MET CS 777 Project Presentation

Gaussian Mixture Models with Expectation-Maximization Algorithm: Implementation from Scratch in PySpark

Yi Rong
(yirong@bu.edu)

Advisor: Prof. Kia Teymourian

Boston University Metropolitan College
05/2021

Data Distributions: Circular and Oblong

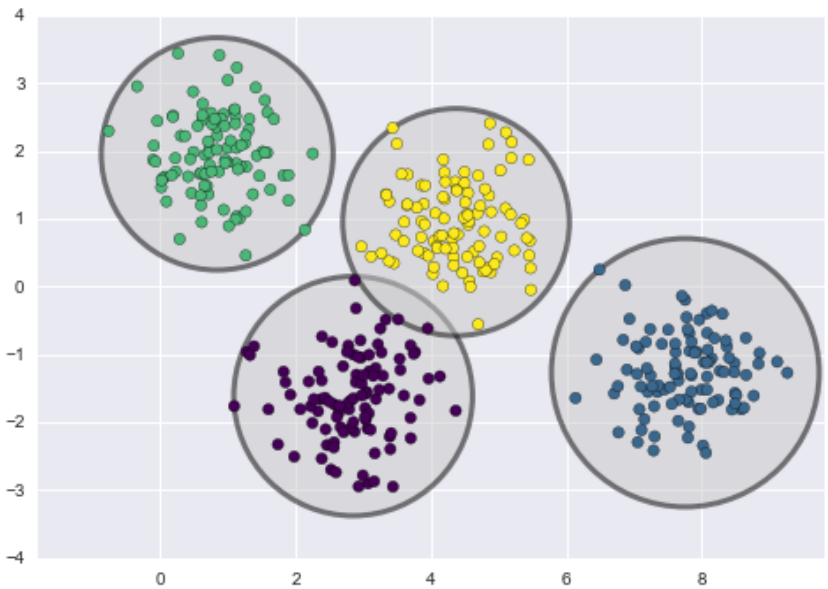


Figure: Circular Data Clustering

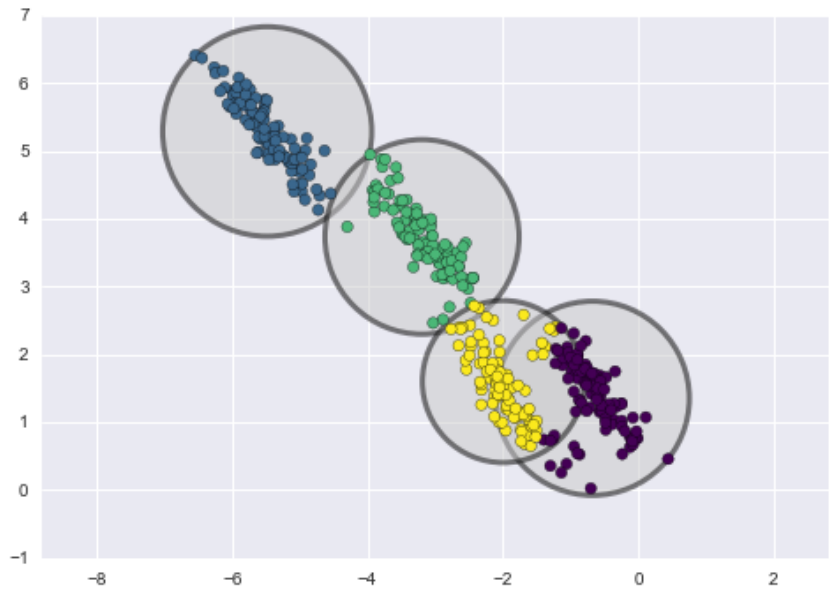


Figure: Oblong Data Clustering

Ref: https://towardsdatascience.com/gaussian-mixture-models-d13a5e915c8e

Clustering Methods: k-means vs GMM

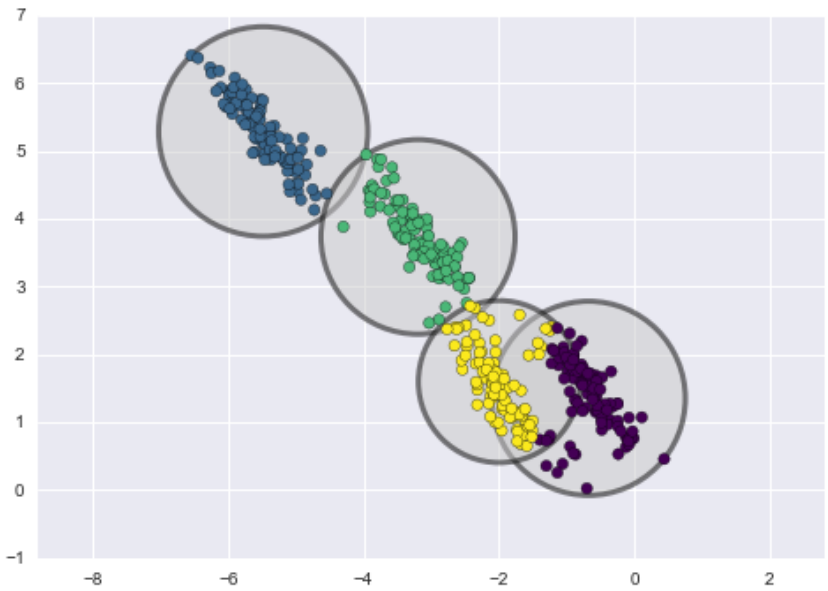


Figure: K-Means Clustering

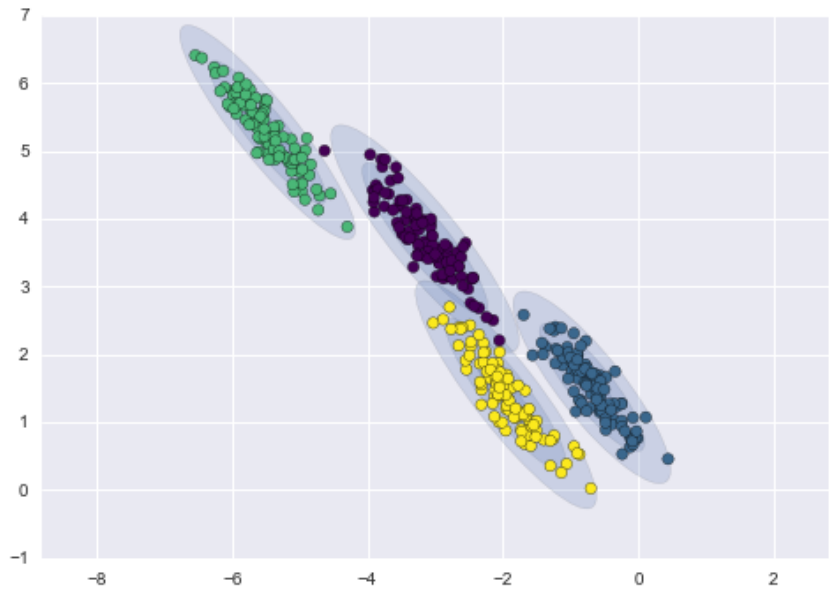


Figure: GMM Clustering

Ref: https://towardsdatascience.com/gaussian-mixture-models-d13a5e915c8e

Implementation Methods

- GMM with EM implemented from scratch
- GMM with EM implemented from library
- K-means implemented from library

Data Generation Code

```
n_feature = 2 # the number of features
n_components = 3 # the number of clusters
max_iter = 100 # the number of EM iterations
```

```
# generate isotropic Gaussian blobs (data from mixture Gaussian distributions)
X, y = make_blobs(n_samples=300, # the number of total samples
                  centers=n_components, # the number of clusters
                  n_features=n_feature, # the number of features
                  cluster_std = [0.8, 1.5, 1],
                  random_state=2) # 2
```

Figure: Generate Data with sklearn.datasets.make_blobs

Ref: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html

Data Visualization

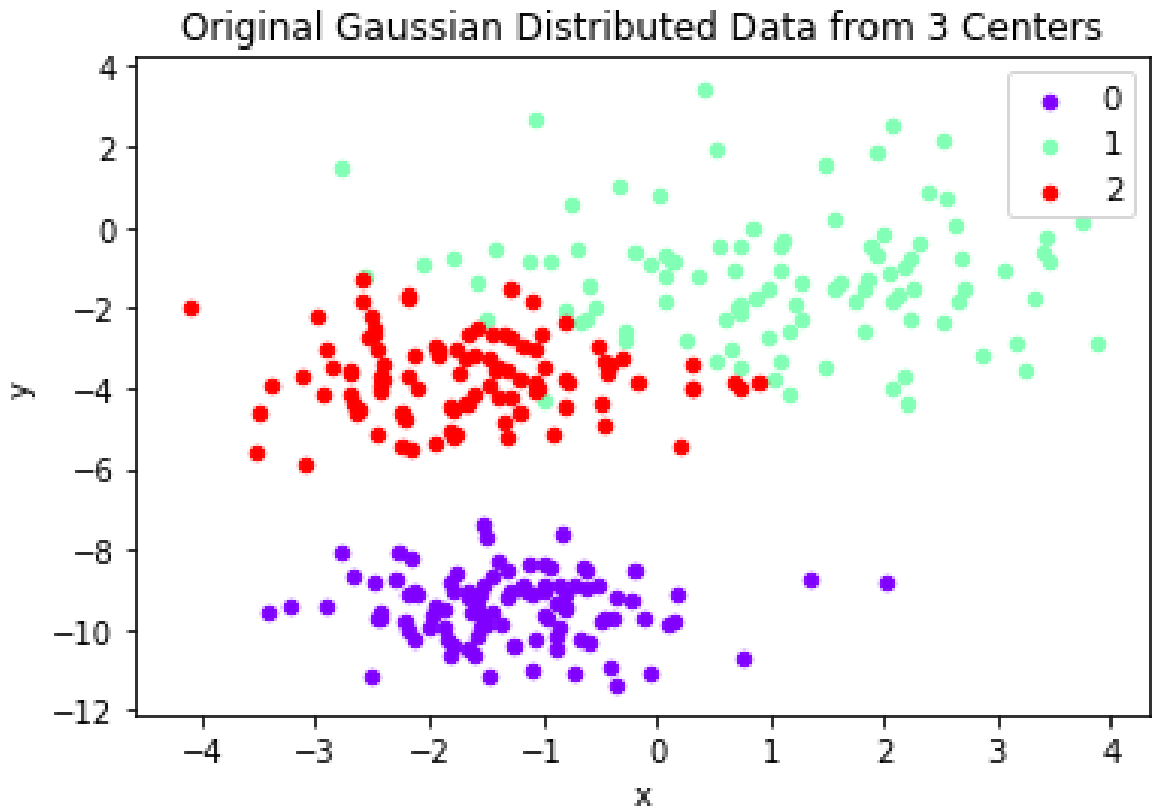


Figure: Generate Data from Mixture Guassian Distributions

Advantages of GMM over K-means

- K-means works well on simple and well-separated data, while GMM can handle overlapped clusters.
- In k-means, cluster must be circular, however, GMM's cluster can be non-circular. So, Gmm is more flexible in determining cluster shape.
- K-means uses a circle as a hard cutoff for cluster assignment. But, GMM measures the uncertainty or probabilities in cluster assignment.

Expectation-Maximization Introduction

- EM can estimate GMM's parameters by finding (local) maximum likelihood.
- Training with EM is an iterative process between conducting an expectation (E) step, which estimates the current parameters by computing the expectation of log-likelihood and a maximization (M) step, which calculates new parameters.
- GMM's log-likelihood math expression which should be maximized is:

$$l(\theta) = \sum_{i=1}^{n} log(\sum_{c=1}^{k} \pi_c N(x_i|\mu_c, \Sigma_c))$$

$$\text{where } \theta = \{\mu_1, ..., \mu_k, \Sigma_1, ..., \Sigma_k, \pi_1, ..., \pi_k\}$$

Initialization

- Assume k components over n data points
- Randomly assign cluster to each row of data
- pi: $\pi_j = \frac{|c_i=j|}{n}$
- mean_vector: $\mu_j = \frac{\sum(x_i|c_i=j)}{|c_i=j|}$
- covariance_matrixes: $\Sigma_j = \frac{\sum((x_i-\mu_j)^T \cdot (x_i-\mu_j)|c_i=j)}{|c_i=j|-1}$

Initialization Code

```
# assign a cluster to each row, make sure each cluster has data
data_comp = data_raw.zipWithIndex().map(lambda x: (np.array(x[0]), x[1] % n_components))
data_comp.cache()
```

```
PythonRDD[3294] at RDD at PythonRDD.scala:53
```

```
# Initialize pi with equal proportion
pi = [1/n_components for k in range(n_components)]
```

```
# Initial the mean_vector and covariance_matrixes matrix based on the formulas
mean_vector = []
covariance_matrixes = []
for  j in range(n_components):
    comp_j = data_comp.filter(lambda p: p[1] == j).map(lambda p: p[0])
    n_count = comp_j.count()

    mean_vector.append(comp_j.reduce(lambda x, y: x + y) / n_count)

    cov_sum = comp_j.map(lambda x: x - mean_vector[j])\
            .map(lambda x: np.multiply(np.reshape(x, (n_feature, 1)), x))\
            .reduce(lambda x, y: x + y)
    covariance_matrixes.append(cov_sum / (n_count-1))
```

Figure: Initialize GMM-EM

E - STEP

- Calculate a "soft" assignment of each row to each cluster, which is r matrix:

$$r_{ij} = \frac{\pi_j N(x_i|\mu_j, \Sigma_j)}{\sum_{c=1}^{k} \pi_c N(x_i|\mu_c, \Sigma_c)}$$

E - STEP Code

```
''' -------------------------   E - STEP   ------------------------- '''
# Calculating the r matrix, evrey row contains the probabilities
# for every cluster for this row

# r'shape (1, n_components) and each row's sum is 1
r = data_raw.map(lambda x: np.array([pi[j] * multivariate_normal.pdf(x, mean_vector[j], covariance_matrixes[j])\
                                     for j in range(n_components)]))\
             .map(lambda x: x / sum(x))

# Calculating the N, the sum of r_ic, when c = j
N = r.reduce(lambda x, y: x + y)
```

Figure: E - STEP Code

M - STEP

- Perform an MLE for each component and update μ_j, Σ_j, π_j.

$$\pi_j = \frac{\sum_{i=1}^{n} r_{ij}}{n}$$

$$\mu_j = \frac{\sum_{i=1}^{n} (r_{ij} \times x_i)}{\sum_{i=1}^{n} r_{ij}}$$

$$\Sigma_j = \frac{\sum_{i=1}^{n} r_{ij} \times ((x_i - \mu_j)^T \cdot (x_i - \mu_j))}{\sum_{i=1}^{n} r_{ij}}$$

M - STEP Code

```
''' -------------------------   M - STEP   ------------------------- '''
# Initializing the mean_vector
_mean_vector = np.zeros((n_components, n_feature))
# Initiating the covariance_matrixes
_covariance_matrixes = [np.zeros((n_feature, n_feature)) for _k in range(n_components)]

# r_data_raw's row is (r_i, X_i)
r_data_raw = r.zip(data_raw)
r_data_raw.cache()
for j in range(n_components):

    comp_j = r_data_raw.map(lambda x: x[0][j] * x[1])

    _mean_vector[j] = 1 / N[j] * comp_j.reduce(lambda x, y: x + y)

    cov_sum = r_data_raw.map(lambda x: (x[0], x[1] - mean_vector[j]))\
            .map(lambda x: x[0][j] * np.multiply(np.reshape(x[1], (n_feature, 1)), x[1]))\
            .reduce(lambda x, y: x + y)
    _covariance_matrixes[j] = 1 / N[j] * cov_sum

# Update for pi list
pi = [N[j]/n_rows for j in range(n_components)]
# Update for mean_vector
mean_vector = _mean_vector
# Update for covariance_matrixes
covariance_matrixes = _covariance_matrixes
```

Figure: M - STEP Code

Prediction

- For each data point, we calculate the probabilities of the each component and assign label with the maximal one

```
''' --------------------------   Prediction   -------------------------- '''
# predict the probabilities of the 3 clusters and assign cluster with the maximal one
y_pred = data_raw.map(lambda x: [multivariate_normal.pdf(x, mean_vector[j], covariance_matrixes[j])\
                                 for j in range(n_components)])\
        .map(lambda x: x.index(max(x))).collect()
```

Figure: Prediction Code

Prediction Visualization

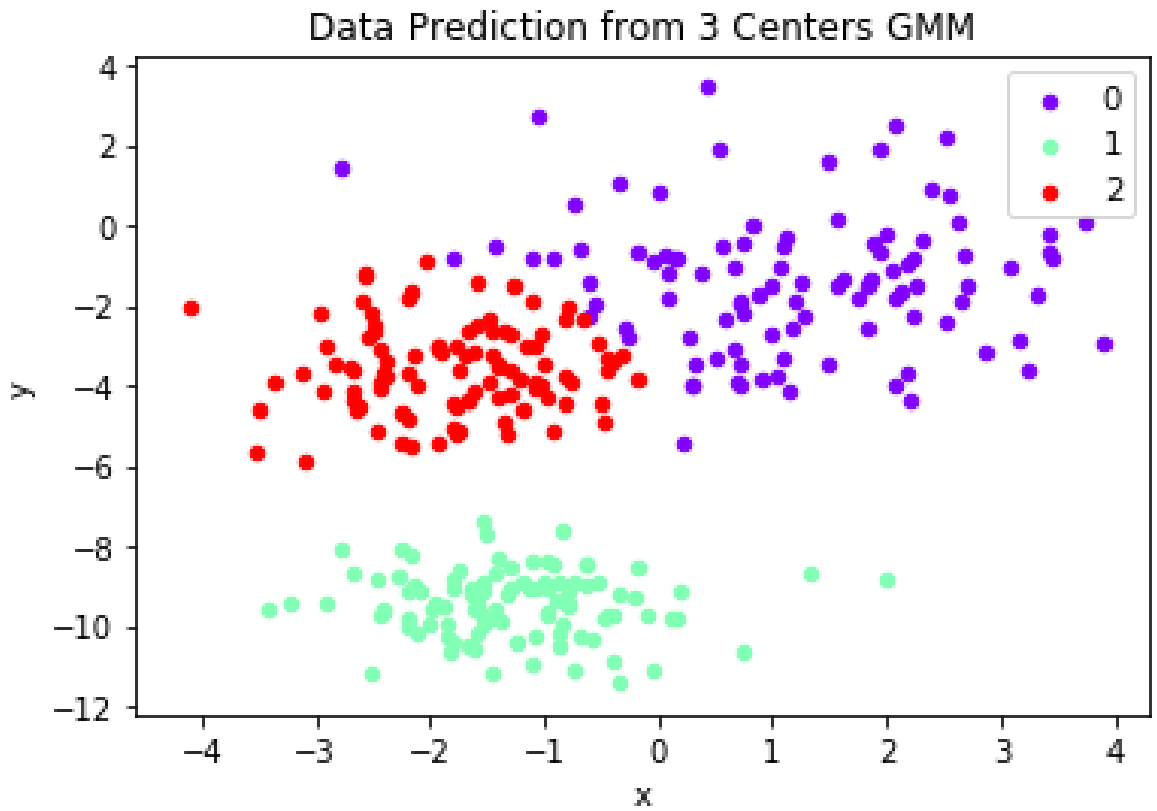


Figure: Prediction Visualization

GMM-EM Library Code

- from pyspark.ml.clustering import GaussianMixture

```
from pyspark.ml.clustering import GaussianMixture
from pyspark.ml.linalg import Vectors
```

```
spark = SparkSession \
    .builder \
    .appName("GMM-lib") \
    .getOrCreate()
```

```
# loads data
data_raw_v = data_raw.map(lambda x: (Vectors.dense(x.tolist()), ))
df = spark.createDataFrame(data_raw_v, ["features"])
```

```
gm = GaussianMixture(k=3, tol=0.0001, seed=10)
gm.setMaxIter(100)
```

```
GaussianMixture_47c7cbc0ebc6
```

```
model = gm.fit(df)
```

```
y_pred_lib = model.transform(df).select('prediction').rdd.flatMap(lambda x: x).collect()
```

Figure: GMM-EM Implementation from Library Code

Prediction Visualization

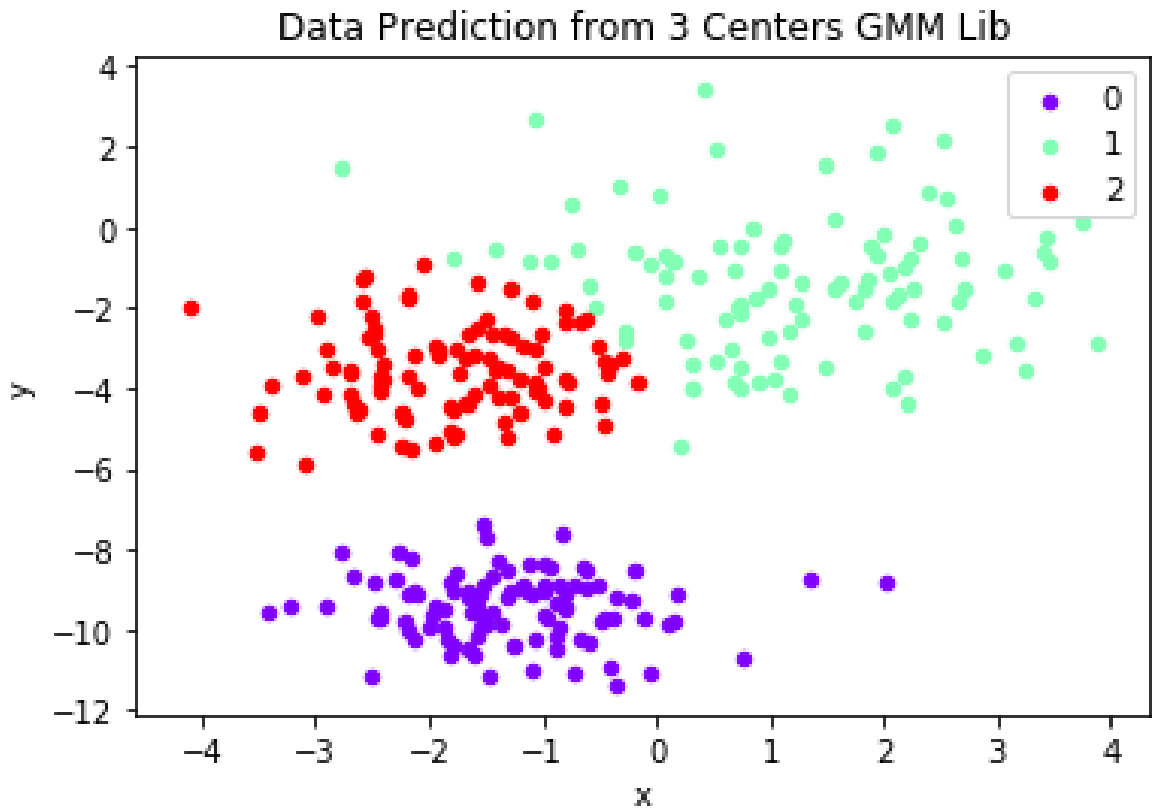


Figure: GMM-EM Library Implementation Prediction Visualization

GMM Log-likelihood

- In GMM library implementation, GMM's log-likelihood is maximized to -1200.238
- In implementation from scratch, GMM's log-likelihood is maximized to -1200.258. It is based on the formula below:

$$l(\theta) = \sum_{i=1}^{n} log(\sum_{c=1}^{k} \pi_c N(x_i|\mu_c, \Sigma_c))$$

where $\theta = \{\mu_1, ..., \mu_k, \Sigma_1, ..., \Sigma_k, \pi_1, ..., \pi_k\}$

```
llh = data_raw.map(lambda x: np.array([pi[j] * multivariate_normal.pdf(x, mean_vector[j], covariance_matrixes[j])
                                       for j in range(n_components)]))\
            .map(lambda x: np.log(sum(x)))\
            .reduce(lambda x, y: x + y)
```

Figure: GMM Log-likelihood Code

Clustering Accuracy

- Assume the trained cluster has almost the same features as the original one, so we can check if each data point is assigned to the same cluster by the model.
- Implementation from scratch accuracy: 95.3%
- Implementation from library accuracy: 95.0%

K-means

- from pyspark.ml.clustering import K-means

```
# loads data
data_raw_v = data_raw.map(lambda x: (Vectors.dense(x.tolist()), ))
df = spark.createDataFrame(data_raw_v, ["features"])
```

```
# Trains a k-means model.
kmeans = KMeans().setK(3).setSeed(1)
kmeans.setMaxIter(100)
model_k = kmeans.fit(df)
```

```
# Make predictions
y_pred_kmeans = model_k.transform(df).select('prediction').rdd.flatMap(lambda x: x).collect()
```

Figure: K-means Implementation from Library Code

K-means Prediction Visualization

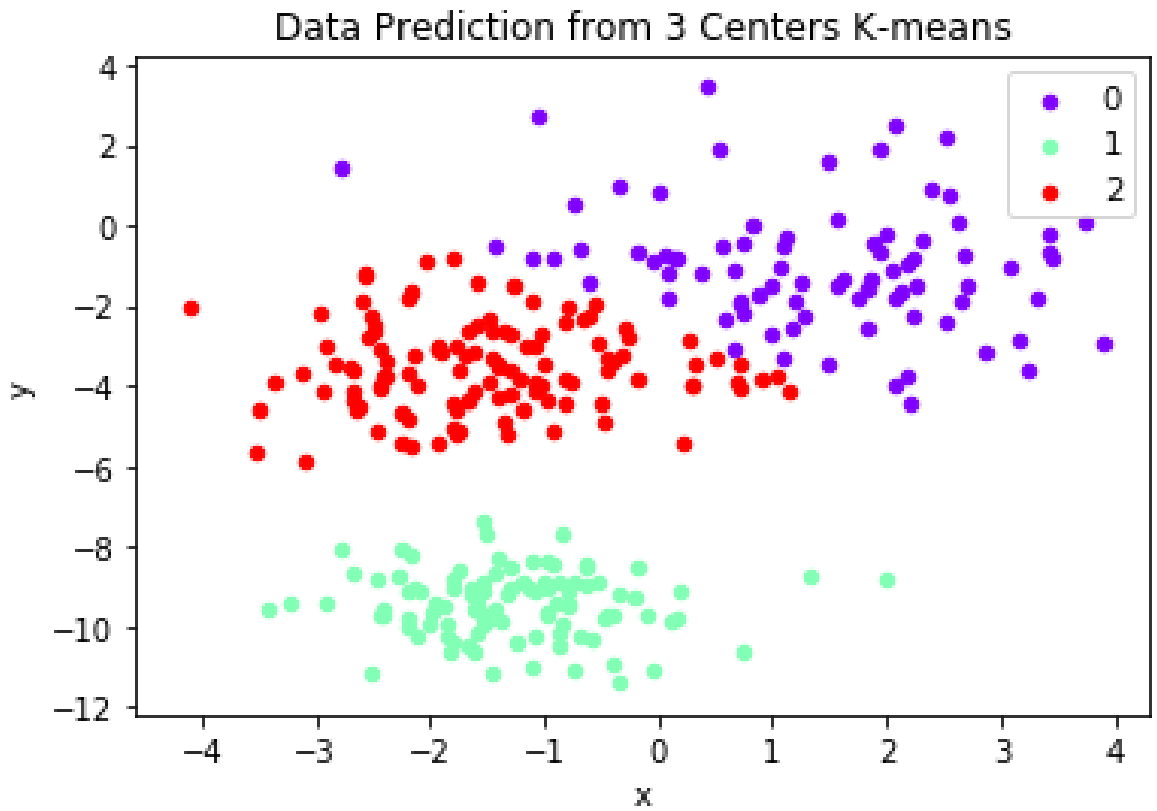


Figure: K-means Prediction Visualization

Visualization Comparison

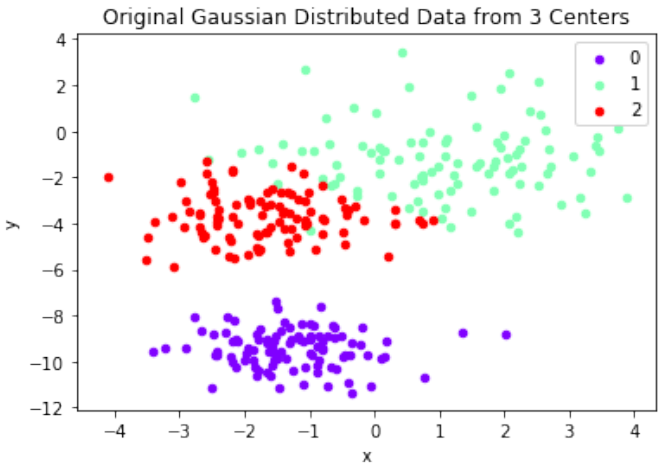


Figure: Original Data

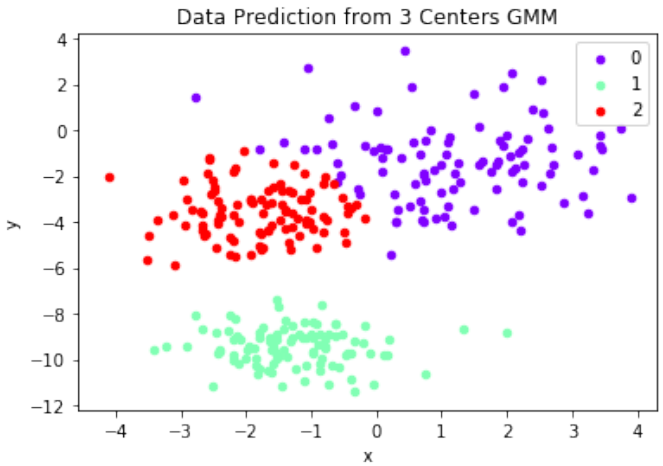
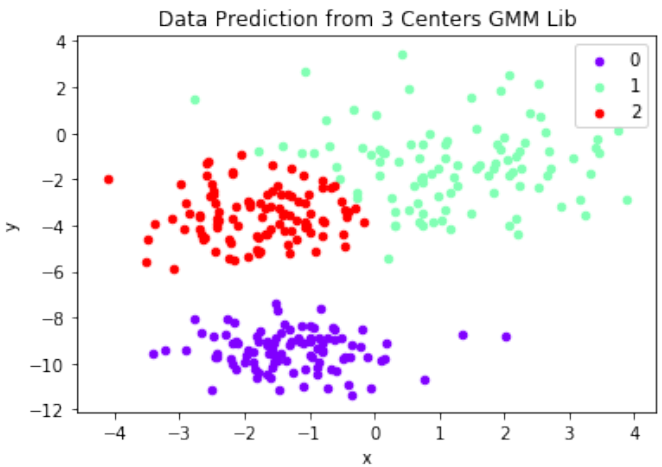


Figure: GMM from Scratch



gure: GMM Lib

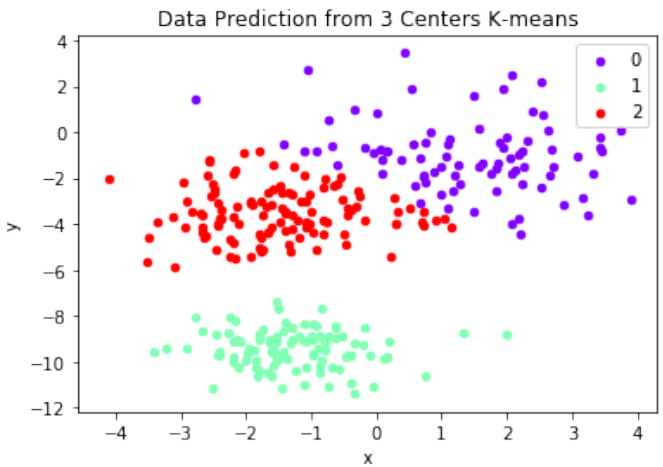


Figure: K-means

Accuracy Comparison

	GMM from scratch	GMM Lib	K-means
Accuracy	95.3%	95.0%	94.3%

Table: Clustering Accuracy Comparison

Model Extensibility

- The GMM-EM from scratch model can be extended to large-scale data because the huge r matrix of size (n, n_components) is never saved to local and only 3 small-size variables are saved in local, which are μ_j, Σ_j, π_j, and their sizes are shown below:
- μ_j size: (1, n_feature) * n_components
- Σ_j size: (n_feature, n_feature) * n_components
- π_j size: (1, n_components)
- However, if it is needed to visualize the predictions, we need to save y_pred, whose size is (1, n). If the amount of data is huge, it may cause problems.

Model Performance

- Compared with GMM-EM Lib, GMM-EM from scratch gets very close accuracy and log-likelihood, which indicates the correctness of this implementation.
- Compared with baseline K-means, the GMM model can predict with slightly higher accuracy, but in general the differences of their performances are very small.