Homework 9: K-Means and GMM Clustering

Due: 21 Nov, 2025 11:59pm ET

The first part of this homework asks you to fill in portions of classes that you can then use to perform k-means analysis. The second part asks you to perform Gaussian Mixture Model (GMM) clustering on a given dataset.

Goals

In this assignment you will:

- Get familiar with using objects and classes by defining some methods and using objects to perform a computation
- Implement k-means
- Get practice with implementing Gaussian Mixture Models (GMMs) using sklearn

Background

Classes and Objects

Please see the slide on objects and classes on Brightspace.

k-means and GMMs

Please see the slides on clustering and specifically k-means and GMMs on Brightspace.

Instructions

0) Set up your repository for this homework.

The repository should contain the following files:

1. This README.
2. `cluster.py` which contains the definition of the `Cluster` class and some testing code.
3. `point.py` which contains the definition of the `Point` class and some testing code.
4. `kmeans.py` which contains the skeleton of the k-means algorithm and some testing code.
5. `gmm.py` which contains the skeleton of the `gaus_mixture()` function.
6. `gmm_data_x1.csv` and `gmm_data_x2.csv` which contain the data that will be used to test your GMM function.

1) Homework Problem 1: Implementing K-Means Clustering

This homework problem comprises of three steps which work together to implement k-means on the given dataset. You are required to complete the specified methods in each class which would be used for k-means clustering in step 3.

Step 1: Complete Point class

Complete the missing portions of the `Point` class, defined in `point.py`:

1. `distFrom`, which calculates the (Euclidean) distance between the current point and the target point. Be sure to account for the fact that a point may be in more than two dimensions (Euclidean distance generalizes: square the difference in each dimension and take the square root of the sum). In general, for points given n-dimensional Euclidean space, the Euclidean distance d between two n dimensional vectors, say p and q is given by:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_i - q_i)^2 + \cdots + (p_n - q_n)^2}.$$

   It is completely fine to use `math.sqrt()` to calculate the square root.
2. `makePointList`, which takes in a data p-by-d input matrix `data` and returns a list of p `Point` objects. Hint: Instantiate a point object for every row in the input, `data`. Note that `makePointList` is outside the `Point` class.

If you test your code by running `python3 point.py`, you should get the following:

```
Point list: [Point([0.5 2.5]), Point([0.3 4.5]), Point([-0.5  3. ]), Point([0.
Distance between first two points: 2.009975124224178
```

(Your floating point numbers may be a little off due to rounding, we will check upto 4 decimal places.)

Step 2: Complete Cluster class

Complete the missing portions of the `Cluster` class, defined in `cluster.py`:

1. `avgDistance`, which computes the average distance from the center of the cluster (stored in `self.center`) to all of the points currently in the cluster (stored in `self.points`). This can most easily be done by summing the distances between each point and the current center and then dividing the sum by the total number of points. Function returns the final computed value of average distance. (Note that `self.center` is a Point object and `self.points` is a set of Point objects).
2. `updateCenter`, which updates the center of the cluster (stored in `self.center`) to be the average position of all the points in the cluster. Function returns the coordinates of the center.

Note that if there are no points in the cluster (i.e., `self.points` is an empty set), then self.center should be unchanged

> Note: We have defined `dim` and `coords` as properties that return information about the center of the cluster -- this means that if you pass a cluster into a method that is expecting a point, operations that access `dim` and `coords` will use the center of the cluster. Think about how that might be useful in conjunction with the `closest` method defined for `Point`.

If you test your code by running `python3 cluster.py`, you should get the following:

```
Cluster: 0 points and center = [2.0, 2.0]
Updated Cluster: 3 points and center = [2.0, 2.0]
Average distance: 2.2126813451167684
Updated Cluster: 3 points and center = [2.1666666666666665, 4.0]
Updated average distance: 1.0572107851777524
```

(Your floating point numbers may be a little off due to rounding, we will check upto 4 decimal places.)

Step 3: Implement k-means

Use the methods in `Point` and `Cluster` to implement the missing `kmeans` method in `kmeans.py`. The basic recommended procedure is outlined in `kmeans.py`.

If you test your code by running `python3 kmeans.py`, you should get the following clusters:

```
Cluster: 3 points and center = [0.2 4. ]
   [0.5 2.8]
   [1.2 5.3]
   [-1.1  3.9]
Cluster: 7 points and center = [10.05714286 -4.85714286]
   [10.3 -4.8]
   [ 8.9 -3.6]
   [12.1 -7.1]
   [ 7.4 -2.5]
   [11.5 -6.2]
   [10.8 -5.5]
   [ 9.4 -4.3]
```

> Note: **The order of the points in each cluster doesn't matter (neither does the order of clusters)**, all that's important is that each cluster contains the correct points. In addition, for this part **you are not permitted** to use the k-means implementation in the sklearn library.

(Your floating point numbers may be a little off due to rounding, we will check upto 4 decimal places.)

Homework Problem 2: Find best number of clusters to use on GMM algorithms

> Note: This problem is independent of the three problems above. In addition, **you are permitted** to use the GMM implementation in the sklearn library.

In this homework problem, you will employ GMM to cluster a data set and identify the right number of clusters in the data. The data points consist of two features, each stored in one of the two data files provided: 'gmm_data_x1.csv' and 'gmm_data_x2.csv.' As the first step, you will write the function `concatenate_features()` to concatenate these features into a single data structure. Notice that the order of data points is the same in the two files, i.e., the i-th value in 'gmm_data_x1.csv' and the i-th value in 'gmm_data_x2.csv' represent the first and second feature of the i-th data point. Then, you will write the function `gaus_mixture()` to use these concatenated features to run GMM clustering and obtain the optimal number of clusters given a list of candidate cluster numbers.

`concatenate_features()` takes in 2, 1-D numpy array of same length, say n. Each array represents one of the features of the n datapoints. The function returns an unified numpy array of the shape `(n, 2)` by concatenating the two numpy array arguments given to the function, where each column in the unified array represents the 1-D numpy arrays provided as input.

`gaus_mixture()` takes in a `(n, 2)` and a list of positive integers (possible number of clusters for the data), and find the optimal number of clusters from the given list for the gaussian mixture model clustering.

Given the concatenated features and a list of the number of clusters as input, the function should return the best number of clusters to use (from the input list of candidate cluster numbers) on the GMM.

The best number of clusters is determined by (1) fitting a GMM model using a specific number of clusters, (2) calculating its corresponding Bayes Information criterion (BIC - see formula below), and then (3) setting the number of clusters corresponding to the lowest BIC as the best number of clusters to use.

This function should be completed using the algorithm outlined in the skeleton code. In addition, consider the following hints:

1. The GMM algorithm can be implemented using the sklearn library using `gm = GaussianMixture(n_components=d, random_state=0).fit(data)`, where `d` corresponds to the number of clusters to use and `data` is an (for our case) `n-by-2` array of data. Lastly, `random_state=0` is a random seed that allows for reproducibility. **In your code, you must set** `random_state=0` when you call `GaussianMixture`.

2. The BIC formula is given by `BIC = -2log(L) + log(N)d`, where `L` is the maximum likelihood of the model, `d` is the number of parameters, and `N` is the number of data samples. When using the sklearn library, however, the BIC is given by `bic = gm.bic(data)`, where `gm` is the object returned when `GaussianMixture()` from Hint 1 is instantiated. `gm.bic` is how the BIC should be calculated in your implementation.

Here we have set the possible number of clusters as, `n_components=[3, 4, 5, 6, 7, 8, 9]`. If you test your code by running `python3 gmm.py`, you should get the below output:

```
Shape of the first feature array:  (400,)
Shape of the second feature array:  (400,)
Shape of the concatenated array:  (400, 2)
Best fit is when k = 4 clusters are used
```

> Note: You can ignore the UserWarning related to KMeans memory leak on Windows.

What you need to submit

Upload your completed versions of `kmeans.py`, `cluster.py`, `point.py`, and `gmm.py` to Gradescope. Do NOT change the names of the submissions files, this will cause autograder to fail and an automatic zero will be assigned.