?

```
first_principal_components_by_cluster = []
unique_clusters = np.unique(final_classes)
print("Q7 Answer")
for cluster in unique_clusters:
    data_in_cluster = all_data[final_classes == cluster]
    principal_component = get_first_principle_component(data_in_cluster)
    first_principal_components_by_cluster.append(principal_component)
    print("The first principle component of cluster", cluster + 1, ":", principal_component)
```

```
Q7 Answer
The first principle component of cluster 1 : [[0.99992533 0.01222027]]
The first principle component of cluster 2 : [[0.99989374 0.01457781]]
The first principle component of cluster 3 : [[0.99990986 0.01342629]]
The first principle component of cluster 4 : [[0.99993306 0.01157047]]
The first principle component of cluster 5 : [[0.99993527 0.01137789]]
```