K-mean 实现点集合分类

1.实验环境：

- OS: Windows 10
- CPU: intel CORE i5
- Python 3.6
- Sckikit_learn 0.18.1
- 其他 Python 科学计算模块
- 编译器：PythonCharm 2017.3.3

2.主要参考文档：

- Sklearn 0.18 官方文档

3.实验过程：

3.1.实验问题描述

K-mean 算法是思想较为简单的分类算法，本实验主要是实现一个简单的二维坐标点集合分类，并且，运行比较 Sklearn 中的聚类手写数字示例与小批量 K-mean 示例，比较其中的异同，得到实验结果与结论。

3.2.实验资源库：

这个实验参考了 sklearn 中的 clustering 部分官方文档，并且运用到了其中的部分库资源

网址：http://scikit-learn.org/stable/modules/clustering.html#clustering

3.3 实验原理与过程：

本实验参考了 sklenarn 0.18 版本的官方文档，首先说明一下 K-mean 算法的原理：

非监督学习的特点是训练数据中只有输入数据而没有输入对应的期望输出。因此其学习对象是输入数据中隐含的规律。

聚类是指根据数据之间的相似性，将数据集合分成若干个子集的问题。其中每个子集内的数据相似，不同集合之间的数据应不相似，体现相似数据被聚合在一起的要求。这些相似数据聚合在一起形成的数据子集通常被称为簇。

在 K-mean 方法中的相似性使用数据之间的距离来衡量，相互之间距离近的数据聚合为一个簇，是一种典型的划分聚类方法。

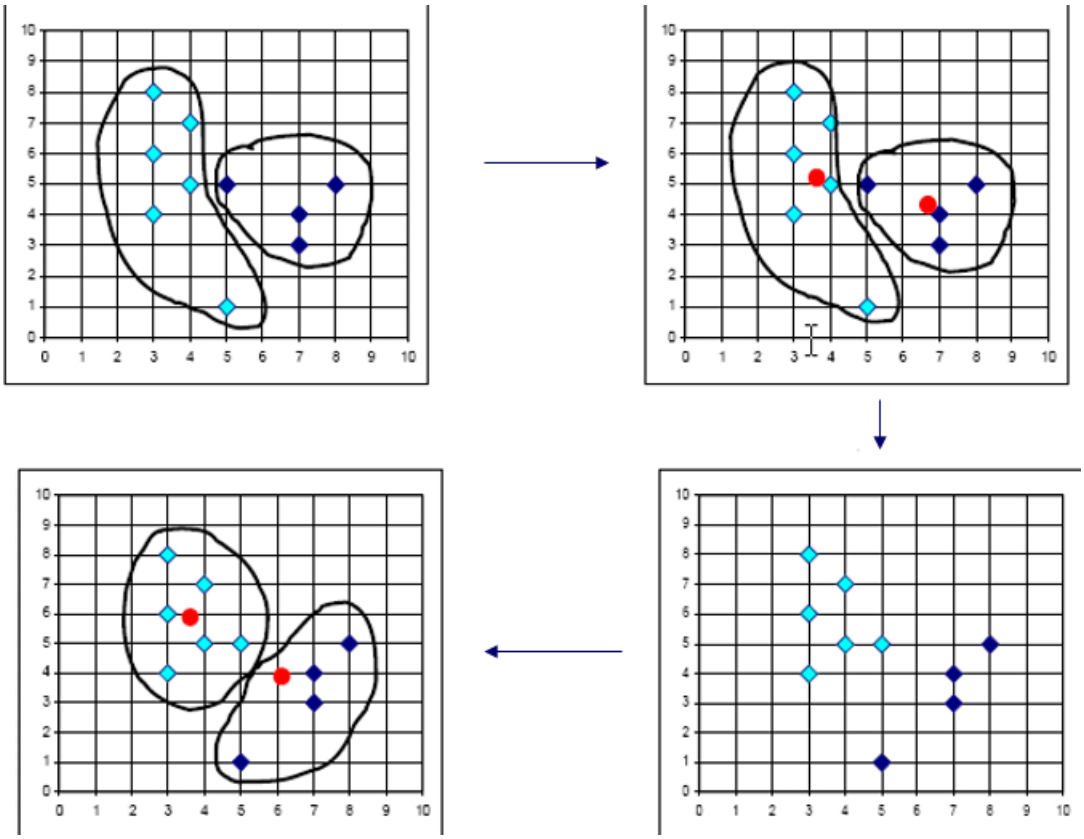
设数据个数和聚类个数分别为 N 和 k，$x_i|_{i=1}^{N}$表示第 i 个数据，v=$\{v_1, v_2, \cdots, v_k\}$表示所有簇的均值，u =$\{u_{ij}\}$， i=1,2, …N; j=1,2, …K,表示各个数据到各个簇的明确的隶属度，即每个 uij 为 1 或 0，则 k-均值聚类问题是找到最优的$\{\boldsymbol{u}, \boldsymbol{v}\}^*$,达到以下的目标：

$$\min_{u,v} \sum_{i=1}^{N} \sum_{j=1}^{k} U_{ij} D(x_i, v_j)$$

$$\text{subject to } \sum_{j=1}^{k} u_{ij} = 1, i = 1,2,\cdots,N$$

其中$D(x_i, v_j)$表示两个数据之间的距离，目前最为常用的是欧式距离；约束条件表示每个数据最多只能属于一个簇。

为达到这种优化目标，k-均值算法在更新 u(数据归属)与归属 v(簇均值)的两个过程之间进行交替运算，达到结果稳定，在老师的课件中给出了下面的图片非常直观得说明了问题。



算法流程结合上图：

- 从数据集中随机选择 k 个数据，作为初始的簇均值。
- 对剩余的每个数据，根据其与各个簇之间的距离，将它划分给最近的簇。
- 针对更新后的每个簇，重新计算其均值。
- 重复第二步与第三步，直到聚类结果不再发生变化或者到达最大的迭代次数。

在具体的实验过程中，我设置了两种情况，指定需要聚类的点集合；由在一定范围内随机生成的点集合进行指定的聚类，都需要事先指定分类的个数。

```
1.  def __init__(self, dataset=None, centroid=None):
2.      '''''
3.
```

```
    :param dataset: 传入的点的数据集合
    :param centroid:  指定的初始的均值点的个数
    '''
    self.dataset = dataset
    self.centroid = centroid


@classmethod
def get_centroid_numbers(self):
    '''''
    分类的类数
    :return:
    '''
    return 8


@classmethod
def get_random_points(self, numbers, begin=1, end=10):
    '''''
    生成了随机的点集合
    :param numbers: 生成的点集合的数目
    :param begin: 范围下限
    :param end: 范围上限
    :return:
    '''
    dataset = [[random.uniform(begin, end), random.uniform(begin, end)] for i in range(numbers)]
    dataset = [[float("{0:.2f}".format(data)) for data in point] for point in dataset]
    return dataset
```

在本实验中，对于初始的均值点的处理方法较为简单，直接根据数据集合和给出的初始的均值点的数量，随机的在数据的范围内生成初始点和初始的均值点，然后进行迭代聚类：

```
@classmethod
    def get_clusters_points(self, dataset, centroids):
        '''''
        将所有的点进行分类，分为 K 类
        :param dataset: 数据集合
        :param centroids: 均值点集合
        :return:
        '''
        clusters_points = []
        for i in range(len(centroids)):
```

```
            clusters_points.append([])
        for point in dataset:
            Distances = [self.get_Distance(point, centroid) for centroid in c
entroids]
            belonging_cluster_id = Distances.index(min(Distances))
        # 进行点的分类划分
            clusters_points[belonging_cluster_id].append(point)
        return clusters_points
```

迭代聚类的方式如上图所示，每次聚类完成之后，将当前的聚类的均值中心点的集合 mu 赋值给 centroids，然后再次将数据集合 dataset 和 centroids 传入函数中，将数据集合中的每一个点与均值点集合中的每一个点比较距离，并且将这个点归类到离其最近的均值点对应的集合中去。

接下来就是整个迭代过程什么时候停止。

在这个实验中，我设定的停止条件为上一次聚类结果和本次聚类结果没有偏差，则算法结束。

```
@classmethod
def converged(self, old_mu, new_mu):
    for (i, k) in zip(old_mu, new_mu):
        for (x, y) in zip(i, k):
            if x != y: return False
    return True
```

整个实验需要返回聚类的结果，和每个聚类的均值点的集合，返回均值点集合的函数如下：

```
@classmethod
    def move_centroids(self, clusters_points):
        '''''
        得到了均值点的集合
        :param clusters_points: 均值点集合
        :return:
        '''
        mu = []
        for points in clusters_points:
            mu.append(numpy.mean(points, axis=0))
        return mu
```

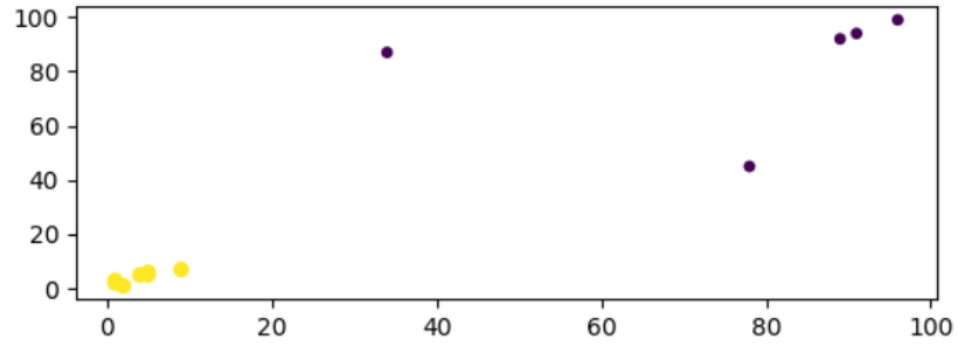
3.4 实验结果与分析

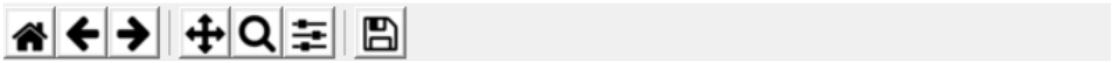
在输入的数据如图所示，分为两类时：

```
points = [[1, 2], [2, 1], [3, 1], [5, 4], [5, 5], [6, 5], [7, 9], [99, 96],
[94, 91], [92, 89],[87, 34],[45, 78]]
```

```
2.  algo =Kmean.Kmean(points, 2)
```

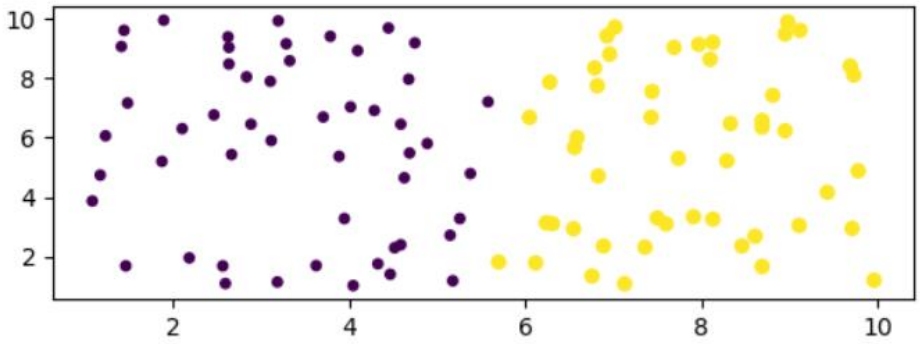
输出如图所示：

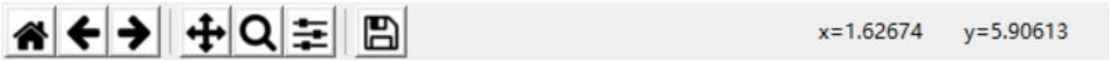




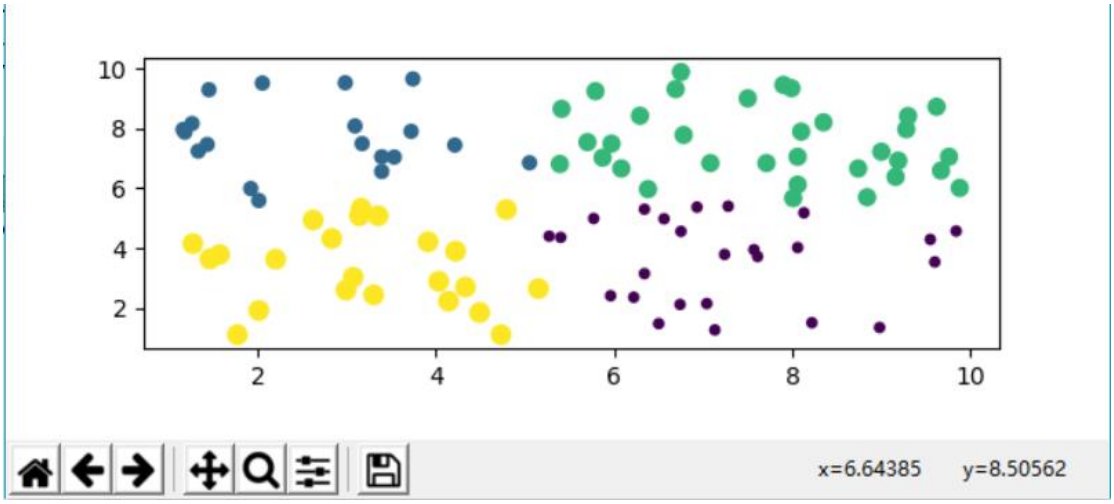
```
[array([4.14285714, 3.85714286]), array([83.4, 77.6])]
[[[1, 2], [2, 1], [3, 1], [5, 4], [5, 5], [6, 5], [7, 9]], [[99, 96], [94, 91], [92, 89], [87, 34], [45, 78]]]
```

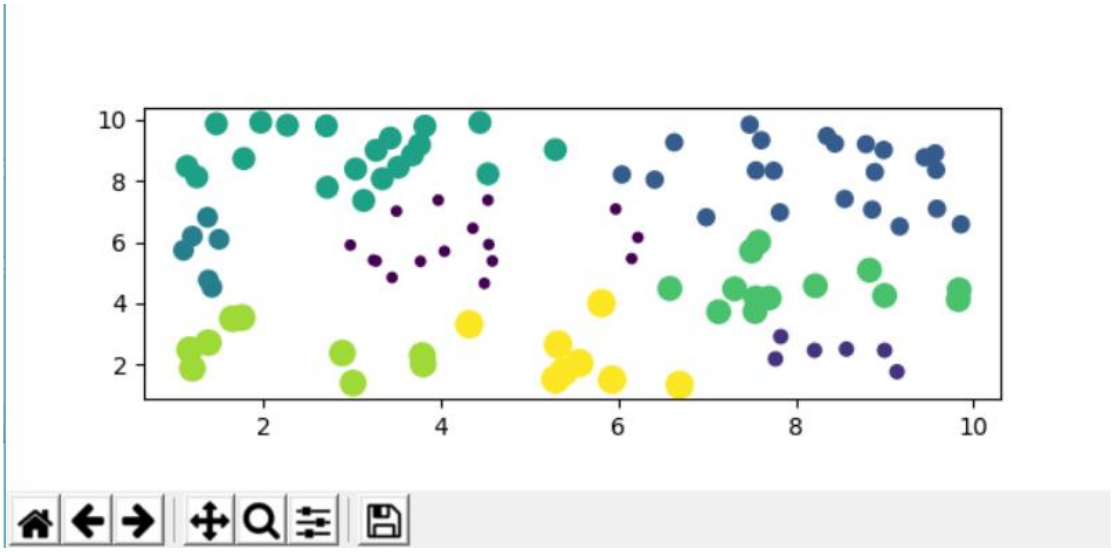
由于手动输入的数据集合数目较小，在默认生成的点集合进行簇聚类的时候，可以得到更为直观的效果：





可以指定分为更多的簇：





通过这个简单的 K-mean 算法的实验可以得到：

K 均值算法主要特征是简单，容易实现，并且是可以保证收敛到局部最优的。并且通过分析原理，其具有多项式时间复杂度。设 N，k 的意义同前，t 为算法迭代次数，则 k-均值聚类算法的时间复杂度为 O(rkN),所以 k 均值算法可以用于大规模的数据集。

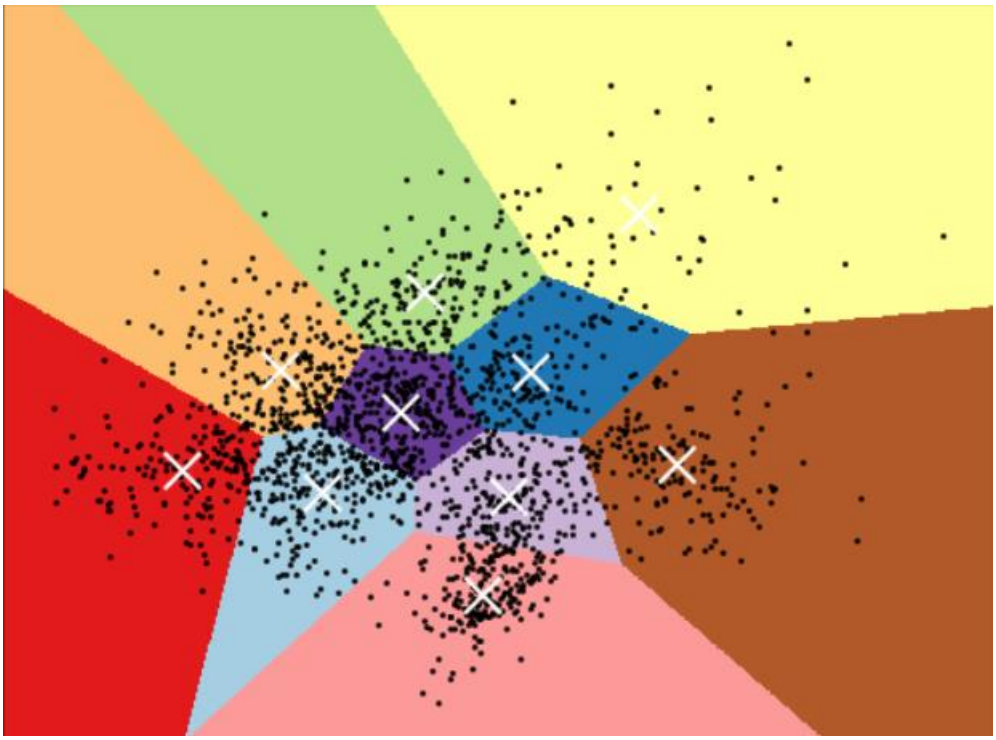
但是其也存在明显的不足：

- 是一种局部最优的算法，比较容易受初始值的影响。
- 采用均值来代表簇还是不够鲁棒，易受到噪声的影响。
- 需要事先给定聚类的个数，即 k 值，但是在很多情况下，我们并不知道需要划分成多少个聚类才是最合适的。

当然针对这三个问题，也有解决的办法，比如多次执行 k 均值算法，从而选择最好的结果，多次改变 k 值，也取最终最好的结果，为了得到全局最优解，也可以采用模拟退火和进化算法等，还有使用 ISODATA 算法和模糊 k-均值算法。

目前，在 sklearn 中已经对聚类有了较好的处理效果，并且，其考虑的也比较全面，比如考虑了数据的惯性，“维度诅咒”等因素。在运行 k 均值算法之前对数进行了注入 PCA 之类的降维算法，从而在一定程度上解决了这个问题，并且加速了计算。

Scikit-learn 中使用了 k-means++初始化方法，将均值点初始化为彼此远离，从而得到更快的收敛速度和更好的聚类效果。在官方给出的示例的处理效果是非常漂亮迷人的，效果非常不错，在官方的文档中给出了一个 K-mean 在手写数字数据上的聚类方法，就使用 PCA 方法等进行了一定的前期处理，由于使用了 sklearn 的库，代码显得十分简洁，并且其还得到了 V-means(v-度量),HOMO（同质性评分）,完整性评分，ARI（调整兰特指数）,AMI（互调整信息），轮廓等数据



```
=========== ============================================================
Shorthand    full name
=========== ============================================================
homo         homogeneity score
compl        completeness score
v-meas       V measure
ARI          adjusted Rand index
AMI          adjusted mutual information
silhouette   silhouette coefficient
=========== ============================================================
n_digits: 10,    n_samples 1797,    n_features 64
_____________________________________________________________________________
init        time    inertia homo    compl   v-meas  ARI AMI silhouette
k-means++   0.49s   69432   0.602   0.650   0.625   0.465   0.598   0.146
random      0.39s   69694   0.669   0.710   0.689   0.553   0.666   0.147
PCA-based   0.05s   70804   0.671   0.698   0.684   0.561   0.668   0.118
```

对三种不同的实现方式做出了比较，可以发现随意生成的初始化节点的效果确实要差一些。以下是代码：

```
1. """
2. ==========================================================
3. A demo of K-Means clustering on the handwritten digits data
4. ==========================================================
```

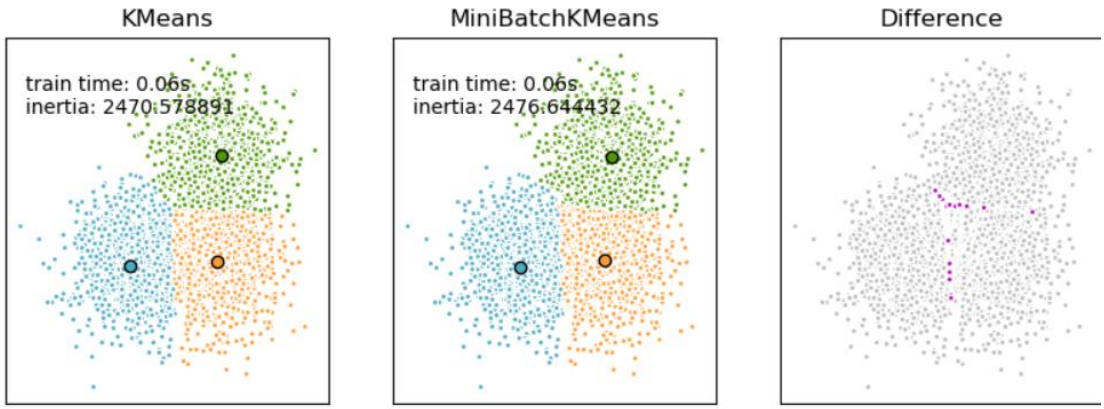
```
5.
6.  In this example we compare the various initialization strategies for
7.  K-means in terms of runtime and quality of the results.
8.
9.  As the ground truth is known here, we also apply different cluster
10. quality metrics to judge the goodness of fit of the cluster labels to the
11. ground truth.
12.
13. Cluster quality metrics evaluated (see :ref:`clustering_evaluation` for
14. definitions and discussions of the metrics):
15.
16. =========== ========================================================
17. Shorthand    full name
18. =========== ========================================================
19. homo         homogeneity score
20. compl        completeness score
21. v-meas       V measure
22. ARI          adjusted Rand index
23. AMI          adjusted mutual information
24. silhouette   silhouette coefficient
25. =========== ========================================================
26.
27. """
28. print(__doc__)
29.
30. from time import time
31. import numpy as np
32. import matplotlib.pyplot as plt
33.
34. from sklearn import metrics
35. from sklearn.cluster import KMeans
36. from sklearn.datasets import load_digits
37. from sklearn.decomposition import PCA
38. from sklearn.preprocessing import scale
39.
40. np.random.seed(42)
41.
42. digits = load_digits()
43. data = scale(digits.data)
44.
45. n_samples, n_features = data.shape
46. n_digits = len(np.unique(digits.target))
47. labels = digits.target
48.
```

```
sample_size = 300

print("n_digits: %d, \t n_samples %d, \t n_features %d"
      % (n_digits, n_samples, n_features))


print(82 * '_')
print('init\t\ttime\tinertia\thomo\tcompl\tv-meas\tARI\tAMI\tsilhouette')


def bench_k_means(estimator, name, data):
    t0 = time()
    estimator.fit(data)
    print('%-9s\t%.2fs\t%i\t%.3f\t%.3f\t%.3f\t%.3f\t%.3f\t%.3f'
          % (name, (time() - t0), estimator.inertia_,
             metrics.homogeneity_score(labels, estimator.labels_),
             metrics.completeness_score(labels, estimator.labels_),
             metrics.v_measure_score(labels, estimator.labels_),
             metrics.adjusted_rand_score(labels, estimator.labels_),
             metrics.adjusted_mutual_info_score(labels,  estimator.labels_),

             metrics.silhouette_score(data, estimator.labels_,
                                      metric='euclidean',
                                      sample_size=sample_size)))

bench_k_means(KMeans(init='k-means++', n_clusters=n_digits, n_init=10),
              name="k-means++", data=data)

bench_k_means(KMeans(init='random', n_clusters=n_digits, n_init=10),
              name="random", data=data)

# in this case the seeding of the centers is deterministic, hence we run the

# kmeans algorithm only once with n_init=1
pca = PCA(n_components=n_digits).fit(data)
bench_k_means(KMeans(init=pca.components_, n_clusters=n_digits, n_init=1),
              name="PCA-based",
              data=data)
print(82 * '_')

# ##############################################################################
# Visualize the results on PCA-reduced data

```

```
reduced_data = PCA(n_components=2).fit_transform(data)
kmeans = KMeans(init='k-means++', n_clusters=n_digits, n_init=10)
kmeans.fit(reduced_data)

# Step size of the mesh. Decrease to increase the quality of the VQ.
h = .02     # point in the mesh [x_min, x_max]x[y_min, y_max].

# Plot the decision boundary. For that, we will assign a color to each
x_min, x_max = reduced_data[:, 0].min() - 1, reduced_data[:, 0].max() + 1
y_min, y_max = reduced_data[:, 1].min() - 1, reduced_data[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h)
    )

# Obtain labels for each point in mesh. Use last trained model.
Z = kmeans.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure(1)
plt.clf()
plt.imshow(Z, interpolation='nearest',
           extent=(xx.min(), xx.max(), yy.min(), yy.max()),
           cmap=plt.cm.Paired,
           aspect='auto', origin='lower')

plt.plot(reduced_data[:, 0], reduced_data[:, 1], 'k.', markersize=2)
# Plot the centroids as a white X
centroids = kmeans.cluster_centers_
plt.scatter(centroids[:, 0], centroids[:, 1],
            marker='x', s=169, linewidths=3,
            color='w', zorder=10)
plt.title('K-means clustering on the digits dataset (PCA-reduced data)\n'
          'Centroids are marked with white cross')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())
plt.show()
```

在 sklearn 中还介绍了一种有意思的 kmean 的另一种处理方式，称为 MiniBatchKMeans 算法，叫做小批量 K-均值算法。其是在大的数据集合中选择一个小数据集合样本，在此中使用传统的 K 均值聚类，然后再将所有的数据分配到这三个聚类之中，这样做的方法最为明显的效果就是大大减少了汇聚到本地解决方案中多需要的计算量。这种方法的优点就是可以大幅度的降低运行的时间，但

是最后的结果也比较理想，在 sklearn 中给出了具体的程序，我做了实验，可以看看具体的运行的效果，并进行比较：



从上图可以看出，两种方法的效果差异是非常小的，所以，在处理超大规模的数据的时候，这不失为一个非常好的处理方法。

在 sklearn 的官方文档文档中还给出了两个有意思的项目，一个是使用 K-mean 分类文本，另外一个是使用 K-mena 分类人脸的部分，都非常的有趣，我也自己把下载下来进行了实验，发现其效果非常好，基本达到了 100%的正确率，当然这也得益于其前期对于训练数据的一些比较好的处理的。