Implementing a Gaussian Mixture Model

by Siraj Hatoum, Nathan Murstein

March 2, 2022

Contents

1 Gaussian Mixture Model Overview

A Gaussian Mixture Model (GMM) is a probabilistic density estimation model that describes the distribution of data using a mixture of several Gaussians. It is an unsupervised method, meaning that it does not require labeled data and we do not need to know the distributions, or the distribution from which each instance was generated in the training data. GMMs are most commonly used for soft clustering, which differs from hard clustering in that we get probabilistic cluster assignments. An important fundamental property of GMMs is that it is based on a generative assumption that the data comes from a mixture of several Gaussian distributions, the parameters (means, mixture probabilities, and covariances) of which we will estimate using the Expectation-Maximization algorithm.

A very simple illustrative example is modeling human height: heights for each gender are normally distributed within the population, with a higher mean for men than for women. Therefore, we can think of all human heights as coming from one of those two distributions. In GMMs, given a dataset of heights, we would learn the Gaussian distributions and assign each instance a probability of whether they come from the male or female distribution.

1.1 Advantages/Disadvantages

GMMs have several advantages over other common clustering methods. First, in many real-world applications, the simple and deterministic nature of common methods like k-means leads to poor performance. When clustering errors have a high cost, the certainty of cluster assignment becomes important. GMMs can assign a higher clustering certainty to instances near cluster centers, unlike k-means. Additionally, k-means creates spherical clusters (in which the radius is the distance between the cluster centroid and the furthest instance in the cluster), which is not applicable to every dataset. In many cases, either due the nature of the dataset itself, or due to applied transformations, spherical clusters would not fit the data well, and GMMs can handle a greater variety of cluster shapes, especially ellipses.

However, GMMs also have drawbacks. One drawback is that they rely on stochastic initialization, so they can miss the global optimum solution and converge on a local optimum. However, as we will discuss, this can be mitigated either using multiple random initializations, or by using a method like k-means to obtain initialization parameters. Additionally, GMMs are not as fast as simpler models, so if convergence time is a priority, they may not be the best choice.

1.2 Applications

GMMs have many applications. Broadly, a GMM model may be a good fit in any situation where data appears multimodal, since GMMs can be thought of as modeling multimodal data as a mixture unimodal gaussians. GMMs are also an especially useful tool when data can be assumed to follow a Gaussian distribution. For example, GMMs have been applied in gene expression analysis, where they were used to detect differentially expressed genes between two conditions and model which genes could contribute to a certain disease. Other applications have included feature extraction from speech data to use in speech recognition, and multiple object tracking in videos (where each Gaussian represents an object, and the mixtures and means represent the object's location at each frame).

It is important to note that although GMMs are mostly used for clustering due to their flexibility and performance, they are fundamentally a density estimation algorithm that describes a data distribution. Many of the most exciting applications of GMMs for generating data rely on this property to create new data. By using a very large number of gaussians (more than would be visually apparent), we can create a fairly accurate density model that facilitates data creation, such as learning and recreating a unique handwriting style.

2 Optimization Problem and Solution Approach

GMMs perform soft clustering based on a generative assumption, whereby all observations have been generated by a cluster who's generative function follows some probabilistic distribution. Specifically, he model assumes that the underlying distribution of each clusters k is a multivariate Gaussian with mean μ_k and covariance COV_k.

Based on these assumptions, GMMs calculate the probability that a given observation $\vec{x}$ with p features is generated by (belongs to) a cluster using the definition of the probability function of a multivariate Gaussian distribution.

$$p(x|\mu, COV) = \frac{1}{(2\pi)^{\frac{p}{2}}|COV|^{\frac{1}{2}}} \exp\bigg(-\frac{1}{2}(x-\mu)^T COV^{-1}(x-\mu)\bigg)$$

Using this equation, the GMM performs soft clustering based on the relative probability of every observation in the data set belonging to each cluster. However, as this is an unsupervised clustering task the cluster means and covariances are unknown and need to be estimated based on the input observations and number of clusters.

As such, the variables of the model are the cluster means μ_k and cluster covariances COV_k. In order to complete the clustering task, the GMM needs to first estimate these variables to maximize the likelihood that all n observations in the data set were generated on of K clusters. The optimization approach also uses latent variables z, which take two possible values: 1 when an instance comes from Gaussian k, and 0 otherwise. We use the assumption that all latent variables are conditionally independent to calculate the probability of a set of all latent variables z which is used to derive the equation. The derivation of the optimization problem invokes these latent variables. The optimization problem can expressed as:

$$maximize\bigg(\prod_{i=1}^{n}\prod_{k=1}^{K} p(\vec{x}_i|\vec{\mu}_k, COV_k)\bigg)$$

A popular approach to solving this optimization problem is the expectation-maximization (EM) algorithm. EM is an iterative two step process that begins by calculating the Expectation z_k^i of a observation x_i belonging to cluster k, defined as:

$$z_k^i = \frac{p(\vec{x}_i|\vec{\mu}_k, COV_k)}{\sum_{k=1}^{K} p(\vec{x}_i|\vec{\mu}_k, COV_k)}$$

The expectation calculation adjusts the likelihood terms across all observation-cluster combinations such that the expectation that each observation is generated by any one of the k clusters is 1. This is key for the next step of the process, Maximization, where the assumed means and covariances are updated based on the expectation values.

$$\vec{\mu}_k = \frac{1}{z_k}\sum_{i=1}^{n} z_i^k \vec{x}_i$$

$$COV_k = \frac{1}{z_k}\sum_{i=1}^{n} z_k^i(\vec{x}_i - \vec{\mu}_k)(\vec{x}_i - \vec{\mu}_k)^T$$

Expectations are used to weight the contribution of each observation when calculating the cluster mean and covariance updates. The expectation step is then repeated based on the new values and the process iterates until the values converge.

3 Model Implementation

Having outlined the optimization problem and the Expectation-Maximization solution approach, we will detail the programming functions defined to implement the GMM from scratch and compare their performance to the black-box GMM provided in the sci-kit learn module.

3.1 Modeling Approach

Appendix A.1 outlines all the functions defined to support and run the GMM model. The process begins by initializing the clusters to kick-off the EM process, calculating the expectations based on cluster means and covariance, updating the cluster means and covariance based on the expectation output to maximize the likelihood function, and finally assessing the convergence of the model to stop iterating and output the results. The following sections detail our implementation for each of these functional steps.

3.1.1 Initialization

The initialization requirement is often overlooked when discussing the theory behind the EM algorithm. This is likely due to the variety of available approaches, nonetheless it is an essential step in the implementation that can significantly impact the performance of the model.

The key output of the initialization step can either be an expectation matrix or cluster means and covariance matrices, required to kick-off the EM process. A potential approach is to output an expectation matrix that assuming each observation is equally likely to fall within any of the available clusters. Another fairly direct approach involves randomly assigning observations to clusters and calculating the cluster means and covariances based on this assignment.

The main principle behind these approaches is to avoid biasing the model output by imposing uniform or random distributions at the beginning. While this is reasonable given our lack of prior knowledge, it is unlikely that the initialized parameters are close to the desired output. As such, the model may take longer to converge or my converge on a local optimum that is not truly representative of the dataset. A common method of avoiding these pitfalls is to run the dataset through another clustering model and use the output of that model to initialize the EM parameters. The assumption here is that the output of the initialization model will yield values closer to the desired output, allowing the GMM to converge more quickly and with higher certainty while refining the initial model's outputs. For this approach to be effective, the ideal initialization model would be:

- **Quick to converge** relative to the GMM in order to improve run-time
- **Simple**; the model should not require any additional hyperparameters to produce reliable results, this avoids complicating the GMM application
- **Robust to unbiased assumptions** to maintain a high degree of certainty in results across varying datasets

Based on these requirement KMeans is a popular initialization model in GMM applications due to its relative efficiency and simplicity. Appendix A.1.1 details the function used to implement a simple KMeans clustering model for the initialization of the cluster means. We assumed the covariance of the clusters were equal to the covariances of the full dataset to avoid biasing the results in cases where the cluster size output by the KMeans was not sufficiently large.

3.1.2 Expectation

After initialization of the cluster means and covariance matrices, the GMM function enters the EM loop, beginning with the expectation step. As discussed previously the expectation step calculates the relative likelihood of each observation belonging to each cluster using the following formulas.

$$p(x|\mu, COV) = \frac{1}{(2\pi)^{\frac{p}{2}}|COV|^{\frac{1}{2}}} \exp\Big(-\frac{1}{2}(x-\mu)^T COV^{-1}(x-\mu)\Big)$$

$$z_k^i = \frac{p(\vec{x}_i|\vec{\mu}_k, COV_k)}{\sum_{k=1}^K p(\vec{x}_i|\vec{\mu}_k, COV_k)}$$

Appendix A.1.2 details the functions used to perform this step. The primary adjustment made involved splitting the probability calculations into two steps. We first calculate the natural logarithm of the probability before calculating the probability. This served to avoid math range errors when executing the function, caused by evaluating the exponent and fraction terms separately. Additionally we applied a smoothing factor ϵ to avoid math errors caused by rounding small numbers to 0, such as evaluating the log of 0 or assigning an observation a probability of 0 of being assigned to a cluster.

Worth noting is the iterative approach of calculating the expectation for each observation-cluster combination. A potential improvement for future versions of the model may consider leveraging Matrix and Vector operations to reduce the number of calculations and improve run-time metrics.

3.1.3 Maximization

Having calculated the expectation matrix, we are able to begin the Maximization step. Here the key outputs are updated cluster means and covariances based on the estimated expectation. As discussed previously, the maximization step is defined by the following operations:

$$\vec{\mu}_k = \frac{1}{z_k}\sum_{i=1}^n z_i^k \vec{x}_i$$

$$COV_k = \frac{1}{z_k}\sum_{i=1}^n z_k^i(\vec{x}_i - \vec{\mu}_k)(\vec{x}_i - \vec{\mu}_k)^T$$

Appendix A.1.3 details the programmed implementation of these calculations.

3.1.4 Convergence

The final component of the GMM implementation is the stop condition, which is required to break the EM loop once the model has converged. In our implementation, a threshold value is used to assess the convergence of the model, as shown below.

```
diff = norm_z-new_norm_z
l2_diff = np.einsum('ij,ji->i', diff, diff.T).sum()/diff.size

if l2_diff < threshold:
    break
```

Where diff is a matrix of the difference between the previous expectation matrix and the current expectation matrix. We utilize Numpy's Einstein sum function to calculate the value of the squared elements of the diff matrix by evaluating the diagonal elements of the matrix multiplication of the matrix and it's transpose. This method was used in place of a for loop iteration over the elements to improve run-time performance. The diagonal elements are summed and divided by the number of elements in the diff matrix (clusters x observations) to arrive at a value representing the change in expectation matrix. If that value is less than the assigned threshold we determine that the model is close enough to convergence and break the loop.

Appendix A.1.4 shows the full implementation of the GMM function. The threshold can have a significance impact on model performance as it balances between the time required to converge and the quality of the output. As such, we implemented the threshold as an option input parameter to allow the user to vary to match their use case.

3.2 Black-box Comparison

After building the model and performing basic functionality tests, we move on to a side-by-side comparison between our model and the black-box sci-kit learn GMM function. Below we discuss our chosen performance metrics, testing approach, and results.

3.2.1 Performance Metrics

Run-time

We define the run-time of the model simply as the time taken between calling the model and outputting model results. We chose this as a key metric since it is directly related to the efficiency and scalability of the model. When comparing models, a lower run-time is more desirable all things being equal because it implies the model is better optimized and more efficient. During testing the run-time is determined as the difference between timestamps generated right after the function completes its run and right before the function is called.

Silhouette Score

The silhouette score is a measure for the validity of a cluster that incorporates both the between-cluster and within cluster separation, as shown in the metric definition below.

$$s = \frac{b-a}{max(a,b)}$$

Where a is the mean distance between the sample and all other observations in the cluster and b is the distance between the sample and all other points in the nearest cluster. The silhouette score of the model output is calculated as he mean of the silhouette score for each sample.

The silhouette score can range between -1 to 1 with a value of 0.5 or above being a strong indicator of valid cluster. The silhouette score is a popular method to measure clustering output quality in the absence of cluster labels, which is often the case for most clustering tasks, as such it is a very relevant metric for our model comparison. In the test we use the silhouette score function available in the sci-kit learn module to calculate the silhouette score of the model outputs.

Rand Index Score

The rand index is used when the ground truth cluster assignment is known to assess the similarity of the model clusters with the known clustering. The metric is bounded between 0 and 1 with a score of 1 indicating perfect alignment between the model output and the ground truth. Unlike the silhouette score the rand index makes no assumptions as to the structure of the clusters when evaluating model output.

Although rand index is not a popular approach as the ground truth is often not available when performing unsupervised clustering tasks, the use of a synthetic data generator in our testing allows us leverage this metric for our analysis. The synthetic data is generated using sci-kit learn's make blobs function and the rand index is calculated using their rand score function.

3.2.2 Testing Approach

After defining the relevant performance metrics for comparison, we outline the variables across which the models will be tested.

Number of Observations: we vary the size of the data set between 100 to 900 observations to assess how the performance of the model varies with record number.

Number of Features: we vary the number of features in the data set between 2 and 30 to assess variations in performance with increasing features.

Number of Clusters: we vary the number of clusters in the data set between 2 and 20 to assess variations in performance with increasing clusters.

In addition we run the test 5 times for each variable o asses variability in the model performance. Appendix A.2.1 outlines the functions defined to support the iterative testing process described above.

3.2.3 Results

Figure 1 in Appendix A.2.2 describes the model performance based the previously outlined metrics over our three independent variables.

Cluster Validity Analysis

The graphs suggest that the clustering validity scores of the two models are comparable, both in value and variation, over increasing observations and feature counts. Consistently scoring silhouette scores above 0.5 and rand index scores of 0.9 or higher, both models appear to be correctly identifying clusters at a high rate.

Worth noting, is the decreasing variation in both scores as the number of features increase while the average value of the scores remains fairly constant. This implies that the models produce more consistent outputs more feature data is available. The decrease in output variability may be due to the increase in data points per observation, resulting in a higher degree of certainty in cluster assignments. This hypothesis is consistent with the trend seen when the number of observations increases, namely, the variability of the scores across multiple runs remains constant.

The cluster validity performance of the two models appears to diverge when varying the number of clusters in the data. Our GMM model seems to perform worse than the black-box model in both cluster validity metrics when the number of clusters exceeds 4. Interestingly, after the performance drops noticeably at that threshold, the difference in performance remains consistent over the range of observed values.

The average silhouette score of both models appears to decrease linearly as the number of cluster increases. In comparison, the average rand index score remains somewhat constant after an initial drop. Based on these observations, it is reasonable to assume that the consistent drop in the silhouette score for larger cluster numbers is an inherent characteristic of the metric or a unintended effect of the synthetic data generation process and not a significant decline in model performance. Our working assumption, at the time of writing, is that the separation of the generated clusters is not scaling with the number of clusters generated, resulting in a reduced silhouette score. More testing is recommended to adequately support this explanation.

Another key difference observed when varying cluster numbers is the variability of the rand index score. As the number of clusters increases the rand index score appear to become less variable while our GMM implementation shows no such trend. Based on these results we recommend future improvements to the model focus on investigating the cause of the relative drop in performance and consistent variability in rand index as the number of observed clusters increases. We advise testing stricter thresholds for the stopping condition, to assess the potential impact on relative performance.

Run-time Performance

While the cluster identification performance of both models was encouragingly comparable, the run-time performance is not. In particular, our model takes a considerably longer period of time to converge across all observations.

Even more concerning is the trend in run-time performance with increasing observation numbers and cluster numbers. Our GMM average run-time increases linearly with either variable, although at a significantly higher slope for cluster number, while the black-box model maintains a constant run time at less than 0.5 seconds.

While increasing the cluster number to 20 resulted in run-times as high as 12 seconds, increasing feature numbers from 2 to 30 resulted in a relatively consistent run-time between 0.4 to 0.7 seconds. Based on these trends, believe that the primary cause of the run time discrepancy is the use of for-loops to compute the expectation step, as that requires our implementation to run through each combination of cluster and observation separately. As mentioned previously, we recommend replacing the current for-loop reliant expectation calculation process with one that leverages the efficiency of matrix multiplications, perhaps by taking a linear approximation of the likelihood function that is more amenable to representation by matrices.

As it stands, despite satisfactory cluster validity performance in most observed use cases, the required run-time of the model is the most significant hindrance to the solubility and usability of our implementation.

3.3 Challenges and Lessons Learned

During the implementation of the model, a key challenge was avoiding math errors caused by invalid operation (e.g. divide by zero) or extremely large numbers. These often occurred due to large scale features, poorly initialized parameters, or extremely small numbers that rounded to 0. These errors primarily triggered during the expectation calculation step. In order to resolve these issues we developed a few workarounds:

- **KMeans initialization** to output higher expectation numbers and avoid 0 expectation sums for clusters in the normalization step
- **MinMax standardization** to avoid large number errors due to large scale features without significantly changing data distribution
- **Log Expectation** to scale down resultant terms and avoid large number errors
- **Epsilon Addition** to avoid zero division errors and log(0) errors

A Model Implementation

A.1 Modeling Approach

A.1.1 KMeans Initialization

```
# Define basic KMeans function to initialize GMM clusters

def fit_kmeans(X, k):

    X = pd.DataFrame(X)

    obs = len(X)
    clus_lab = list(range(k))

    clus_0 = r.choices(population=clus_lab   ,k=obs)
    clus_1 = []

    means_0 = {}

    for lab in clus_lab:
        means_0[lab] = X[pd.Series(clus_0)==lab].mean().values

    iteration = 0
    while True:
        iteration+=1

        clus_means = np.array(list(means_0.values()))
        clus_labels = np.array(clus_0)
        new_colors = clus_means[clus_labels]

        clus_1 = []

        for i in range(obs):
            observation = X.iloc[i].values
            min_dist = distance.euclidean(observation, means_0[lab])

            for lab in clus_lab:
                dist = distance.euclidean(observation, means_0[lab])
                if dist < min_dist or lab == 0:
                    min_dist = dist
                    cluster = lab

            clus_1.append(cluster)

        for lab in clus_lab:
            means_0[lab] = X[pd.Series(clus_1)==lab].mean().fillna(0).values

        if clus_0 == clus_1:
            break

        clus_0 = clus_1

    return clus_0, means_0
```

A.1.2 Expectation Calculation

```
epsilon = 0.0000001

# calculate log of observation-cluster probability
log_z_i_k = lambda x_i, u_k, cov_k: -0.5*(len(x_i)*log(2*pi)
                                          +log(
                                              np.linalg.det(cov_k)
                                              +epsilon
                                          )
                                          +(
                                              (x_i-u_k)
                                              @ np.linalg.inv(cov_k)
                                              @ (x_i-u_k).T)
                                         )

# calculate observation-cluster probability
z_i_k = lambda x_i, u_k, cov_k : exp(log_z_i_k(x_i, u_k, cov_k))

# Function to calculate observation
# expectations based on cluster distributions
def gmm_proba(X, clus_means, clus_covs):
    epsilon = 0.0000001
    k = clus_means.shape[0]
    obs = X.shape[0]
    z = np.empty((k, obs))

    for cluster in range(k):
        for observation in range(obs):
            # Calculate new expectation value
            z[cluster, observation] = z_i_k(
                X[observation],
                clus_means[cluster],
                clus_covs[cluster]
            )
            + epsilon # avoid divide by 0 error

    return normalize(z, norm='l1', axis=0)
```

A.1.3 Maximization Calculation

```
# calculate cluster mean based on observations and
# cluster-observation probabilities
clus_mean = lambda X, z_k: 1/z_k.sum()*sum(
    [z_k[i]*X[i] for i in range(len(z_k))]
)

# calculate covarience using cluster mean
clus_cov = lambda X, u_k, z_k: 1/z_k.sum()*sum(
    [z_k[i]*(X[i]-u_k).T@(X[i]-u_k)
     for i in range(len(z_k))]
)
```

A.1.4 GMM Fit

```
def gmm_fit(X, k, threshold = 0.0005):

    obs = X.shape[0] # number of observations
    p = X.shape[1] # number of parameters

    _, clus_means = fit_kmeans(X, k)
    clus_means = np.matrix(
        pd.DataFrame(clus_means).T.values.reshape(k, p)
    )

    # initialize cluster cov based on full dataset and initialized means
    x_cov = pd.DataFrame(X).cov().values
    clus_covs = [x_cov for cluster in range(k)]

    # kick-off iterative expectation_maximiztion step

    change = True # iniialize change flag to enter loop
    i = 0

    norm_z = np.empty((k, obs))

    while True:
        means_int = clus_means.copy()
        covs_int = clus_covs.copy()

        # Expectation step
        new_norm_z = gmm_proba(X, clus_means, clus_covs)
        diff = norm_z-new_norm_z

        l2_diff = np.einsum('ij,ji->i', diff, diff.T).sum()/diff.size

        if l2_diff < threshold:
            break

        norm_z = new_norm_z


        # Maximization step
        for cluster in range(k):
            clus_means[cluster] = clus_mean(
                X, norm_z[cluster]
            ) # update cluster mean
            clus_covs[cluster] = clus_cov(
                X, clus_means[cluster], norm_z[cluster]
            ) # update cluster cov

    return clus_means, clus_covs
```

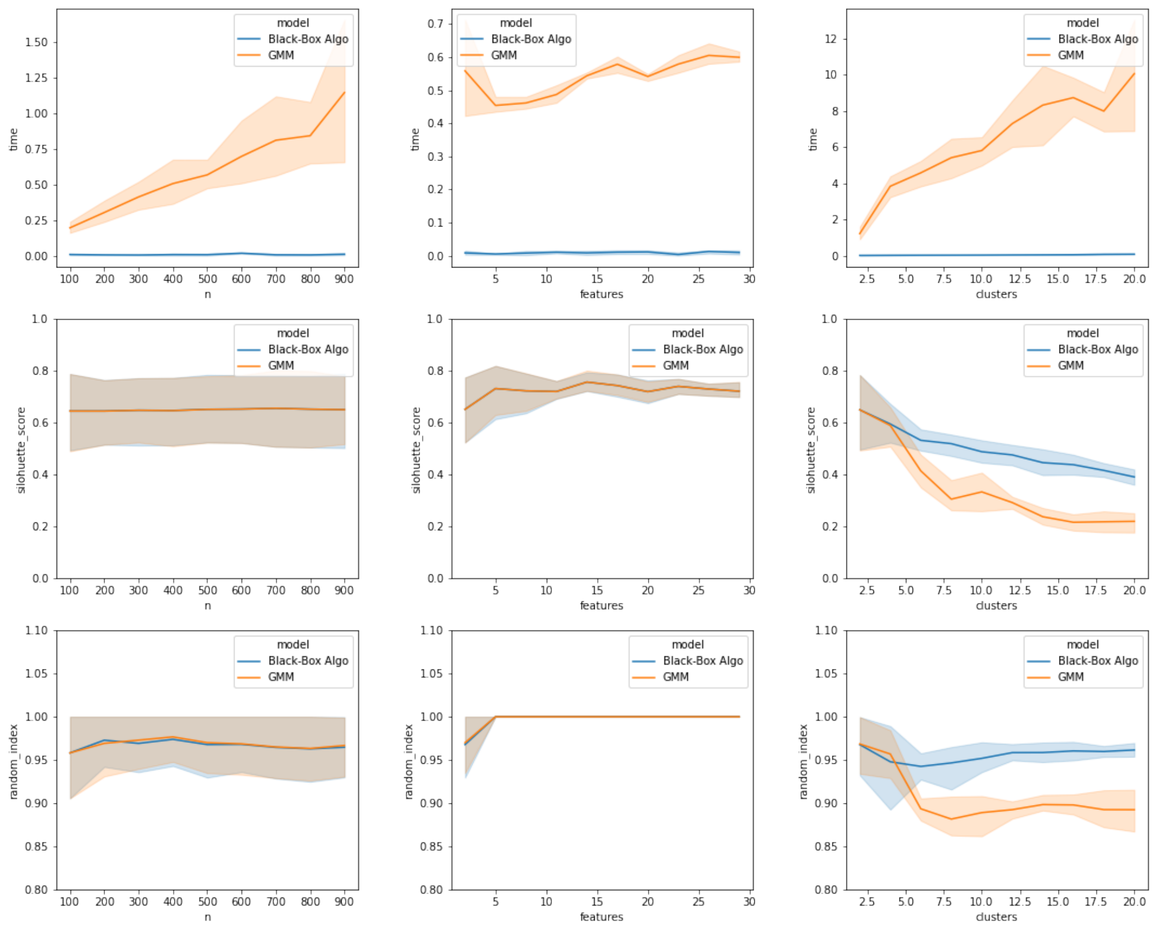
A.2 Black-box Comparison

A.2.1 Testing Approach

```python
# Function to store testing results for black-box and GMM
def perf_test(n= 500, feats= 2, k= 2, gmm_threshold = 0.0005):
    res = {'model': [], 'n': [], 'features': [], 'clusters': [],
           'time': [], 'log_threshold': [], 'silohuette_score': [],
           'random_index': []}

    for i in range(5):
        X, y = make_blobs(n_samples= n,
                          n_features = feats,
                          centers= k,
                          random_state= i)
        scaler = MinMaxScaler()
        X = scaler.fit_transform(X)

        gmm = GaussianMixture(n_components=k, random_state=0)

        # black box
        start = time.time()
        model = gmm.fit(X)
        end = time.time()

        bb_lab = model.predict(X)

        res['model'].append('Black-Box_Algo')
        res['n'].append(n)
        res['features'].append(feats)
        res['clusters'].append(k)
        res['time'].append(end-start)
        res['silohuette_score'].append(
            silhouette_score(X, bb_lab)
        )
        res['random_index'].append(rand_score(y, bb_lab))
        res['log_threshold'].append(log(gmm_threshold))

        # GMM
        start = time.time()
        means, covs = gmm_fit(X, k, threshold = gmm_threshold)
        end = time.time()

        gmm_lab = gmm_labels(X, means, covs)
        res['model'].append('GMM')
        res['n'].append(n)
        res['features'].append(feats)
        res['clusters'].append(k)
        res['time'].append(end-start)
        res['silohuette_score'].append(
            silhouette_score(X, gmm_lab)
        )
        res['random_index'].append(rand_score(y, gmm_lab))
        res['log_threshold'].append(log(gmm_threshold))

    return pd.DataFrame(res)
```

```
# Function to assign observations to clusters
# based on cluster distributions
def gmm_labels(X, clus_means, clus_covs):
    prob = gmm_proba(X, clus_means, clus_covs)
    return prob.argmax(axis=0)
```

A.2.2 Results



(a) Observation Varying Results (b) Feature Varying Results (c) Cluster Varying Results

Figure 1: Performance comparison results across our independent variable