



## 本科生实验报告

学生姓名： 丁晓琪

学生学号： 22336057

专业名称： 计科

### 一：PCA降低图像特征维度

(1). 算法流程

(2). 具体实现

### 二：Kmeans

(1). 算法流程(参数初始化方法):

(2). 实验结果

### 三：EM算法训练GMM模型

(1). 算法流程

(2). 实验结果

### 四：EM算法训练GMM VS Kmeans

## 一：PCA降低图像特征维度

由于样本的图像为784维，则需要对图像降维处理

### (1). 算法流程

- 对训练样本进行数据中心化,  $X_{\text{centered}} = X - \bar{X}$
- 求训练数据中心化后的协方差矩阵:  $C = \frac{1}{m} X_{\text{centered}}^T X_{\text{centered}}$
- 对该协方差矩阵计算特征值和特征向量
  - 选择主成分: 根据特征值的大小, 选择前  $n_{\text{components}}$  个最大的特征值对应的特征向量作为主成分。这些特征向量构成了一个转换矩阵  $W$ 。
- 数据转化: 将原始训练数据和原始测试数据投影到主成分空间上, 得到降维后的数据
$$X_{\text{reduced}} = X_{\text{centered}} W$$

### (2). 具体实现

```
1 class PCA():
2     def __init__(self):
3         self.mean=None
4         self.components=None
5
6     def fit(self,X,n_components):
```

```

7      # 功能：根据X计算n_components个主成分向量
8      m,n=X.shape
9      # 1 求协方差矩阵
10     self.mean=np.mean(X,axis=0)
11     X_centered=X-self.mean #会广播的
12     covariance_matrix=np.matmul(X_centered.T,X_centered)/m
13     # 2.计算特征值和特征向量（列向量）
14     eigenvalues,eigenvectors=np.linalg.eig(covariance_matrix)
15     # 3.取出最大的前n个特征向量
16     #将特征向量最大到小的索引找出来
17     index=np.argsort(eigenvalues)[::-1]
18     #排序
19     eigenvectors=eigenvectors[:,index]
20     # 选择前n个特征向量
21     self.components=eigenvectors[:, :n_components]
22
23     def transform(self,X):
24         # 功能：将向量转到主成分空间
25         return np.dot(X-self.mean,self.components)

```

zh

## 二：Kmeans

### (1).算法流程(参数初始化方法):

1. 初始化阶段：选择k个初始聚类中心（如果要划分为k个聚类）

聚类中心的初始化：

- 法一：随机选择训练样本内的k个顶点为k个聚类的初始化中心

```

1      def init_center_random(self):
2          # 功能：随机从训练实例中挑选10个样本初始化聚类中心
3          init_num = np.random.choice(self.train_images.shape[0],
self.k, replace=False)
4          self.centers= self.train_images[init_num]

```

- 法二：

- 思想：依据距离初始化聚类中心，使得聚类中心尽可能分散。
- 实现：先随机从样本中挑选出一个点为第一个聚类中心，然后在样本中选出离当前已存在聚类中心最远的样本为新的聚类中心，直到存在k个聚类中心为止

```

1      def init_center_distance(self):
2          # 功能：根据距离初始化聚类中心
3          # 1. 选择第一个中心点为随机样本
4          self.centers = np.zeros((self.k,
self.train_images.shape[1]))
5          self.centers[0] =
self.train_images[np.random.choice(self.train_images.shape[0])]
6          # 2. 计算所有点到第一个中心点的距离
7          distances = np.linalg.norm(self.train_images -
self.centers[0], axis=1)
8          # 3. 选择剩余的中心点
9          for i in range(1, self.k):

```

```

10         # 选择与已有中心点距离最远的点作为新的中心点
11         farthest_index = np.argmax(distances)
12         self.centers[i] = self.train_images[farthest_index]
13         # 更新距离, 考虑新中心点
14         new_distances = np.linalg.norm(self.train_images -
15 self.centers[i], axis=1)
16         # 更新到新中心点的距离
17         distances = np.minimum(distances, new_distances)

```

## 2. 迭代阶段:

在 `Kmeans.train()` 中实现:

- 计算数据集中每个点到K个聚类中心的距离, 使用欧氏距离作为度量标准

```

1     def compute_closest_index(self, sample):
2         # 功能: 计算并返回样本sample距离(欧几里得距离)最近的聚类中心的索引
3         # 输入:
4         #     sample: 要计算的样本
5         distances = np.sum((self.centers - sample) ** 2, axis=1)
6         closest_center_index = np.argmin(distances)
7         return closest_center_index

```

将每个点分配给距离其最近的聚类中心, 形成K个聚类

```

1         # 1. 对样本判断所属簇
2         # 1.1 创建数组来存储每个样本的簇分配
3         cluster_assignments =
4 np.zeros(self.train_images.shape[0], dtype=int)
5         for i in range(0, self.train_images.shape[0]):
6             # 计算样本到每个聚类中心的距离
7             # 找到最近的聚类中心的索引
8
9         closest_index = self.compute_closest_index(self.train_images[i])
10        cluster_assignments[i] = closest_index

```

- 对于每个聚类, 重新计算其聚类中心, 即计算该聚类中所有点的均值作为新的聚类中心(聚类内所有点在每个维度上的均值是该聚类新的中心的取值)

```

1         # 2: 更新聚类中心
2         new_centers = np.zeros((self.k,
3 self.train_images.shape[1]))
4         for i in range(0, self.k):
5             # 找到第i簇的所有样本
6
7         cluster_i_sample = self.train_images[cluster_assignments == i]
8         # 簇不为空时, 更新聚类中心
9         if cluster_i_sample.size > 0:
10            new_centers[i] = cluster_i_sample.mean(axis=0)
11        self.centers = new_centers
12        self.cluster_assignment = cluster_assignments

```

## 3. 测试阶段:

训练出k个聚类中心后，聚类中心在 `self.center` 中的索引并不代表聚类的标签。由于训练的样本是有标签的数据，则这里取最后一次训练过程中被分到聚类内的训练样本的标签众数为该聚类的标签。（训练样本的被分配到的聚类中心的索引存储在 `self.cluster_assignment` 中）

```
1 def test(self, test_labels, test_images):
2     # 功能：用test_images样本和它对应的标签test_labels测试模型，计算模型正确
    率
3     # 输入：
4     # test_images: 测试样本
5     # test_labels: 测试样本的标签
6     correct_sum=0
7     for i in range(0, test_images.shape[0]):
8         # 1. 计算样本到每个聚类中心的距离
9         # 找到最近的聚类中心的索引
10        closest_index=self.compute_closest_index(test_images[i])
11        # 2. 找该聚类的标签（聚类内样本标签的众数）
12        cluster_labels =
self.train_labels[self.cluster_assignment == closest_index]
13        mode_result = mode(cluster_labels)
14        predict_label = mode_result.mode[0]
15        # 3. 比较聚类标签和样本真实标签
16        if predict_label==test_labels[i]:
17            correct_sum+=1
18        accuracy = correct_sum / test_images.shape[0]
19        self.test accuracies.append(accuracy)
```

## (2). 实验结果

- 实验结果获取：
  - 分别采用两种初始化方法初始化参数，并且训练聚类，每次训练都迭代50次
  - 在模型训练过程中：每隔5次迭代就会用 `test_image` 训练集测试聚类的正确性
  - 记录从聚类参数初始化到训练聚类结束的总训练时间（包括训练过程中测试聚类正确性的时间）

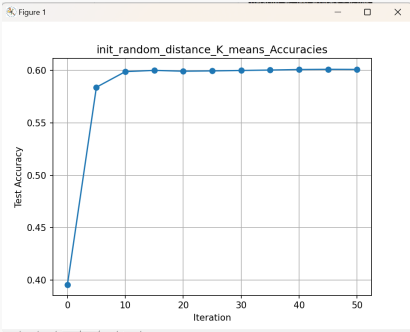
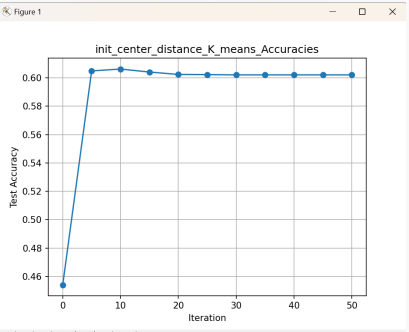
```
1 kmeans_random=kmeans(10,50,train_labels=train_labels,train_images=t
rain_images)
2 start_time = time.time()
3 kmeans_random.init_center_distance()
4
5 kmeans_random.train(test_images=test_images,test_labels=test_labels
)
6 end_time = time.time()
7 training_duration = end_time - start_time
8 print(f"训练过程耗时 {training_duration:.2f} 秒")
9 print(f"final test accruacy:
{kmeans_random.test(test_labels=test_labels,test_images=test_images)
}")
10
11 kmeans_random.plot accuracies("init_random_distance_K_means_Accurac
ies")
```

```

12 kmeans_center=kmeans(10,50,train_labels=train_labels,train_images=t
rain_images)
13 start_time = time.time()
14 kmeans_center.init_center_distance()
15
16 kmeans_center.train(test_images=test_images,test_labels=test_labels
)
17 end_time = time.time()
18 training_duration = end_time - start_time
19 print(f"训练过程耗时 {training_duration:.2f} 秒")
20
21 kmeans_center.test(test_labels=test_labels,test_images=test_images)
print(f"final test accruacy:
{kmeans_random.test(test_labels=test_labels,test_images=test_images)
}")
22
kmeans_center.plot_accuracies("init_center_distance_K_means_Accurac
ies")

```

• 实验结果：

	随机初始化聚类中心	根据距离初始化聚类中心
训练总时长	训练过程耗时 66.91 秒	训练过程耗时 69.95 秒
训练过程中的正确率	 <pre> Iteration 0: Test Accuracy = 0.3956 Iteration 5: Test Accuracy = 0.584 Iteration 10: Test Accuracy = 0.5989 Iteration 15: Test Accuracy = 0.6001 Iteration 20: Test Accuracy = 0.5993 Iteration 25: Test Accuracy = 0.5996 Iteration 30: Test Accuracy = 0.6 Iteration 35: Test Accuracy = 0.6004 Iteration 40: Test Accuracy = 0.6009 Iteration 45: Test Accuracy = 0.6011 </pre>	 <pre> Iteration 0: Test Accuracy = 0.454 Iteration 5: Test Accuracy = 0.6049 Iteration 10: Test Accuracy = 0.6062 Iteration 15: Test Accuracy = 0.6041 Iteration 20: Test Accuracy = 0.6024 Iteration 25: Test Accuracy = 0.6023 Iteration 30: Test Accuracy = 0.6021 Iteration 35: Test Accuracy = 0.6021 Iteration 40: Test Accuracy = 0.6021 Iteration 45: Test Accuracy = 0.6021 </pre>
训练完的聚类对测试集的正确率	60.01% final test accruacy: 0.601	60.01% final test accruacy: 0.601

• 实验结果分析：

- 训练总时长：随机初始化聚类中心比根据距离初始化聚类中心短。

原因：根据距离初始化聚类中心在初始化阶段需要迭代对每个样本进行和聚类中心的距离计算，则耗时较长

- 正确率：

- 训练开始时的正确率：根据距离初始化比随机初始化的正确率高  
原因：根据距离初始化的初始聚类中心尽可能分散，可能更接近真实样本的分类
- 训练过程的正确率：根据距离初始化在第10次迭代收敛达到最高值60.62%比随机初始化的第45次迭代60.11%更快且正确率更高。根据距离初始化在15次迭代左右开始会由于迭代次数过多出现过拟合导致正确率下降  
原因：根据距离初始化的初始聚类中心尽可能分散，可能更接近真实样本的分类
- 最终正确率：两个初始化方法的正确率相近甚至相同

### 三：EM算法训练GMM模型

#### (1).算法流程

GMM模型：  $p(x|\theta) = \sum_{k=1}^K \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$

- GMM包含K个参数模型的概率密度函数，这些函数的加权组合构成了GMM的概率密度函数。
- 在GMM中，每个高斯分布都代表了一个聚类，而整个模型则能够描述数据的多峰分布特性。
- GMM引入了隐变量 $z$ ，用于表示数据属于哪一个聚类。
- 计算样本在高斯分布中的概率

```
1 mvn = multivariate_normal(self.means[index],
2 self.covariance_matrices[index])
3 return mvn.pdf(sample)
```

##### 1. 初始化：

- 高斯分布的数量：指定为10
- 每个高斯分布的均值：同上面的kmeans获取10个聚类中心的两种初始化方法：随机初始化和根据距离初始化

```
1 # 2. 初始化每个高斯分布的均值
2 # 随机抽取k个样本初始化均值
3 if(means_init_method==0):
4     init_num = np.random.choice(self.train_images.shape[0],
5 self.k, replace=False)
6     self.means= self.train_images[init_num]
7     # 根据距离来选择哪个样本作为高斯分布的初始化均值
8 elif(means_init_method==1):
9     # 选择第一个平均值为随机样本
10    self.means = np.zeros((self.k,
11 self.train_images.shape[1]))
12    self.means[0] =
13 self.train_images[np.random.choice(self.train_images.shape[0])]
14    # 计算所有点到第一个中心点的距离
15    distances = np.linalg.norm(self.train_images -
16 self.means[0], axis=1)
17    # 选择剩余的中心点
18    for i in range(1, self.k):
19        # 选择与已有中心点距离最远的点作为新的中心点
20        farthest_index = np.argmax(distances)
21        self.means[i] = self.train_images[farthest_index]
22    # 更新距离，考虑新中心点
```

```

19         new_distances = np.linalg.norm(self.train_images -
self.means[i], axis=1)
20         # 更新到新中心点的距离
21         distances = np.minimum(distances, new_distances)

```

- 每个高斯分布的权重：随机生成整数并归一化的方法

```

1         random_weights = np.random.randint(0, 101, size=10)
2         weights_sum = np.sum(random_weights.astype(float))
3         self.weights = random_weights.astype(float) / weights_sum

```

- 每个高斯分布的协方差矩阵：

- 法一：使用样本的协方差矩阵的对角矩阵进行初始化

```

1         # 3. 初始化每个高斯分布是协方差矩阵
2         # 要初始化为样本的协方差矩阵的对角矩阵(对角元素不同)
3         if(cov_init_method==0):
4             cov = np.cov(self.train_images, rowvar=False) + 1e-6
* np.eye(self.train_images.shape[1])
5             cov = np.diag(np.diag(cov))
6             self.covariance_matrices = cov[np.newaxis,
:].repeat(self.k, axis=0)

```

- 法二：始化为一个所有对角元素相等的对角矩阵。

```

1         # 3. 初始化每个高斯分布是协方差矩阵
2         # 初始化对角矩阵元素都相等，球形初始化
3         avg_var = np.mean(np.var(self.train_images, axis=0))
4         cov = avg_var * np.ones((self.train_images.shape[1],
self.train_images.shape[1]))
5         # 由于cov需要是对角矩阵，这里仍使用对角矩阵形式
6         cov = np.diag(np.full(self.train_images.shape[1],
avg_var))
7         self.covariance_matrices = cov[np.newaxis,
:].repeat(self.k, axis=0)

```

## 2. EM算法迭代：

每次迭代交替做E-M步

- E步：

- 通过计算每个样本在每个高斯分布下的后验概率（已知样本 $x$ ， $x$ 在第 $i$ 个高斯分布中出现的概率）来计算落在第 $k$ 个高斯分布的概率

$$\gamma_{jk} = E_{p(z|x;\theta^{(t)}[z_k])} = \frac{\alpha_k N(x|\mu_k, \Sigma_k)}{\sum_{l=1}^K \alpha_l N(x|\mu_l, \Sigma_l)} = p(z = 1_k|x;\theta^{(t)})$$

其中， $N(x_j|\mu_k, \Sigma_k)$ 是样本  $x_j$  在第  $k$  个高斯分布下的概率密度函数。

```

1      # 1.计算gamma
2
3      gamma_matrix=np.zeros((self.train_images.shape[0],self.k))
4      for j in range(0,self.k):
5          # 一次性计算所有样本的概率密度
6          gamma_matrix[:, j] =
7      self.Gaussian_probability(j,self.train_images) * self.weights[j]
8      #print(gamma_matrix)
9      gamma_matrix/=gamma_matrix.sum(axis=1,keepdims=True) #按行
10     求和保持维度

```

o M步: N为样本特征值维度

- 根据E步得到的 $\gamma$ 矩阵, 更新高斯分布的参数(权重、均值、协方差矩阵)。
- 计算  $N_k$ :  $N_k = \sum_{j=1}^N \gamma_{jk}$

```

1  N=np.sum(gamma_matrix,axis=0) #按行求和

```

- 更新权重  $\alpha_k$ :  $\alpha_k = \frac{N_k}{N}$

```

1  self.weights[i]=N[i]/self.train_images.shape[0]

```

- 更新每个高斯分布的均值  $\mu_k$ :  $\mu_k = \frac{\sum_{j=1}^N \gamma_{jk} x_j}{N_k}$

```

1  self.means[i]=
2  (1/N[i])*np.dot(gamma_matrix[:,i].reshape(1,-1),self.train_images
3  )

```

- 更新协方差矩阵  $\Sigma_k$ :

$$\Sigma_k = \frac{\sum_{j=1}^N \gamma_{jk} (x_j - \mu_k)^T (x_j - \mu_k)}{N_k}$$

```

1  covariance_matrix = (1 / (N[i] + 1e-6)) * np.dot((gamma_matrix[:,
2  i].reshape(-1, 1) * diff).T, diff)
3  self.covariance_matrices[i]=covariance_matrix

```

3. 测试阶段: 同k\_means中的测试方法, 用最后一次训练的分到各个高斯分布的训练样本的标签众数为该高斯分布的预测标签。计算测试样本在各个高斯分布下的概率, 取最大概率的高斯分布的所属标签为测试样本的预测标签

```

1  def test(self,test_images,test_labels):
2      # 功能: 已知x求属于哪个聚类z
3      correct_sum=0
4      gamma_vector=np.zeros((test_images.shape[0],self.k))
5      for j in range(0,self.k):
6          # 计算每个样本在第j个高斯分布下的概率
7          gamma_vector[:, j] = self.Gaussian_probability(j,
8          test_images) * self.weights[j]

```



```

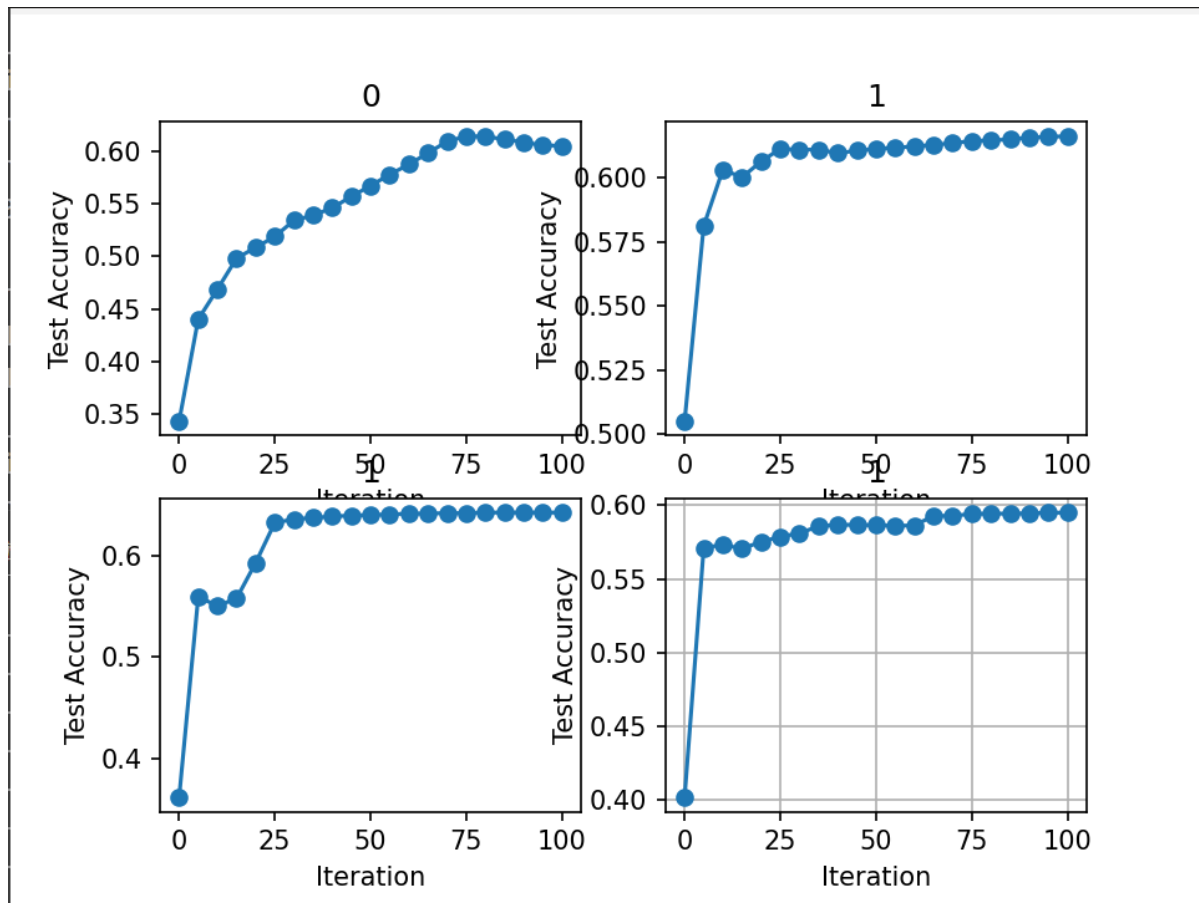
9      gamma_vector /= np.sum(gamma_vector, axis=1, keepdims=True)
10     predict_test_indexes=np.argmax(gamma_vector,axis=1)
11     for i in range(0,test_images.shape[0]):
12         cluster_labels = self.train_labels[self.predict_indexes ==
predict_test_indexes[i]]
13         mode_result = mode(cluster_labels)
14         predict_label = mode_result.mode[0]
15         #print(i)
16         if predict_label == test_labels[i]:
17             correct_sum += 1
18
19     accuracy = correct_sum / test_images.shape[0]
20     self.test_accuracies.append(accuracy)
21     return accuracy

```

## (2). 实验结果

对上述提到的不同初始化方法分别训练迭代100次。

在训练过程中每隔5次迭代用测试集测试模型的正确率。



	随机初始化高斯分布的均值	根据距离初始化高斯分布的均值
以训练样本的协方差矩阵的对角矩阵为高斯分布的协方差矩阵	<pre>Iteration 0: Test Accuracy = 0.3435 Iteration 5: Test Accuracy = 0.4483 Iteration 10: Test Accuracy = 0.4687 Iteration 15: Test Accuracy = 0.4981 Iteration 20: Test Accuracy = 0.5083 Iteration 25: Test Accuracy = 0.5187 Iteration 30: Test Accuracy = 0.534 Iteration 35: Test Accuracy = 0.5388 Iteration 40: Test Accuracy = 0.5459 Iteration 45: Test Accuracy = 0.5567 Iteration 50: Test Accuracy = 0.5669 Iteration 55: Test Accuracy = 0.5775 Iteration 60: Test Accuracy = 0.5921 Iteration 65: Test Accuracy = 0.5979 Iteration 70: Test Accuracy = 0.6089 Iteration 75: Test Accuracy = 0.6138 Iteration 80: Test Accuracy = 0.6136 Iteration 85: Test Accuracy = 0.6189 Iteration 90: Test Accuracy = 0.6088 Iteration 95: Test Accuracy = 0.6054 训练过程耗时 189.90 秒 final test accuracy: 0.6046</pre> <p>收敛速度：最慢</p>	<pre>Iteration 0: Test Accuracy = 0.362 Iteration 5: Test Accuracy = 0.5594 Iteration 10: Test Accuracy = 0.5504 Iteration 15: Test Accuracy = 0.5577 Iteration 20: Test Accuracy = 0.5928 Iteration 25: Test Accuracy = 0.6322 Iteration 30: Test Accuracy = 0.6355 Iteration 35: Test Accuracy = 0.637 Iteration 40: Test Accuracy = 0.6382 Iteration 45: Test Accuracy = 0.6393 Iteration 50: Test Accuracy = 0.6397 Iteration 55: Test Accuracy = 0.6404 Iteration 60: Test Accuracy = 0.641 Iteration 65: Test Accuracy = 0.6412 Iteration 70: Test Accuracy = 0.6416 Iteration 75: Test Accuracy = 0.6416 Iteration 80: Test Accuracy = 0.6421 Iteration 85: Test Accuracy = 0.6421 Iteration 90: Test Accuracy = 0.6421 Iteration 95: Test Accuracy = 0.6421 训练过程耗时 194.35 秒</pre> <p>训练时间最长：计算训练样本呢的协方差矩阵和计算每个样本到聚类的距离花费的时间最长 正确率：最高</p>
以训练样本某个维度的平均值为对角矩阵的元素的对角矩阵为高斯分布的协方差矩阵	<pre>Iteration 0: Test Accuracy = 0.509 Iteration 5: Test Accuracy = 0.5012 Iteration 10: Test Accuracy = 0.6031 Iteration 15: Test Accuracy = 0.6001 Iteration 20: Test Accuracy = 0.6063 Iteration 25: Test Accuracy = 0.6112 Iteration 30: Test Accuracy = 0.6109 Iteration 35: Test Accuracy = 0.6109 Iteration 40: Test Accuracy = 0.6095 Iteration 45: Test Accuracy = 0.6105 Iteration 50: Test Accuracy = 0.6112 Iteration 55: Test Accuracy = 0.6117 Iteration 60: Test Accuracy = 0.6121 Iteration 65: Test Accuracy = 0.6128 Iteration 70: Test Accuracy = 0.6134 Iteration 75: Test Accuracy = 0.6143 Iteration 80: Test Accuracy = 0.6145 Iteration 85: Test Accuracy = 0.6149 Iteration 90: Test Accuracy = 0.6156 Iteration 95: Test Accuracy = 0.6159 训练过程耗时 198.85 秒 final test accuracy: 0.6161</pre> <p>最快收敛</p>	<pre>Iteration 0: Test Accuracy = 0.4016 Iteration 5: Test Accuracy = 0.5712 Iteration 10: Test Accuracy = 0.5732 Iteration 15: Test Accuracy = 0.5707 Iteration 20: Test Accuracy = 0.5752 Iteration 25: Test Accuracy = 0.5787 Iteration 30: Test Accuracy = 0.5811 Iteration 35: Test Accuracy = 0.5858 Iteration 40: Test Accuracy = 0.5869 Iteration 45: Test Accuracy = 0.5867 Iteration 50: Test Accuracy = 0.5866 Iteration 55: Test Accuracy = 0.5861 Iteration 60: Test Accuracy = 0.5864 Iteration 65: Test Accuracy = 0.5928 Iteration 70: Test Accuracy = 0.593 Iteration 75: Test Accuracy = 0.5941 Iteration 80: Test Accuracy = 0.5945 Iteration 85: Test Accuracy = 0.5943 Iteration 90: Test Accuracy = 0.5947 训练过程耗时 187.09 秒 final test accuracy: 0.5951</pre> <p>正确率：最低，且比kmeans略低</p>

## 四：EM算法训练GMM VS Kmeans

- 实现：K\_means实现简单，主要需要调整的参数仅为簇的数量K。EM算法训练GMM实现较为复杂，参数包含均值、协方差和混合系数
- 效率：
  - K\_means计算复杂度低，只需要计算样本间的距离，求和，取最大最小值等简单计算，所以训练速度快。
  - EM训练GMM聚类计算复杂度高：需要计算协方差矩阵，求特征值特征向量等复杂计算，训练速度低
- 收敛速度：K\_means的收敛速度快，50次之内收敛。EM训练GMM的收敛速度慢，50次内无法收敛
- 正确率：在本次实验中，EM算法训练GMM模型在测试集达到的最终模型正确率更高，
  - GMM模型能够处理簇形状不规则、大小不一的数据集。
  - 每个高斯分布的均值和协方差矩阵决定了对应聚类的位置和形状，因此GMM模型能够灵活地适应不同形状的聚类。

