INF367A Exercise 8

Naphat Amundsen

March 9, 2020

Introduction

This exercise is about using the Expectation-Maximization algorithm on Gaussian Mixture Models.

1 EM algorithm for GMMs

In this exercise, we try to find clusters of large cities. Download the data set largest_cities.csv which contains the coordinates (longitude and latitude) of the 500 largest cities in the world.

1. Implement the EM algorithm for GMMs using the formulas in the lecture slides
2. Compute log-likelihood $\log P(x|\theta) = \sum_{i=1}^{n} \log \left[\sum_{k=1}^{K} \pi_k N(x_i|\mu_k, |\Sigma_k) \right]$ after each M-step. Sanity check: log-likelihood should increase after each step
3. Use BIC to choose the number of components K. Note that representing one d-dimensional component requires $d + (d+1) \times d/2$ parameters (d parameters for the mean vector and $(d+1) \times d/2$ parameters for the covariance matrix). In addition, we need $K - 1$ parameters for mixing coefficients (-1 is there because mixing coefficients sum to one). Thus, the total number of parameters is $K \times (d + (d+1)d/2) + (K-1)$.

Hints: The EM algorithm is sensitive to initialization. Therefore, you may want to try several different initializations. You may encounter

```
ValueError: array must not contain infs or NaNs
```

because some responsibilities are `nan`. This is typically due to some data points lying so far away from all the cluster centers that Python interprets all probabilities as 0. In this case, you should try to initialize covariance matrices with larger variances.

1.1 Implementing the EM-algorithm for a GMM

The EM algorithm consists of two steps: Expectation and Maximization. Where the Expectation step (E-step) calculates the probability π_{ij} of a data point $\boldsymbol{x}_i$ coming from each component C_j, i.e

$$\boxed{\pi_{ij} = P(C_j|\boldsymbol{x}_i)} \tag{1}$$

The Maximization step (M-step) then assumes that each π_{ij} (also called weights) are correct, and then updates the parameters for the components which are modelled by the Multivariate Gaussians $\mathcal{N}_c(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c, \Phi_c)$. The formulas used to update the parameters are essentially the "weighted" versions of the MLEs for the case of known labels:

$$\boxed{\hat{\boldsymbol{\mu}_j} = \frac{\sum_{i=1}^{N} w_{ij}\boldsymbol{x}_i}{\sum_{i=1}^{N} w_{ij}}} \tag{2}$$

$$\boxed{\hat{\boldsymbol{\Sigma}_j} = \frac{\sum_{i=1}^{N} w_{ij}(\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{x}_i - \boldsymbol{\mu})^T}{\sum_{i=1}^{N} w_{ij}}} \tag{3}$$

Where the mixing probabilities are

$$\boxed{\hat{\Phi_j} = \frac{\sum_{i=1}^{N} w_{ij}}{N}} \tag{4}$$

It should also be noted that one initializes the algorithm with arbitrary values for the parameters.

1.2 Design

Parameters

- N = Number of data points
- k = Number of components / predicted clusters

In the initialization phase, the program creates array $\mathbf{W}$ of size $(N \times k)$ as a container for each data-weight π_{ij}. Each column in the array is respective to each component. Each component are defined by their respective sets of variables, which are also stored in arrays.

E-step

1. Calculate the data weights for each component using (1) and store them in $\mathbf{W}$
2. Divide each row of $\mathbf{W}$ by the sum of the same row of itself.

M-step

1. For each component do:

   (a) Update the mean using (2)

   (b) Update the covariance matrix using (3)

   (c) Update the mixing probability using (4)

The E and M steps are repeated until either a maximum number of iterations are reached or the likelihood changes with a sufficiently low delta between iterations.

Table 1: Using the EM-Algorithm with number of components between 2 and 18 with yielded the following table. To accommodate the sensitivity to starting parameters, the EM algorithm is ran six times, of which gives us multiple columns for BICs. Bolded elements indicates best performances.

models	BIC_1	BIC_2	BIC_3	BIC_4	BIC_5	BIC_6
GMM(k=2)	-5000.13	-4952.60	-5000.13	-4952.60	-5000.13	-5000.13
GMM(k=3)	-4863.11	-4859.00	-4948.97	-4859.00	-4920.91	-4948.97
GMM(k=4)	-4864.77	-4820.61	-4820.61	-4820.61	-4809.86	-4809.86
GMM(k=5)	-4816.59	-4806.08	-4780.18	-4794.38	-4794.38	-4814.87
GMM(k=6)	-4799.90	-4779.96	-4795.46	-4768.47	-4801.31	-4786.50
GMM(k=7)	-4772.40	-4750.28	-4765.29	-4791.78	-4773.76	-4797.49
GMM(k=8)	-4790.45	-4761.65	-4766.06	-4774.73	-4767.99	-4806.17
GMM(k=9)	-4768.56	-4781.17	-4758.29	-4810.91	-4793.60	-4746.56
GMM(k=10)	-4753.43	**-4734.07**	-4798.40	-4776.24	-4796.92	-4778.69
GMM(k=11)	-4770.90	-4763.49	-4762.60	-4737.75	-4777.17	-4744.58
GMM(k=12)	-4781.50	-4765.72	-4761.48	-4743.37	-4784.90	-4796.56
GMM(k=13)	-4742.97	-4765.77	-4748.39	-4771.71	-4759.71	-4768.28
GMM(k=14)	-4769.40	-4755.35	-4757.60	-4748.50	-4743.99	-4781.08
GMM(k=15)	-4790.03	-4763.56	-4794.54	-4786.38	-4764.57	-4759.05
GMM(k=16)	-4797.05	-4785.86	-4773.59	-4774.39	-4806.46	-4786.34
GMM(k=17)	-4776.43	-4797.44	-4784.10	-4780.90	-4821.30	-4806.14
GMM(k=18)	-4803.84	-4789.69	-4771.69	-4806.42	-4790.30	-4784.86

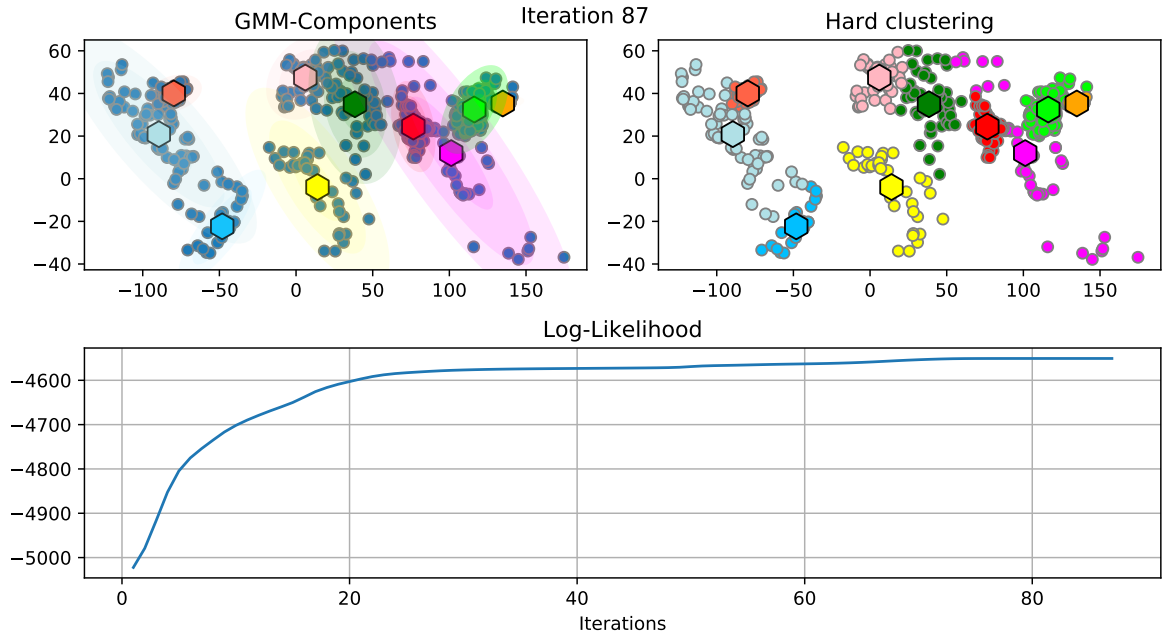

Figure 1: The output and log likelihood history of the best model. The upper left subplot visualizes the components. The opacity of the ellipses are linked to the mixing values.

2 Extension of the simple example

Suppose that we have n independent observations $x = (x_1, \ldots, x_n)$ from a two-component mixture of univariate Gaussian distribution with unknown mixing coefficients and unknown mean of the second component: $P(x_i|\mu, p) = (1-p) \cdot N(x_i|0, 1) + p \cdot N(x_i|\mu, 1)$.

- Write down the complete data log-likelihood and derive the EM-algorithm for learning maximum likelihood estimates for μ and p.
- Implement the EM-algorithm. Load data ex8_2.csv and learn the maximum likelihood estimates for $\hat{\mu}$ and $\hat{p}$. Plot the distribution $P(x_i|\hat{\mu}, \hat{p})$. Does it match the observed data? Plot the log-likelihood of the observed data for each iteration (This should increase after each iteration) Hint: You can use simple_example.pdf as a starting point.

Ultimate goal is to maximize $\hat{\theta} = \arg\max_\theta \log P(x|\theta)$. Let the variables $z = z_1, \ldots, z_i$ denote the actual component responsible for generating observation x_i. In detail in the context of this task:

$$z_i = (z_{i1}, z_{i2})^T = \begin{cases} (1,0)^T, & x_i \text{ is from } N(x_i|0,1) \\ (0,1)^T, & x_i \text{ is from } N(x_i|\mu,1) \end{cases}$$

where we define the distributions for z as follows:

$$P(z_{i1} = 1) = (1 - p)$$
$$P(z_{i2} = 1) = p$$

and

$$z_i = (z_{i1}, z_{i2})^T = \begin{cases} (1-p) \cdot N(x_i|0,1), & \text{if } z_{i1} = 1 \\ p \cdot N(x_i|\mu,1), & \text{if } z_{i2} = 1 \end{cases}$$

Then to obtain the probability of x_i with respect to $\theta = (p, \mu)$, we can simply marginalize away z:

$$P(x_i|\theta) = \sum_{i=1}^{n} P(z_i)P(x_i|z_i, \theta)$$

Then we can obtain the likelihood of the complete data (x, z) (see lecture notes for definition):

$$\log P(x, z|\theta) = \log \left[\prod_{i=1}^{N} P(x_i, z_i|\theta) \right] = \sum_{i=1}^{n} \log P(x_i, z_i|\theta) \tag{5}$$
$$= \sum_{i=1}^{n} \log \left[P(z_i) N(x_i|0,1)^{z_{i1}} N(x_i|\mu,1)^{z_{i2}} \right] \tag{6}$$
$$= \sum_{i=1}^{n} \log P(z_i) + z_{i1} \log N(x_i|0,1) + z_{i2} \log N(x_i|\mu,1) \tag{7}$$
$$= \sum_{i=1}^{n} \log(z_{i1}(1-p) + z_{i2}p) + z_{i1} \log N(x_i|0,1) + z_{i2} \log N(x_i|\mu,1) \tag{8}$$

E-step

First want to get expression for posterior distribution of latent variables given an estimate θ_t of θ:

$$P(z_{i1} = 1|x_i, \hat{\theta}_t) \propto P(z_{i1} = 1) \overbrace{P(\theta_t|z_{i1})}^{=1} P(x_i|\theta_t, z_{i1}) \tag{9}$$
$$= P(z_{i1} = 1) P(x_i|\theta_t, z_{i1}) \tag{10}$$
$$= (1-p) \cdot N(x_i|0,1) \tag{11}$$

$$P(z_{i2} = 1|x_i, \hat{\theta}_t) \propto P(z_{i2} = 1) \overbrace{P(\theta_t|z_{i2})}^{=1} P(x_i|\theta_t, z_{i2}) \tag{12}$$
$$= P(z_{i2} = 1) P(x_i|\theta_t, z_{i2}) \tag{13}$$
$$= p \cdot N(x_i|\mu,1) \tag{14}$$

Then by normalizing the values, we get the responsibilities / mixing probabilities:

$$\gamma(z_{i1}) = \frac{(1-p) \cdot N(x_i|0,1)}{(1-p) \cdot N(x_i|0,1) + p \cdot N(x_i|\mu,1)} \tag{15}$$

$$\gamma(z_{i2}) = \frac{p \cdot N(x_i|\mu,1)}{(1-p) \cdot N(x_i|0,1) + p \cdot N(x_i|\mu,1)} \tag{16}$$

Note that $\gamma(z_{i1}) + \gamma(z_{i2}) = 1$

The problem now is that z_i is not observed. The solution is to maximize

$$Q(\theta, \theta_t) = E_{z|x_i,\theta_t}\left[\log P(x_i, z|\theta)\right] \tag{17}$$

$$= \sum_{i=1}^{n} E[z_{i1}] \log N(x_i|0,1) + E[z_{i2}] \log N(x_i|\mu,1) \tag{18}$$

$$= \sum_{i=1}^{n} \gamma(z_{i1}) \log N(x_i|0,1) + \gamma(z_{i2}) \log N(x_i|\mu,1) \tag{19}$$

where $P(z|x,\theta_t)$ is the posterior distribution of the latent variables computed using the estimate θ_t

M-step

Maximize $Q(\theta, \theta_t)$ with respect to $\theta = (p, \mu)$. We maximize by setting derivative to zero. Note that

$$\frac{\mathrm{d}}{\mathrm{d}\mu} N(x_i|\mu,1) = N(x_i|\mu,1)(x_i - \mu)$$

Simply differentiate with respect to each variable separately, lets start with respect to μ

$$\frac{\mathrm{d}}{\mathrm{d}\mu} Q(\theta, \theta_t) = \frac{\mathrm{d}}{\mathrm{d}p} \sum_{i=1}^{n} (1 - \gamma(z_{i2})) \log N(x_i|0,1) + \gamma(z_{i2}) \log N(x_i|\mu,1) \tag{20}$$

$$= \sum_{i=1}^{n} \frac{\gamma(z_{i2})}{N(x_i|\mu,1)} N(x_i|\mu,1)(x_i - \mu) \tag{21}$$

$$= \sum_{i=1}^{n} \gamma(z_{i2})(x_i - \mu) = \sum_{i=1}^{n} \gamma(z_{i2})x_i - \gamma(z_{i2})\mu = 0 \tag{22}$$

$$\Rightarrow -\sum_{i=1}^{n} \gamma(z_{i2})\mu = -\sum_{i=1}^{n} \gamma(z_{i2})x_i \tag{23}$$

$$\Rightarrow \mu = \frac{\sum_{i=1}^{n} \gamma(z_{i2})x_i}{\sum_{i=1}^{n} \gamma(z_{i2})} \tag{24}$$

Then for p

$$\frac{\mathrm{d}}{\mathrm{d}p}Q(\theta,\theta_t) = \frac{\mathrm{d}}{\mathrm{d}p}\sum_{i=1}^{n}(1-\gamma(z_{i2}))\log N(x_i|0,1) + \gamma(z_{i2})\log N(x_i|\mu,1)$$

Recall that we actually have the constraint that $(1-p)+p=1$. Since we have a constrained optimization problem, we use good ol' Lagrange optimization:

$$\frac{\mathrm{d}}{\mathrm{d}p}Q(\theta,\theta_t) = \lambda\frac{\mathrm{d}}{\mathrm{d}p}\left[(1-p)+p-1\right] = \lambda\frac{\mathrm{d}}{\mathrm{d}p}0 = 0 \tag{25}$$

$$\frac{\mathrm{d}}{\mathrm{d}p}Q(\theta,\theta_t) = 0 \tag{26}$$

Wolfram alpha says that

$$\frac{\mathrm{d}}{\mathrm{d}p}\gamma(z_{i2}) = \frac{N(x_i|0,1)\cdot N(x_i|\mu,1)}{(N(x_i|0,1)(1-p)-N(x_i|\mu,1)p)^2}$$

Somehow from that we get (it's late, I am hungry now and want to just submit this)

$$\underline{\underline{p = \frac{1}{n}\sum_{i=1}^{n}\gamma(z_{i2})}} \tag{27}$$

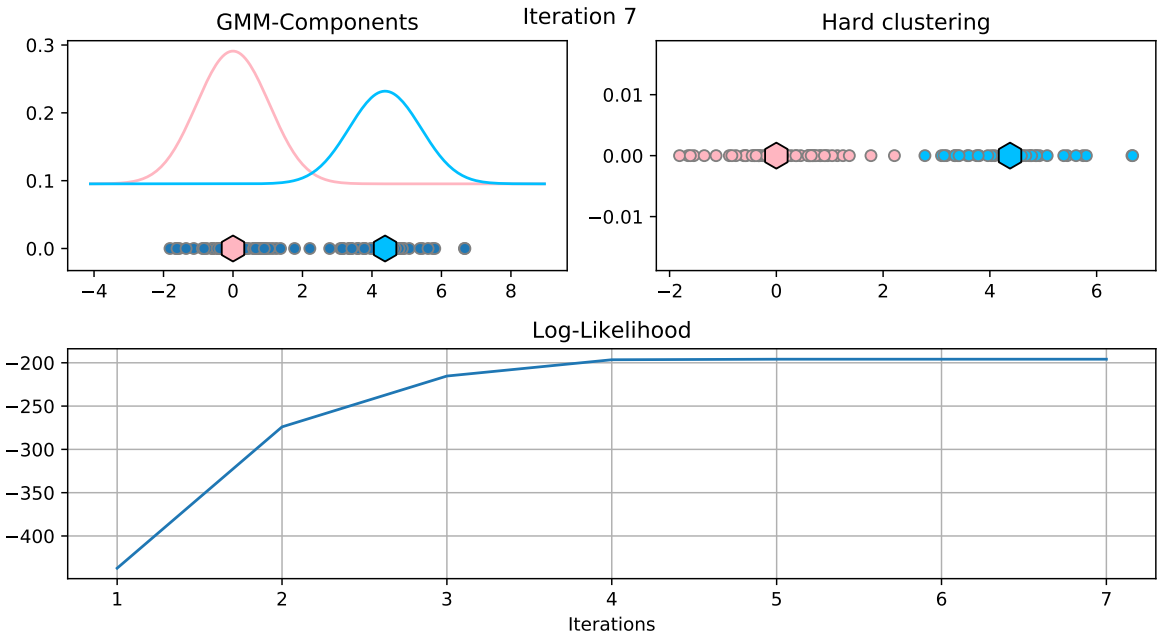

Figure 2: The output and log likelihood history of the best model. The upper left subplot visualizes the components. The opacity of the ellipses are linked to the mixing values.

The model seems to match the data well using the alternative EM algorithm.

A GMM Class

```
'''
Written by Naphat Amundsen
04/03/2020
'''
import numpy as np
from matplotlib import pyplot as plt
from scipy.stats import multivariate_normal as mvn
from scipy.stats import norm
from typing import Union
from matplotlib.patches import Ellipse

class GMM:
    '''General use GMM algorithm with built in viz tools'''
    def __init__(self, k: int = 3, init_covariance: Union[float, str]='
                                            auto') -> None:
        '''
        k: number of centroids / components

        init_covarince: initial value for diagonal elements in covariance
                                                matrix. If
                        'auto' is given as argument (default), the value
                                                            will be the
                                                            overall
                        variance of the data
        '''
        self.k = k
        self.init_covariance = init_covariance
        self._plot_flag=False

    def _init_dtype(self, dim) -> None:
        return np.dtype([
            ('mean', float, dim),
            ('cov', float, (dim,dim)),
            ('mix', float)
        ])

    def __repr__(self):
        return f'GMM(k={self.k})'

    def __str__(self):
        return f'GMM(k={self.k})'

    def _prepare_before_fit(self, X: np.ndarray) -> None:
        '''
        Prepares object attributes and such before fitting
        '''
        self.X = X
        self.N, self.dim = X.shape
        self.X_std = np.std(X)

        if self.init_covariance == 'auto':
```

```
            self.init_covariance = np.var(X)

        # Initialize component placeholders
        self.components = np.empty(self.k, dtype=self._init_dtype(self.dim
                                                                  ))

        # Pick random points as initial mean positions
        # Reshape to handle case for 1-dim data
        self.components['mean'] = \
            X[np.random.choice(range(self.N), self.k, replace=False)].
                                                        reshape(*self.components[
                                                        'mean'].shape)
        # Initialize covariance matrices with scaled identity matrices
        self.components['cov'] = np.repeat(self.init_covariance*np.eye(
                                                    self.dim)[np.newaxis,...],
                                                    self.k, axis=0)
        # Initialize uniform mixing weights
        self.components['mix'] = np.full(self.k, 1/self.k)

        # Weight for each data point, columns are respective to components
        self.weights = np.empty((self.N, self.k))

        self.hood_history = []
        # Calculate starting weights in order to calculate initial
                                                    likelihood
        self._E_step() # This automatically logs likelihood

        #EM iterations
        self.em_iterations = 0

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        ''''Returns log likelihoods for each data point'''
        hood = np.zeros(X.shape[0])
        for component in self.components:
            hood += component['mix'] * mvn.pdf(x=X, mean=component['mean']
                                                    , cov=component['cov'])
        return np.log(hood)

    def score_samples(self, X: np.ndarray) -> np.ndarray:
        '''Returns log likelihoods for each data point'''
        return self.predict_proba(X)

    def _E_step(self) -> None:
        '''
        E-step: Calculates the weights for each datapoint with
        respect to each component while assuming model parameters are
                                                    correct
        '''
        # C for component
        for i, c in enumerate(self.components):
            self.weights[:,i] = c['mix'] * mvn.pdf(x=self.X, mean=c['mean'
                                                    ], cov=c['cov'])
        w_axis_sum = self.weights.sum(axis=1)
```

```
        # Log the log-likelihood
        self.hood_history.append(np.log(w_axis_sum).sum())

        # Row-wise division
        self.weights /= w_axis_sum.reshape(-1,1)

    def _M_step(self) -> None:
        '''
        M-step: Updates the mean, covariance and priors of the
        components.
        '''
        # C for component
        for i, c in enumerate(self.components):
            # Vector of weights for component
            w = self.weights[:,i]
            w_sum = w.sum()

            # Update component mean
            c['mean'] = w@self.X/w_sum
            # Update component covariance
            D = self.X - c['mean']
            c['cov'] = D.T@(D*w.reshape(-1,1))/w_sum
            # Update component mixing probability
            c['mix'] = w_sum/self.N

    def get_labels(self) -> np.ndarray:
        '''Returns hard labels'''
        return np.argmax(self.weights, axis=1)

    def _EM_iterate(self) -> None:
        '''Do one EM iteration and save log-likelihood'''
        # _prepare_before_fit method should have been invoked once
        # before using this method. _prepare_before_fit will call
        # the E_step method once
        self._M_step()
        self._E_step()
        self.em_iterations += 1

    def fit(self, X, maxiter: int=420, rtol: float=1e-8, atol: float=1e-3)
                                                -> None:
        '''
        Fit the thing
        '''
        self._prepare_before_fit(X)

        for i in range(maxiter):
            self._EM_iterate()
            if np.allclose(self.hood_history[-1], self.hood_history[-2],
                                                  rtol=rtol, atol=atol):
                break
        return self
```

```
    def fit_animate(self, X, maxiter: int=420, rtol: float=1e-8, atol:
                                            float=1e-3,
                    figsize:tuple = (12,6), axis: list=[0,1]) -> None:
        '''
        Fit while visualizing
        '''
        from matplotlib import animation as anime
        self._prepare_before_fit(X)
        self._init_plot(figsize)

        ALL = np.array([self.left, self.right, self.lower])
        def animate(i):
            [ax.clear() for ax in ALL]
            self._EM_iterate()
            if np.allclose(self.hood_history[-1], self.hood_history[-2],
                                                  rtol=rtol, atol=atol):
                movie.event_source.stop()
                print('Converged')
            self.plot_result(axis=axis, show=False)

        movie = anime.FuncAnimation(self.fig, animate, frames=maxiter,
                                            interval=16, blit=False,
                                            repeat=False)
        plt.show()
        self._plot_flag = False
        return self

    @staticmethod
    def _draw_ellipse(position, covariance, ax, **kwargs):
        '''
        Source:
        https://jakevdp.github.io/PythonDataScienceHandbook/05.12-gaussian
                                            -mixtures.html

        Draw an ellipse with a given position and covariance

        Expects 2D covariance
        '''
        # Convert covariance to principal axes
        U, S, Vt = np.linalg.svd(covariance)
        angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
        width, height = 2 * np.sqrt(S)

        # Draw the Ellipse
        # Multiple draws for one covariance to express contours
        # print(1/np.linalg.norm(S))
        for nsig in range(1, 4):
            ax.add_patch(Ellipse(
                xy=position,
                width=nsig * width,
                height=nsig * height,
                angle=angle,
                **kwargs)
```

```
        )

    def _init_plot(self, figsize) -> tuple:
        '''Initialize plot attributes'''
        self._plot_flag = True
        # self.colors = ['tomato', 'orange', 'deepskyblue', 'yellow', 'red
                                                ', 'blue']
        self.colors = [
            'lightpink','deepskyblue','orange', 'lime',
            'magenta','yellow','green','red','powderblue',
            'tomato', 'orange', 'deepskyblue', 'yellow', 'blue'
        ]
        self.nc = len(self.colors)
        self.centroid_colors = [self.colors[i%self.nc] for i in range(len(
                                                self.components))]
        self.centroid_kwargs = dict(marker='h', s=200, c=self.
                                                centroid_colors, edgecolor='k
                                                ')
        self.X_kwargs = dict(edgecolor='gray')

        self.fig = plt.figure(figsize=figsize)
        # Upper left
        self.left = self.fig.add_subplot(221)
        # Upper right
        self.right = self.fig.add_subplot(222)
        # Whole lower
        self.lower = self.fig.add_subplot(212)

    def plot_result(self, figsize:tuple = (12,6), axis: list=[0,1], show=
                                            True) -> Union[None, 'Axes']:
        '''
        Plots GMM result. If data is more that two axis', you can select
        which axis' to plot in axis parameter
        '''
        assert len(axis) == 2, 'Length of axis must be 2'
        if self._plot_flag == False:
            self._init_plot(figsize)

        # Handle 1d case
        if self.dim > 1:
            centroids = self.components['mean'][:,axis].T
            X = self.X[:,axis].T
        elif self.dim == 1:
            # Effectively give y-values (zeros) so I can plot them
            centroids = np.column_stack([self.components['mean'], np.
                                                    zeros_like(self.
                                                    components['mean'])]).T
            X = np.column_stack([self.X, np.zeros_like(self.X)]).T

        # EM subplot
        self.left.scatter(*X, **self.X_kwargs)
        self.left.scatter(*centroids, **self.centroid_kwargs)
```

```
        # Handle 1d case
        if self.dim > 1:
            for i, c in enumerate(self.components):
                self._draw_ellipse(c['mean'][axis], c['cov'][axis], self.
                                                        left, alpha=c['mix'],
                                                         color=self.colors[i%
                                                        self.nc])
        else:
            xrange = np.linspace(X[0].min()-self.X_std, X[0].max()+self.
                                                    X_std, 200)
            for i, c in enumerate(self.components):
                self.left.plot(xrange, np.log(c['mix']*norm.pdf(xrange, c[
                                                        'mean'], c['cov'][0][
                                                        0])+1.1), color=self.
                                                        colors[i%self.nc])

        self.left.set_title('GMM-Components')
        # Categorized data point subplot (hard labels)
        labels = self.get_labels()
        self.right.scatter(*X, c=[self.colors[j%self.nc] for j in labels],
                                                     **self.X_kwargs)
        self.right.scatter(*centroids, **self.centroid_kwargs)
        self.right.set_title('Hard clustering')

        # Likelihood subplot
        self.lower.grid()
        # Drop first since it is usually really bad, and makes the plot
                                                    ugly
        self.lower.plot(np.arange(1,len(self.hood_history)), self.
                                                    hood_history[1:])
        self.lower.set_title('Log-Likelihood')
        self.lower.set_xlabel('Iterations')

        plt.suptitle(f'Iteration {self.em_iterations}')

        self.fig.tight_layout()
        if show: self.show()

    def show(self):
        '''Ensures that _plot_flag gets assigned correctly'''
        self._plot_flag = False
        plt.show()

    def get_mise(self, validation_data: np.ndarray) -> float:
        '''Approximation of mean integrated square error'''
        second_term = np.zeros(validation_data.shape[0])

        for c in self.components:
            second_term += c['mix'] * mvn.pdf(x=validation_data, mean=c['
                                                    mean'], cov=c['cov'])
        second_term = 2/validation_data.shape[0] * second_term.sum()

        cum = 0
```

```
        for ci in self.components:
            for cj in self.components:
                cum += ci['mix'] * cj['mix'] * mvn.pdf(x=ci['mean'], mean=
                                                       cj['mean'], cov=ci['
                                                       cov'] + cj['cov'])
        cum = cum - second_term

        return cum

    def get_bic(self) -> float:
        '''Returns BIC value'''
        d = self.dim
        penalty = (self.k*(d+(d+1)*d/2) + (self.k-1))/2 * np.log(self.N)
        return self.hood_history[-1] - penalty

    def get_aic(self) -> float:
        '''Returns AIC value'''
        d = self.dim
        penalty = 2*(self.k*(d+(d+1)*d/2) + (self.k-1))/self.N
        return -2*self.hood_history[-1]/self.N - penalty

if __name__ == '__main__':
    np.random.seed(42069)
    from sklearn.datasets import load_iris
    # data=load_iris()['data'][:,3].reshape(-1,1)
    data=load_iris()['data']

    axis = [0,3]
    k = 3

    JohnWick = GMM(k)
    # JohnWick.fit(data)
    JohnWick.fit_animate(data, axis=axis)
    # JohnWick.plot_result(axis=[0,3])
```

B Rest of code

```
from EM_GMM import GMM
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
np.random.seed(42069)

if __name__ == '__main__':
    def task1():
        df = pd.read_csv('largest_cities.csv')
        names = df['city']
        X = df[['lng', 'lat']].values

        bics2d = []
        models2d = []
```

```
    for i in range(6):
        models = [GMM(k=i).fit(X) for i in range(2, 19)]
        models2d.append(models)
        bics2d.append([m.get_bic() for m in models])

    bics2d = np.array(bics2d)
    models2d = np.array(models2d)
    best_idx = np.unravel_index(np.argmax(bics2d), bics2d.shape)
    best_model = models2d[best_idx]

    print(bics2d)
    print(best_idx)

    bics = bics2d[best_idx[0]]
    models = models2d[best_idx[0]]
    print(f'''
        Best model:
        Number of components = {best_model.k}
        Final loglikelihood = {best_model.hood_history[-1]}
        BIC = {bics[best_idx[1]]}
    ''')

    dicto = {'models':models}
    for i, b in enumerate(bics2d, start=1):
        dicto[f'BIC_{i}']=b

    df_results = pd.DataFrame(dicto)

    print(df_results.to_latex(index=False, float_format=lambda x: f'{x
                                            :.2f}'))

    best_model.plot_result(show=False, figsize=(9,5))
    # plt.savefig('GMM_task1.pdf')
    best_model.show()

def task2():
    df = pd.read_csv('ex8_2.csv', header=None)
    X = df.values.flatten()

    # Inherit old GMM class for its plotting abilities
    class GMM2(GMM):
        def _prepare_before_fit(self, X: np.ndarray) -> None:
            '''
            Prepares object attributes and such before fitting
            '''
            super()._prepare_before_fit(X)
            # Set component 0 mean to 0
            self.components[0]['mean'] = 0
            # Set standard deviations to 1
            self.components['cov'].fill(1)

        def _M_step(self) -> None:
```

```
                w = self.weights[:,1]
                w_sum = w.sum()
                # Update component mean
                self.components[1]['mean'] = w@self.X/w_sum
                # Update component mixing probability
                self.components[1]['mix'] = w_sum/self.N
                self.components[0]['mix'] = 1 - self.components[1]['mix']

        # Override class methods
        gmm = GMM2(k=2)
        gmm.fit(X.reshape(-1,1))
        # gmm.fit_animate(X.reshape(-1,1))
        gmm.plot_result(show=False, figsize=(9,5))
        plt.savefig('GMM_task2.pdf')
        gmm.show()

    task2()
```