CS 509 – Assignment 3

Yi Ren (002269013), Wentao Lu (002276355)

As required by Prof. Mohammed Ayoub in the email, in this assignment we have to use the diagonal case of the covariance, so we assume that each component Gaussian has a diagonal covariance matrix.

First, we load our dataset into Python, and then attribute the missing values of even-numbered x3 data in category w1 to 0.

```
import os
import numpy as np
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture
import visualization

if __name__ == '__main__':
    datafile = os.path.join(os.getcwd(), 'Assignment3_dataset.txt')
    w1 = np.loadtxt(datafile, skiprows=1, max_rows=10)
    w1_missing = np.loadtxt(datafile, skiprows=1, max_rows=10)
    w1_missing[1::2, 2] = 0
    w2 = np.loadtxt(datafile, skiprows=12, max_rows=10)
    w3 = np.loadtxt(datafile, skiprows=23, max_rows=10)

    data_complete = np.vstack((w1, w2, w3))
    data_missing = np.vstack((w1_missing, w2, w3))
```

1. estimate the mean and covariance of the distribution

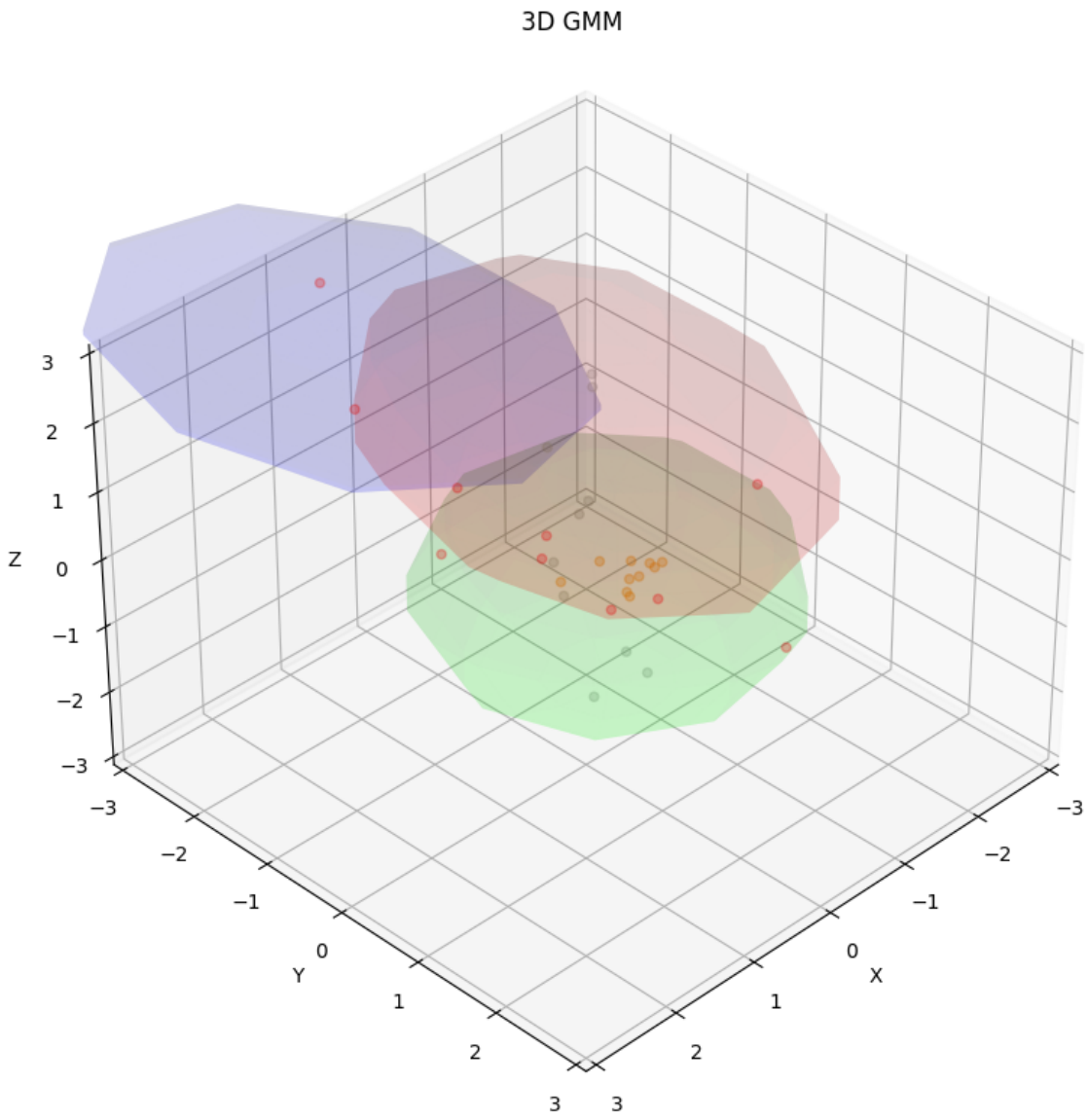
In this problem we have 3 classes w1, w2 and w3, but some data in w1 is missing. Hence, we need a mixture of Gaussian that has 3 component Gaussians, under the hood, the GMM model will apply the EM algorithm to fit our dataset. From the perspective of GMM, it sees all the 30 data points and knows they come from a mixture of 3 Gaussian components, and will try to find the parameters of each component Gaussian as well as their weights. This is the code:

```
    gmm = GaussianMixture(n_components=3, covariance_type='diag', max_iter=1000,
random_state=3)
    gmm.means_ = np.zeros((1, 3))
    gmm.covariances_ = np.identity(3)
    gmm.fit(data_missing)

    with np.printoptions(precision=4, suppress=True):
        print(gmm.weights_)
        print(gmm.means_)
        print(gmm.covariances_)

    visualization.visualize_3d_gmm(data_missing, gmm.weights_, gmm.means_.T,
np.sqrt(gmm.covariances_).T)
```

After the GMM model fits the data (with missing values filled with 0), we print out the weights, means and covariances of the Gaussian mixture, and then visualize the result in a figure.

```
# weight: [ 0.1633  0.7700  0.0667 ]

# mu1:    [ 0.2241  0.2733  2.1275 ]
# mu2:    [ 0.0698  0.3686 -0.0528 ]
# mu3:    [-0.8992 -4.2983 -0.075  ]

# sigma1: [ 0.3408  0       0      ]
#         [ 0       0.7615  0      ]
#         [ 0       0       0.3466 ]

# sigma2: [ 0.3863  0       0      ]
#         [ 0       0.3942  0      ]
#         [ 0       0       0.2527 ]

# sigma3: [ 0.4912  0       0      ]
#         [ 0       1.0052  0      ]
#         [ 0       0       0.0056 ]
```

We can easily verify that the weights sum up to 1, the mean and diagonal covariance matrix is given above for each component Gaussian. It bears mentioning that the true class label is not the same as a component Gaussian, for example, `mu2, sigma2` is not the mean and covariance of class `w2`, it's just the parameters of the second component Gaussian, since the GMM does not know the three classes `w1, w2, w3`, it just tries to fit all data points in a mixture of Gaussians.



In the visualization, 30 data points are scattered in the 3D space, every data point has a color which represents its true class label. Besides, the three red, green and blue sphere represent the space that each component Gaussian covers within one standard deviation, each sphere tries to group some similar data points into a cluster. Ideally, each component Gaussian should be able to cluster data points of the same true label, or the same color, in our case, the data size is too small, and the static 3D plot is also limited in displaying the position of each point (for example, some grey points are floating in the air but they all seem to be grounding on the XY plane), so there's some deviations as we see, however, overall the mixture Gaussian did a good job in clustering.

It is also worth mentioning that the component Gaussian (spheres in the figure) can overlap, so that some data points are enclosed in more than one sphere, in this case the weight of each component is important.

2 & 3. compare your final estimate with that for the case when there is no missing data

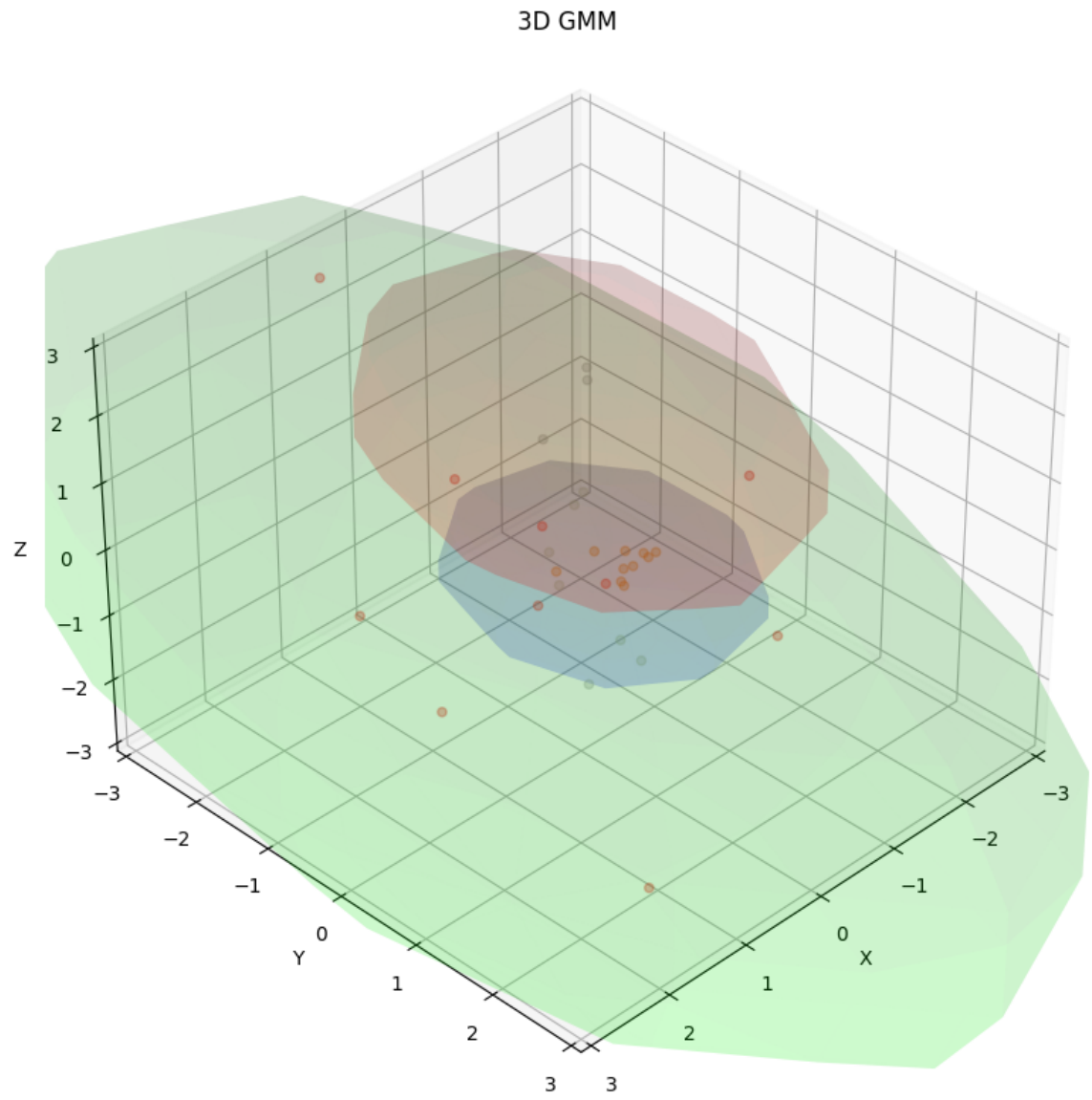
When there's no missing data, we just do the same using GMM to estimate the Gaussian mixture, but passing in the complete dataset rather than the dataset with missing values.

Note that if we only look at one class and focus on the EM algorithm rather than GMM, so there's only one multivariate Gaussian, we can just use MLE estimate for a complete dataset, but this is wrong in this problem since we are dealing with mixture of Gaussians.

```
    # create a new gmm2 model
    gmm2 = GaussianMixture(n_components=3, covariance_type='diag', max_iter=1000,
random_state=3)
    gmm2.means_ = np.zeros((1, 3))
    gmm2.covariances_ = np.identity(3)
    gmm2.fit(data_complete)  # use the complete data without missing values

    with np.printoptions(precision=4, suppress=True):
        print(gmm2.weights_)
        print(gmm2.means_)
        print(gmm2.covariances_)

    visualization.visualize_3d_gmm(data_complete, gmm2.weights_, gmm2.means_.T,
np.sqrt(gmm2.covariances_).T)
```

```
# weight: [ 0.1687  0.1789  0.6525 ]

# mu1:    [ 0.2275  0.2631  2.0964 ]
# mu2:    [-0.5082 -1.4775 -2.4453 ]
# mu3:    [ 0.1271  0.401   0.0116 ]

# sigma1: [ 0.3370  0       0      ]
#         [ 0       0.7594  0      ]
#         [ 0       0       0.3694 ]

# sigma2: [ 1.0598  0       0      ]
#         [ 0       5.5561  0      ]
#         [ 0       0       2.4227 ]

# sigma3: [ 0.2143  0       0      ]
#         [ 0       0.3297  0      ]
#         [ 0       0       0.093  ]
```

In contrast to the case when missing values are present, we can see that the result is very different, it's not easy to look at the numbers so we also made a plot for this case.



When the data is complete, we can see that one sphere is much bigger than before, this means that one component Gaussian has higher variance than before. The reason for this is because, our missing values such as -3.4, -4.7 and -2.6 in x3 of w1 have very large deviations from zero, when they are fed to the GMM model, the model tries to make a larger sphere or component Gaussian to enclose these points. In other words, most missing values in this problem happen to be the "outliers" so the first figure is more compact.

4a. explain AIC and BIC

Akaike's Information Criterion (AIC) and Bayesian Information Criterion (BIC) are two criteria for estimating the relative quality of statistical models for a given dataset, both of them serve the purpose of model selection.

In case of logistic regressions, for example, AIC is defined as: `AIC = -2/N * LL + 2 * k/N` , where `N` is the number of examples in the training dataset, `LL` is the log-likelihood of the model on the training dataset, and `k` is the number of parameters in the model. It estimates the relative amount of information lost by a given model, so the less the value, the less information will be lost, and generally we have better model, so we want to select the model with the lowest AIC. Compared to BIC, AIC puts more emphasis on model performance on the training dataset, so it can select complex models which may prone to overfitting. It penalizes the inclusion of additional parameters and a small data size.

BIC is similar to AIC, but more appropriate for models fit under the maximum likelihood estimation framework, it is defined as: `BIC = -2 * LL + log(N) * k` . Compared to AIC, BIC is more suitable if we have a set of candidate models to select from, and it penalizes the model more for its complexity and additional parameters, so a model selected by BIC is less likely to overfit, but can suffer from underfitting for small datasets.

4b. Implement AIC and BIC in Python

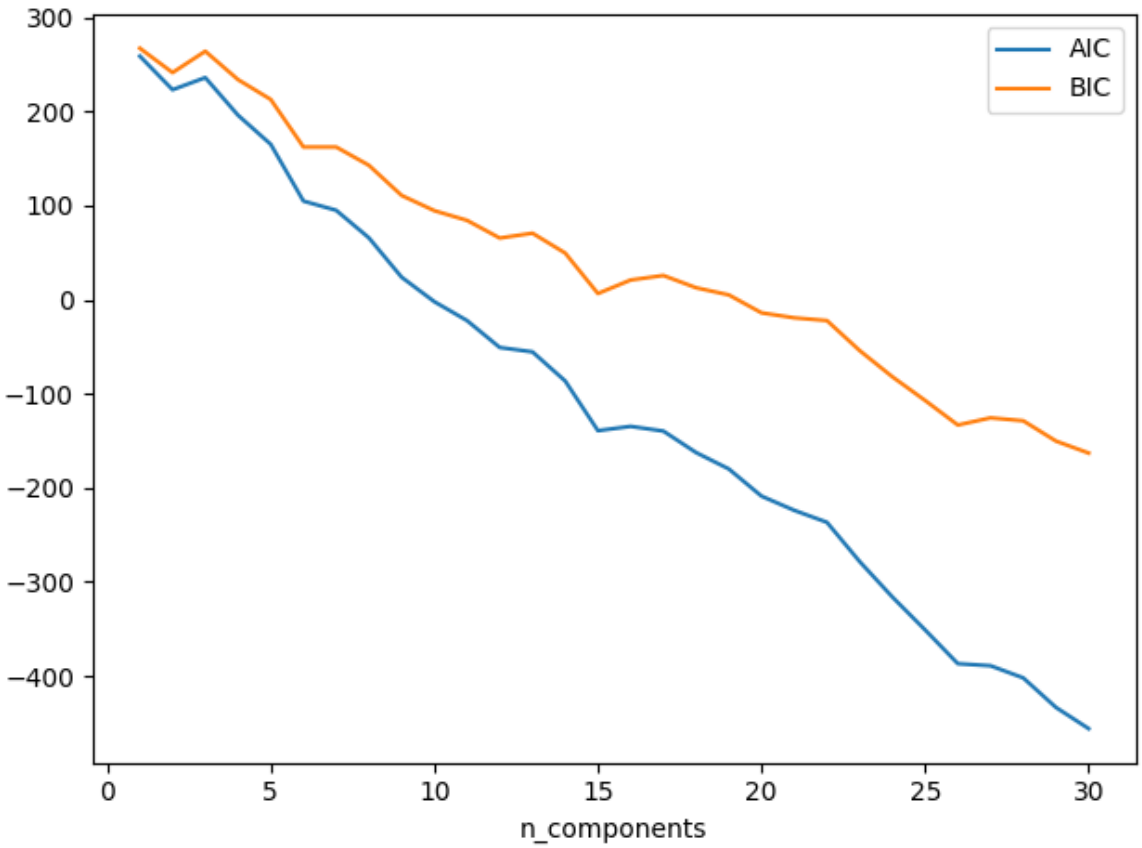
The `GaussianMixture()` method from scikit-learn library has two ready-to-use methods `aic()` and `bic()` , so we can just find AIC, BIC by fitting our dataset to a Gaussian mixture model.

```
from sklearn.mixture import GaussianMixture

def GMM_AIC_BIC(X, n_component):
    clf = GaussianMixture(n_components=n_component, covariance_type='diag', max_iter=1000,
random_state=3)
    clf.means_ = np.zeros((1, 3))
    clf.covariances_ = np.identity(3)
    clf.fit(X)
    aic = clf.aic(X)
    bic = clf.bic(X)
    return aic, bic
```

4c. Draw the AIC and BIC curves for your dataset

We have 30 data points in total, so we can have a mixture of at most 30 component Gaussians, we just loop through each number from 1 to 30, fit the corresponding GMM model, and plot the values of AIC and BIC to visualize this.

```
    aics = []
    bics = []
    for n_component in np.arange(1, 31):
        aic, bic = GMM_AIC_BIC(data_missing, n_component)
        aics.append(aic)
        bics.append(bic)

    plt.plot(np.arange(1, 31), aics, label='AIC')
    plt.plot(np.arange(1, 31), bics, label='BIC')
    plt.xlabel('n_components')
    plt.legend()
    plt.show()
```



4d. What is the best number of components to use

In the figure above, we can see that both AIC and BIC are nearly monotonically decreasing as the number of components increases, with only a few exceptions at the "elbow point". Frankly speaking, we cannot decide a reasonable number of components to use from this graph, given such a small dataset of 30 data points.

Statistically speaking, when data size is too small (less than 50), most statistical rules will not apply even for a single dimension (with only 1 feature), thus, there's no plausible conclusion to make.

In fact, since the data size is so small, our GMM model must be underfitting, that's why any rise in the number of components can make the model slightly better. In the extreme case, each data point belongs to its own component Gaussian distribution, and the model becomes more accurate. Both AIC and BIC are trading off between a model's goodness of fit against model complexity, trying to find a sweet spot in the middle of underfitting and overfitting. If we have thousands of data points in each class `w1, w2, w3`, AIC and BIC would immediately goes up when the number of components is greater than 3, so as to penalize model complexity and prevent overfitting, just like what k-means does, but that's not our case for this problem. In theory, AIC is more lenient about overfitting, while BIC is more rigid on this and tend to penalizes it much more, that's why in our graph BIC decreases less intensely than AIC.