第十六周学习笔记—K 均值聚类算法

王振宽

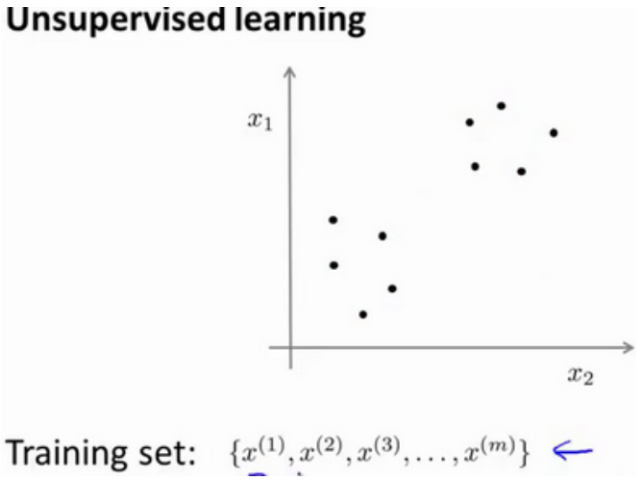
2022-07-31

第一章　K 均值聚类算法

1.1　无监督学习

在一个典型的监督学习中，我们有一个有标签的训练集，我们的目标是找到能够区分正样本和负样本的决策边界，在这里的监督学习中，我们有一系列标签，我们需要据此拟合一个假设函数。

与此不同的是，在非监督学习中，我们的数据没有附带任何标签，我们拿到的数据就是这样的：



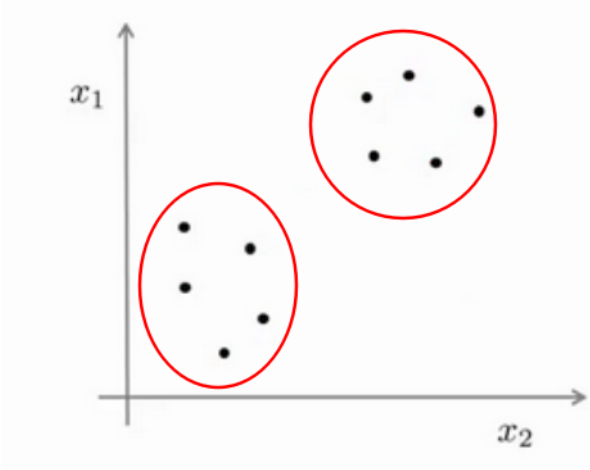
在这里我们有一系列点，却没有标签。

因此，我们的训练集可以写成只有 $x^{(1)}$, $x^{(2)}$ $\cdots$ $x^{(m)}$。我们没有任何标签 y。因此，图上画的这些点没有标签信息。

也就是说，在非监督学习中，我们需要将一系列无标签的训练数据，输入到一个算法中，然后我们告诉这个算法，快去为我们找找这个数据的内在结构给定数据。

我们可能需要某种算法帮助我们寻找一种结构。

图上的数据看起来可以分成两个分开的点集（称为簇），一个能够找到我圈出的这些点集的算法，就被称为聚类算法。

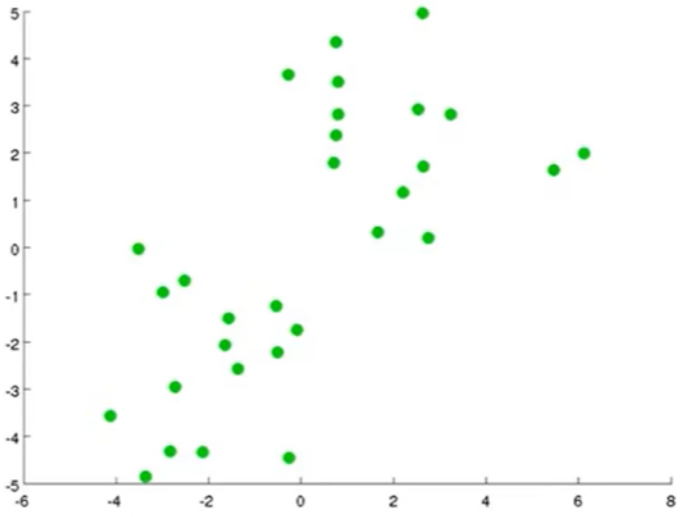


当然，此后我们还将提到其他类型的非监督学习算法，它们可以为我们找到其他类型的结构或者其他的一些模式，而不只是簇。

我们将先介绍聚类算法。

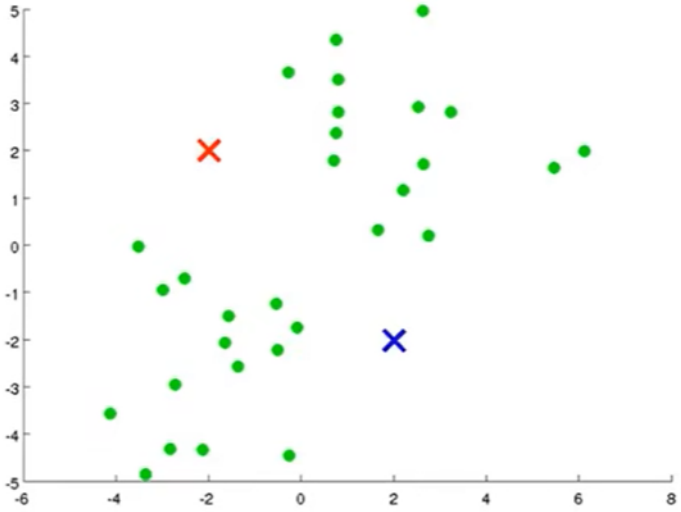
1.2　K-均值算法

K-均值是最普及的聚类算法，算法接受一个未标记的数据集，然后将数据聚类成不同的组。



假设我有一个无标签的数据集，并且我想将其分成两个簇，现在我利用 K 均值算法。具体操作如下:

第一步是随机生成两点，这两点就叫做聚类中心，也就是图上两个叉:

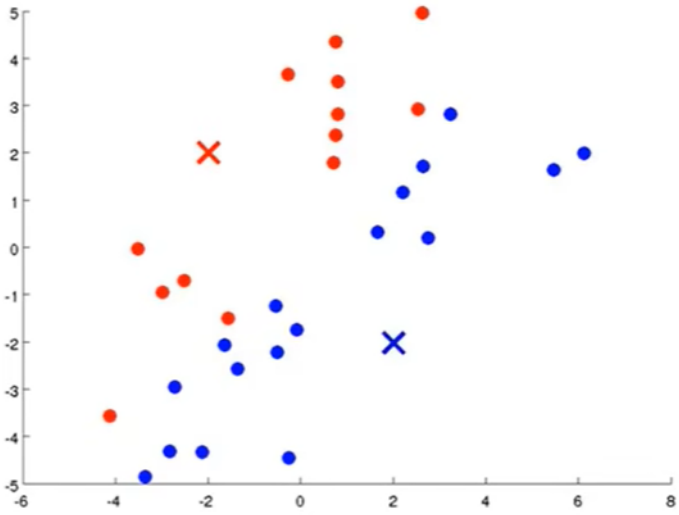


我选取两个点的原因是因为我想将我的数据聚成两类。

K-均值算法是一个迭代算法，它会去做两件事情: 第一个是簇分配，第二个是移动聚类中心。

K 均值算法中，每次内循环的第一步就是要进行簇分配。也就是说，我要遍历每个样本，也就是图上的每个绿色的点，然后根据每一个点是与红色聚类中心更近还是与蓝色聚类中心更近来将每个数据点分配给两个聚类中心之一。

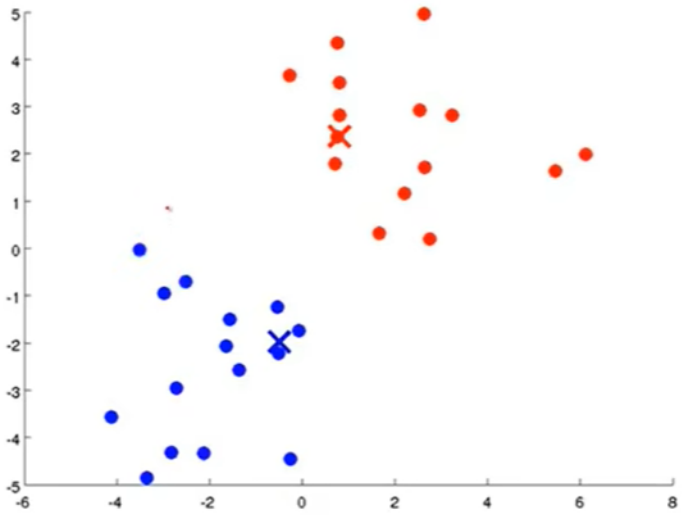
具体来说，就是要遍历我们的数据集，然后将每个点染成红色或者蓝色，这取决于某个点是离红色更近还是蓝色更近。



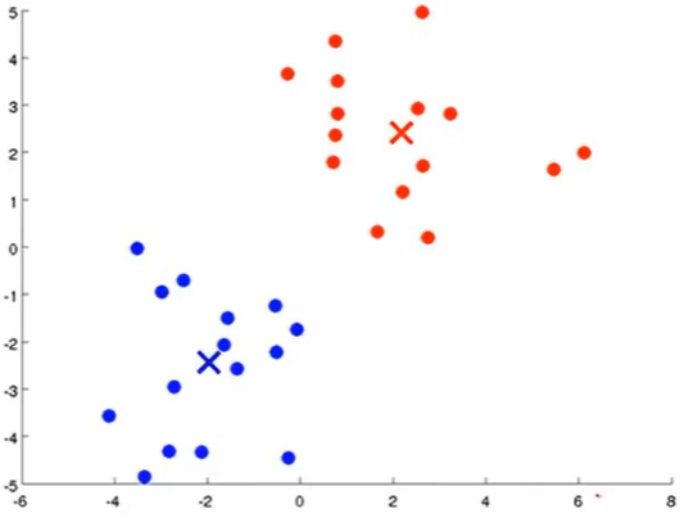
内循环的第二步是移动聚类中心。

我们要做的是将两个聚类中心，也就是上面红蓝两个叉将其移动到同色的点的均值处。因此我们要做的是找出所有红色的点然后计算它们的均值，也就是所有红色的点的

平均位置，然后将红色聚类中心移动到这里。蓝色聚类点实行同样的操作。



接下来我们重复上面的步骤，直到最后完成要求:



K-均值是一个迭代算法，假设我们想要将数据聚类成 n 个组，其方法为:

1. 首先选择 K 个随机的点，称为聚类中心（cluster centroids）

2. 对于数据集中的每一个数据，按照距离 K 个中心点的距离，将其与距离最近的中心点关联起来，与同一个中心点关联的所有点聚成一类。

3. 计算每一个组的平均值，将该组所关联的中心点移动到平均值的位置

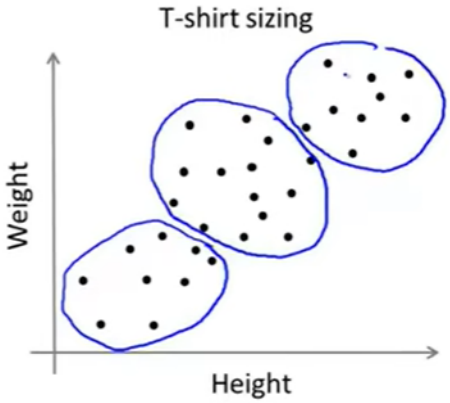
4. 重复步骤 2-4 直至中心点不再变化

我们用 μ^1 , μ^2 $\cdots$ μ^k 来表示聚类中心，用 $c^{(1)}$, $c^{(2)}$, $\cdots$, $c^{(m)}$ 来存储与第 i 个实例数据最近的聚类中心的索引，K-均值算法的伪代码如下：

```
x
Repeat{
  for i = 1 to m:
    c(i):=index(from 1 to k) of cluster centroid closest to x(i)
  for k = 1 to K:
     k := average (mean) of points assigned to cluster k
 }
```

算法分为两个步骤，第一个 for 循环是赋值步骤，即：对于每一个样例 i，计算其应该属于的类。第二个 for 循环是聚类中心的移动，即：对于每一个类 K，重新计算该类的质心。

K-均值算法也可以很便利地用于将数据分为许多不同组，即使在没有非常明显区分的组群的情况下也可以。下图所示的数据集包含身高和体重两项特征构成的，利用 K-均值算法将数据分为三类，用于帮助确定将要生产的 T-恤衫的三种尺寸。



1.3　优化目标

K-均值最小化问题，是要最小化所有的数据点与其所关联的聚类中心点之间的距离之和，因此 K-均值的代价函数（又称畸变函数 Distortion function）为：

$$J(c^{(1)}\ ,\ c^{(2)}\ ,\ \cdots\ ,\ ,c^{(m)}\ ,\ \mu_1\ ,\ \mu_2\ ,\ \cdots\ ,\ \mu_K) = \frac{1}{m}\sum_{i=1}^{m}||X^{(i)} - \mu_{c^{(i)}}||^2$$

其中 $\mu_{c^{(i)}}$ 代表与 $x^{(i)}$ 最近的聚类中心点。我们的的优化目标便是找出使得代价函

数最小的 $c^{(1)}$, $c^{(2)}$, $\cdots$, , $c^{(m)}$ 和 μ_1 , μ_2 , $\cdots$, μ_K:

$$\min_{c^{(1)}\,,\,\cdots\,,\,c^{(m)}\,,\,\mu_1\,,\,\cdots\,,\,\mu_K} J(c^{(1)}\,,\,c^{(2)}\,,\,\cdots\,,\,,c^{(m)}\,,\,\mu_1\,,\,\mu_2\,,\,\cdots\,,\,\mu_K)$$

回顾刚才给出的: K-均值迭代算法，我们知道，第一个循环是用于减小 $c^{(i)}$ 引起的代价，而第二个循环则是用于减小 μ_i 引起的代价。迭代的过程一定会是每一次迭代都在减小代价函数，不然便是出现了错误。

1.4 随机初始化

接下来讨论一下如何初始化 K 均值聚类算法，更重要的是，这将引导我们如何使算法避免局部最优解。

这是我们的 K 均值算法:

K-means algorithm

Randomly initialize K cluster centroids $\mu_1, \mu_2, \ldots, \mu_K \in \mathbb{R}^n$

Repeat {
  for i = 1 to m
    $c^{(i)}$:= index (from 1 to K) of cluster centroid closest to $x^{(i)}$
  for k = 1 to K
    μ_k := average (mean) of points assigned to cluster k
}

其中我们没有讨论的就是最开始的如何初始化聚类中心，有几种不同的方法可以用来随机初始化聚类中心。

但是，事实证明，有一种方法比其他大多数可能考虑到的方法更值得推荐。

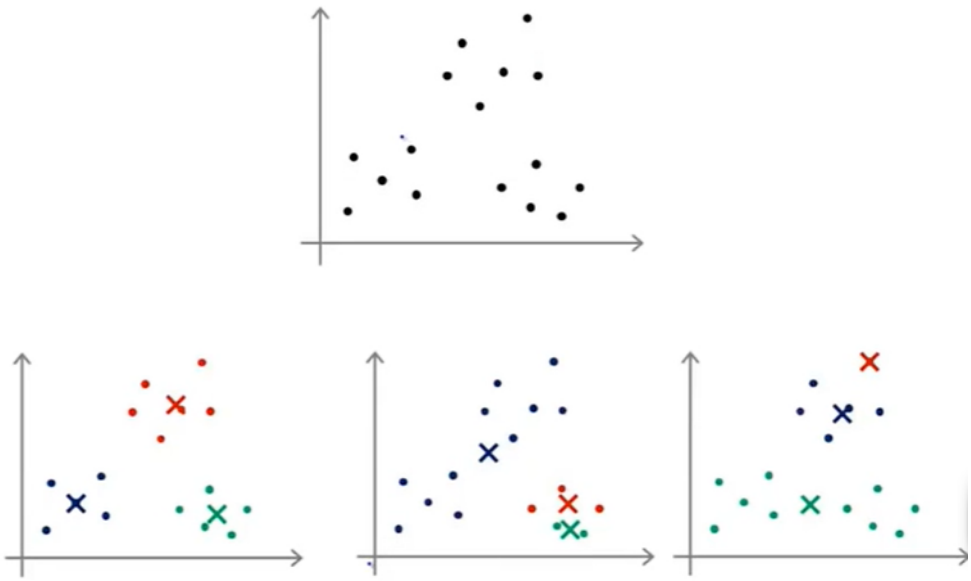
我们通常是这样初始化聚类中心的:

（1）我们应该选择 $K < m$，即聚类中心点的个数要小于所有训练实例的数量

（2）随机选择 K 个训练实例，然后令 K 个聚类中心分别与这 K 个训练实例相等

具体来说，K 均值算法可能会落在局部最优，这取决于初始化的情况。我们给出下面的例子:

我们可能的分类如下:



为了解决这个问题，我们通常需要多次运行 K 均值算法，每一次都重新进行随机初始化，最后再比较多次运行 K 均值的结果，选择代价函数最小的结果。

以下是具体的做法:

Random initialization

For i = 1 to 100 {

Randomly initialize K-means.
Run K-means. Get $c^{(1)}, \ldots, c^{(m)}, \mu_1, \ldots, \mu_K$.
Compute cost function (distortion)
$J(c^{(1)}, \ldots, c^{(m)}, \mu_1, \ldots, \mu_K)$
}

这种方法在 K 较小的时候还是可行的，但是如果 K 值较大，这么做也是不会有明显的改善。

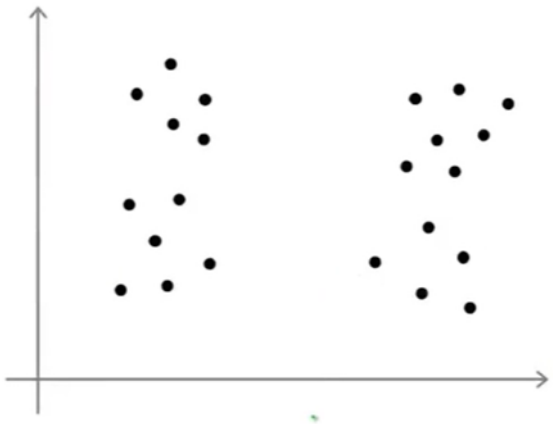
1.5　选择聚类数

如何去选择聚类数量或者说如何去选择参数 K 的值？

说实话，这个问题没有什么好的答案，也没有能自动处理的方法。

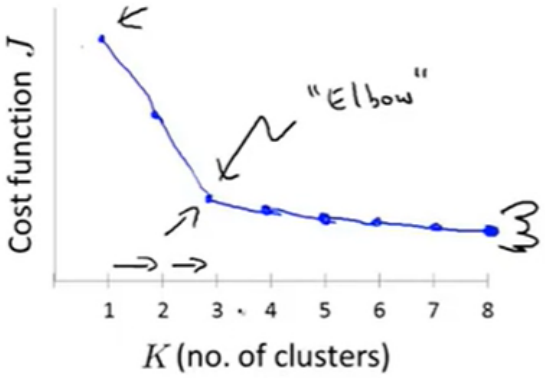
目前为止，用来决定聚类数量最常用的方法仍然是通过观察可视化的图或者通过观察聚类算法的输出等等。

我们举上面的例子来看，对于上面的数据集，我们该如何选择聚类中心的数目，是选择 K=2 还是选择 K=4？



当人们在讨论选择聚类数量的方法时，有一个可能会谈及的方法叫作“肘部法则”。

关于“肘部法则”，我们所需要做的是改变 K 值，也就是聚类类别数目的总数。我们用一个聚类来运行 K 均值聚类方法。这就意味着，所有的数据都会分到一个聚类里，然后计算成本函数或者计算畸变函数 J。K 代表聚类数字。

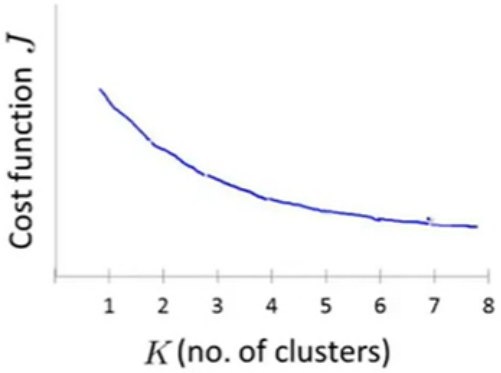


我们可能会得到一条类似于这样的曲线。像一个人的肘部。这就是“肘部法则”所做的，让我们来看这样一个图，看起来就好像有一个很清楚的肘在那儿。好像人的手臂，如果你伸出你的胳膊，那么这就是你的肩关节、肘关节、手。这就是“肘部法则”。你会发现这种模式，它的畸变值会迅速下降，从 1 到 2，从 2 到 3 之后，你会在 3 的时候达到一个肘点。在此之后，畸变值就下降的非常慢，看起来就像使用 3 个聚类来进行聚类是正确的，这是因为那个点是曲线的肘点，畸变值下降得很快，K=3 之后就下降得很慢，那么我们就选 K=3。当你应用“肘部法则”的时候，如果你得到了一个像上面这样的图，那么这将是一种用来选择聚类个数的合理方法。

但是我们有时会得到一条模糊的曲线，如下图所示:

看上去畸变值是连续下降的，可能是 3，也可能是 4。

对于“肘部法则”快速的小结: 它是一个值得尝试的方法，但是不能期望它能够解决任何问题。

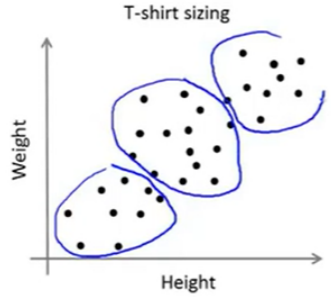


最后还有一种选择 K 值的思路:

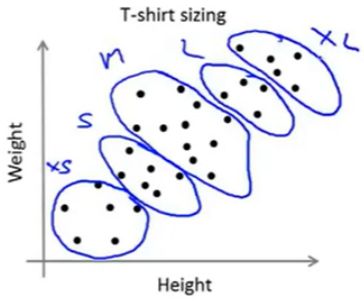
通常人们使用 K 均值聚类是为了得到一些聚类用于后面的目的或者说是下游目的。也许你想用 K 均值来做市场分割，就像我们之前说的 T 恤尺寸的例子。

比如我是否需要选择 3 种 T 恤尺寸。因此我选择 K=3，可能有小号，中号和大号三种尺寸的 T 恤，或者我们可以选择 K=5，那么我们就可能有特小号、小号、中号、大号和特大号尺寸的 T 恤。所以我们可以有 3 种或者 5 种。

如果我们用 K=3 来进行分类，可以得到下面的结果:



然而如果我们用 K=5 来跑 K 均值算法:



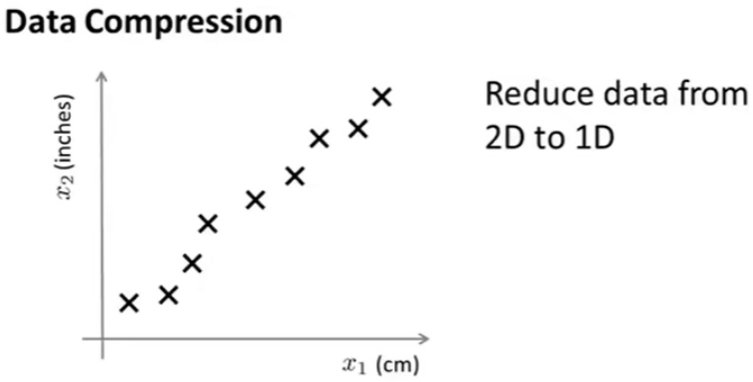
综上所述: 没有所谓最好的选择聚类数的方法，通常是需要根据不同的问题，人工进行选择的。选择的时候思考我们运用 K-均值算法聚类的动机是什么，然后选择能最好服务于该目的标聚类数。

第二章　降维

有几个不同的的原因使你可能想要做降维。一是数据压缩。数据压缩不仅允许我们压缩数据，因而使用较少的计算机内存或磁盘空间，但它也让我们加快我们的学习算法。

2.1　动机一：数据压缩

但首先，让我们谈论降维是什么。我们举一个例子来说明: 我们收集到的数据集，有许多特征，我绘制两个在这里。

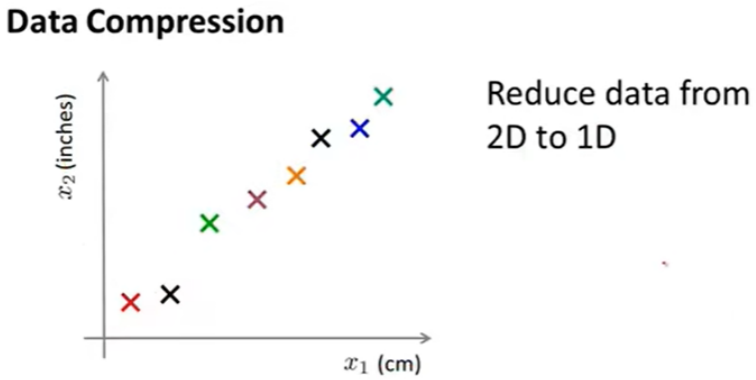


假设我们未知两个的特征：x_1: 长度，用厘米表示；x_2: 是用英寸表示同一物体的长度。

所以，这给了我们高度冗余表示，也许不是两个分开的特征 x_1 , x_2，这两个基本的长度度量。也许我们想做的是要减少数据到一维，只有一个数测量这个长度。

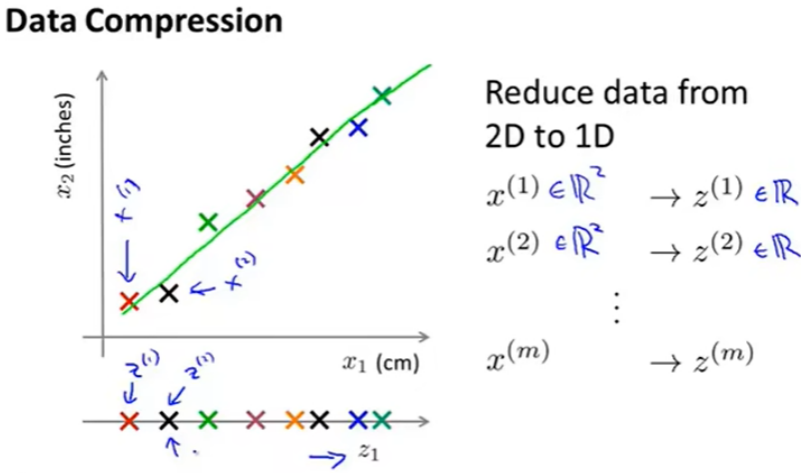
假设我们要采用两种不同的仪器来测量一些东西的尺寸，其中一个仪器测量的结果的单位是英寸，另一个仪器测量的结果是厘米，我们希望将测量的结果作为我们机器学习的特征。现在的问题是，两种仪器对于同一个东西测量的结果不完全相等（由于误差、精度等因素），而将两者都作为特征又有些重复，因此，我们希望将这个二维的数据降到一维。

现在我把不同的样本用不同的颜色标出来，如下所示:



这时，通过降维，意思是，我们想找出一条线，看起来大多数样本所在的线，所有的数据都投影到了刚刚所画的线上。

通过这种做法，我们能做到测量出每个样本在线上的位置，现在我们要做的就是建立新的特征 z_1，我们只需要一个数就能确定 z_1 所在的位置。也就是说，z_1 是一个新的特征，它能够指定上面直线上每一个点的位置。



将数据从三维降至二维：我们要将一个三维的特征向量降至一个二维的特征向量。过程是与上面类似的，我们将三维向量投射到一个二维的平面上，强迫使得所有的数据都在同一个平面上，降至二维的特征向量。

2.2　动机二：数据可视化

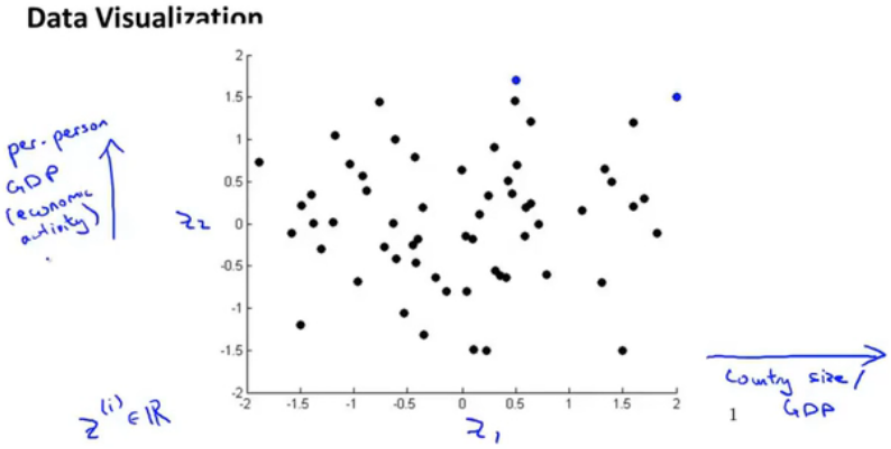
在许多及其学习问题中，如果我们能将数据可视化，我们便能寻找到一个更好的解决方案，降维可以帮助我们。

假使我们有有关于许多不同国家的数据，每一个特征向量都有 50 个特征（如 GDP，人均 GDP，平均寿命等）。

Data Visualization

Country	GDP (trillions of US$)	Per capita GDP (thousands of intl. $)	Human Develop-ment Index	Life expectancy	Poverty Index (Gini as percentage)	Mean household income (thousands of US$)	...
Canada	1.577	39.17	0.908	80.7	32.6	67.293	...
China	5.878	7.54	0.687	73	46.9	10.22	...
India	1.632	3.41	0.547	64.7	36.8	0.735	...
Russia	1.48	19.84	0.755	65.5	39.9	0.72	...
Singapore	0.223	56.69	0.866	80	42.5	67.1	...
USA	14.527	46.86	0.91	78.3	40.8	84.3	...
...	...	...	...	...	...	...	

如果要将这个 50 维的数据可视化是不可能的。使用降维的方法将其降至 2 维，我们便可以将其可视化了。

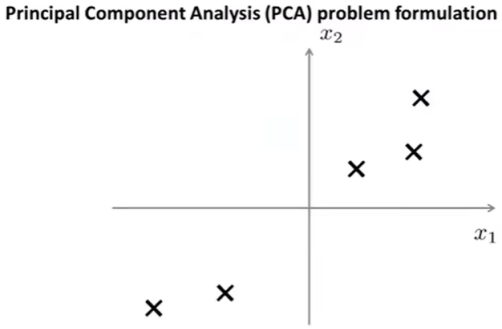


这样做的问题在于，降维的算法只负责减少维数，新产生的特征的意义就必须由我们自己去发现了。

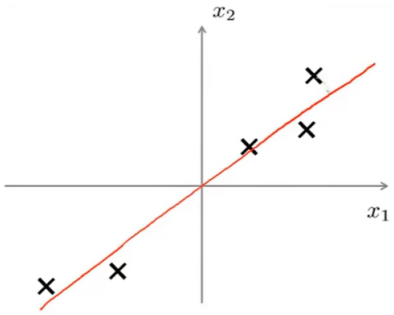
2.3　主成分分析问题一

主成分分析 (PCA) 是最常见的降维算法。

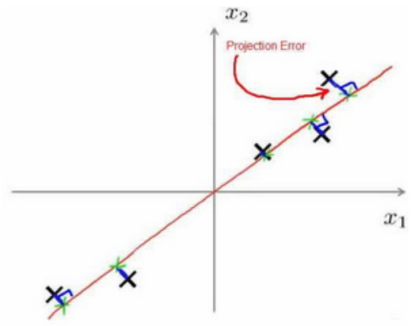
我们先通过一个例子来入手:



假设我想对上面的数据进行降维，从二维降到一维，也就是说我想找到一条能够将数据投影到上面的直线:



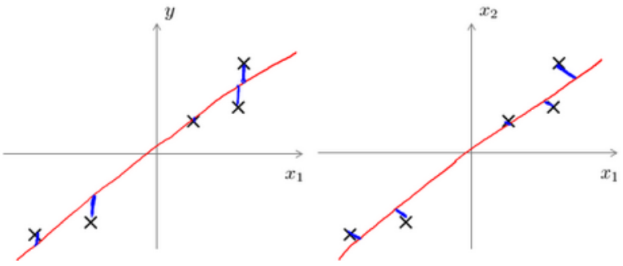
在 PCA 中，我们要做的是找到一个方向向量（Vector direction），当我们把所有的数据都投射到该向量上时，我们希望投射平均均方误差能尽可能地小。方向向量是一个经过原点的向量，而投射误差是从特征向量向该方向向量作垂线的长度。



正式地说，PCA 做的就是找到一个低维平面，然后将数据投影到上面。

下面给出主成分分析问题的描述:

问题是要将 n 维数据降至 k 维，目标是找到向量 $\mu^{(1)}, \mu^{(2)}, \cdots, \mu^{(k)}$ 使得总的投射误差最小。



看到我们上面的图像，我们发现和线性回归的图像很相似。但是，**主成分分析和线性回归是两种不同的算法。**

主成分分析最小化的是投射误差（Projected Error），而线性回归尝试的是最小化预测误差。线性回归的目的是预测结果，而主成分分析不作任何预测。

上图中，左边的是线性回归的误差（垂直于横轴投影），右边则是主要成分分析的误差（垂直于红线投影）。

2.4　主成分分析问题二

PCA 将 n 个特征降维到 k 个，可以用来进行数据压缩，如果 100 维的向量最后可以用 10 维来表示，那么压缩率为 90%。

但 PCA 要保证降维后，还要保证数据的特性损失最小。

PCA 技术的一大好处是对数据进行降维的处理。我们可以对新求出的“主元”向量的重要性进行排序，根据需要取前面最重要的部分，将后面的维数省去，可以达到降维从而简化模型或是对数据进行压缩的效果。同时最大程度的保持了原有数据的信息。

PCA 技术的一个很大的优点是，它是完全无参数限制的。在 PCA 的计算过程中完全不需要人为的设定参数或是根据任何经验模型对计算进行干预，最后的结果只与数据相关，与用户是独立的。

但是，这一点同时也可以看作是缺点。如果用户对观测对象有一定的先验知识，掌握了数据的一些特征，却无法通过参数化等方法对处理过程进行干预，可能会得不到预期的效果，效率也不高。

2.5　主成分分析算法

PCA 减少 n 维到 k 维:

第一步是均值归一化。我们需要计算出所有特征的均值，然后令 $x_j = x_j - \mu_j$。如果特征是在不同的数量级上，我们还需要将其除以标准差 σ^2。

第二步是计算协方差矩阵（covariance matrix）Σ:

$$\Sigma = \frac{1}{m}\sum_{i=1}^{n}(x^{(i)})(x^{(i)})^T$$

第三步是计算协方差矩阵 Σ 的特征向量（eigenvectors）:

$$U = \begin{bmatrix} | & | & & | \\ u^{(1)} & u^{(2)} & \dots & u^{(n)} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{n\times n}$$

$$Sigma = \frac{1}{m}\sum_{i=1}^{n}\left(x^{(i)}\right)\left(x^{(i)}\right)^T$$

对于一个 $n \times n$ 维度的矩阵，上式中的 U 是一个具有与数据之间最小投射误差的方向向量构成的矩阵。如果我们希望将数据从 n 维降至 k 维，我们只需要从 U 中选取前 k 个向量，获得一个 $n \times k$ 维度的矩阵，我们用 U_{reduce} 表示，然后通过如下计算获得要求的新特征向量 $z^{(i)}$: $z^{(i)} = U_{reduce}^T * x^{(i)}$。

2.6　选择主成分的数量

主成分分析是减少投射的平均均方误差。

训练集的方差为:

$$\frac{1}{m}\sum_{i=1}^{m}||x^{(i)}||^2$$

我们希望在平均均方误差与训练集方差的比例尽可能小的情况下选择尽可能小的 k 值。

如果我们希望这个比例小于 1%，就意味着原本数据的偏差有 99% 都保留下来了，如果我们选择保留 95% 的偏差，便能非常显著地降低模型中特征的维度了。

我们可以先令 $k = 1$，然后进行主成分分析，获得 U_{reduce} 和 z，然后计算比例是否小于 1%。如果不是的话再令 $k = 2$，如此类推，直到找到可以使得比例小于 1% 的最小 k 值（原因是各个特征之间通常情况存在某种相关性）。

我们的 S 是一个 $n \times n$ 的矩阵:

$$S = \begin{bmatrix} s_{11} & & & & \\ & s_{22} & & & \\ & & s_{33} & & \\ & & & \cdots & \\ & & & & s_{44} \end{bmatrix}$$

只有对角线上有值，而其它单元都是 0，我们可以使用这个矩阵来计算平均均方误差与训练集方差的比例:

$$\frac{\frac{1}{m}\sum_{i=1}^{m}||x^{(i)} - x_{approx}^{(i)}||^2}{\frac{1}{m}\sum_{i=1}^{m}||x^{(i)}||^2} = 1 - \frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{m} S_{ii}} \leq 1\%$$

也就是:

$$\frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{m} S_{ii}} \geq 0.99$$

在压缩过数据后，我们可以采用如下方法来近似地获得原有的特征:

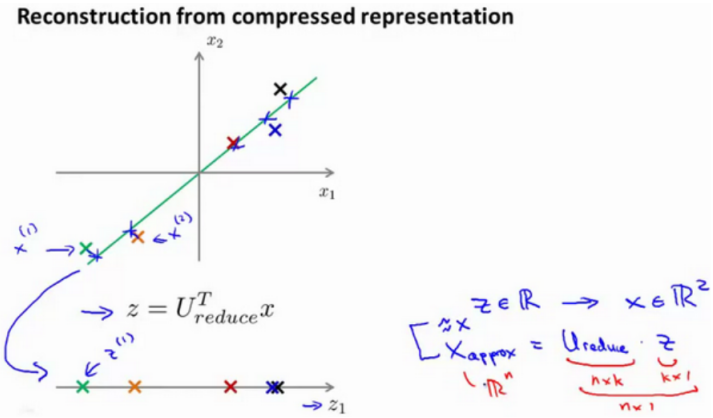
$$x_{approx}^{(i)} = U_{reduce} z^{(i)}$$

2.7　重建的压缩表示

在以前的学习中，我们谈论 PCA 作为压缩算法。在那里你可能需要把 1000 维的数据压缩 100 维特征，或具有三维数据压缩到一二维表示。

所以，如果这是一个压缩算法，应该能回到这个压缩表示，回到你原有的高维数据的一种近似。

所以，给定的 $z^{(i)}$，这可能是 100 维，怎么回到你原来的表示 $x^{(i)}$，这可能是 1000 维的数组？



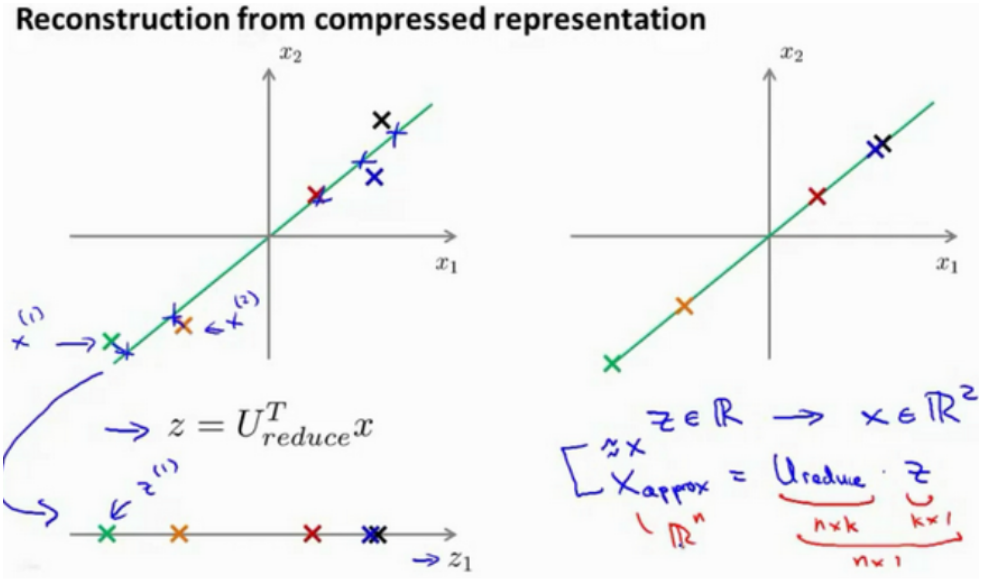
PCA 算法，我们可能有一个这样的样本。如图中样本 $x^{(1)}$，$x^{(2)}$。我们做的是，我们把这些样本投射到图中这个一维平面。然后现在我们只需要使用一个实数，比如 $z^{(1)}$，指定这些点的位置后他们被投射到这一个三维曲面。给定一个点 $z^{(1)}$，我们怎么能回到这个原始的二维空间呢？

x 为 2 维，z 为 1 维，$z = U_{reduce}^T x$，相反的方程为:

$$x_{approx} = U_{reduce} \cdot z$$

$$x_{approx} \approx x$$

如图:



如你所知，这是一个漂亮的与原始数据相当相似。

所以，这就是你从低维表示 z 回到未压缩的表示。我们得到的数据的一个之间你的原始数据 x，我们也把这个过程称为重建原始数据。

当我们认为试图重建从压缩表示 x 的初始值。所以，给定未标记的数据集，您现在知道如何应用 PCA，你的带高维特征 x 和映射到这的低维表示 z。

2.8　主成分分析法的应用建议

假使我们正在针对一张 100×100 像素的图片进行某个计算机视觉的机器学习，即总共有 10000 个特征。

(1) 第一步是运用主要成分分析将数据压缩至 1000 个特征

(2) 然后对训练集运行学习算法

(3) 在预测时，采用之前学习而来的 U_{reduce} 将输入的特征 x 转换成特征向量 z，然后再进行预测。

注：如果我们有交叉验证集合测试集，也采用对训练集学习而来的 U_{reduce}。

我们总结一下错误的原因:

一个常见错误使用主要成分分析的情况是，将其用于减少过拟合（减少了特征的数量）。这样做非常不好，不如尝试正则化处理。原因在于主要成分分析只是近似地丢弃掉一些特征，它并不考虑任何与结果变量有关的信息，因此可能会丢失非常重要的特征。然而当我们进行正则化处理时，会考虑到结果变量，不会丢掉重要的数据。

另一个常见的错误是，默认地将主要成分分析作为学习过程中的一部分，这虽然很多时候有效果，最好还是从所有原始特征开始，只在有必要的时候（算法运行太慢或者占用太多内存）才考虑采用主要成分分析。

第三章　K-means 均值实战项目

3.1　K-means 算法

K-means 算法为一种无监督学习算法，可以实现算法自动区分数据集中不同簇。

无监督学习区别于监督学习算法，无监督学习中没有标签 y，算法需要根据输入的数据集直接将其进行区分为各簇，每簇数据有其聚类中心。

K-means 算法是一种自动将相似数据示例聚类在一起的方法。具体来说，给你一个训练集 $x^{(1)}$；$x^{(m)}$，（其中 $x^{(i)} \in R^n$），并且想要将数据分组到几个有凝聚力的“簇”中。

K-means 背后是一个迭代过程，首先猜测初始聚类中心，然后通过重复将示例分配给它们最接近的聚类中心，然后根据分配重新计算聚类中心来重新猜测。

算法步骤总结如下:

1. 随机选取 k 个聚类中心点

2. 遍历所有数据，将数据划分到最近的那个聚类中心点

3. 计算所有类的平均值，作为新的聚类中心点

4. 重复步骤 2 和步骤 3，直到聚类中心点不再发生变化，或者达到设定的迭代次数

计算每个样本点到聚类中心的距离为:

$$c^{(i)} := j \ \ that \ \ minimizes \ \ ||x^{(i)} - \mu_j||^2$$

计算新的聚类中心点：

$$\mu_k := \frac{1}{|C_k|} \sum_{i \in C_k} x^{(i)}$$

3.2　给定一个二维数据集，利用 k-means 聚类

3.2.1　导入运算包

```
import numpy as np
import pandas as pd
```

```
import matplotlib.pyplot as plt
from scipy.io import loadmat
import matplotlib as mpl
```

3.2.2　导入需要的数据集

```
def load_dataset():
    path='./data/ex7data2.mat'
    # 字典格式 : <class 'dict'>
    data=loadmat(path)
    # data.keys() : dict_keys(['__header__', '__version__', '__globals__',
                                            'X'])
    dataset = pd.DataFrame(data.get('X'), columns=['X1', 'X2'])
    return data,dataset
```

我们可以简单看一下输出结果:

首先是 data:

```
{'__header__': b'MATLAB 5.0 MAT-file, Platform: GLNXA64, Created on: Wed
                                        Nov 16 00:48:22 2011', '__version__':
                                         '1.0', '__globals__': [], 'X': array
                                        ([[ 1.84207953,  4.6075716 ],
       [ 5.65858312,  4.79996405],
       [ 6.35257892,  3.2908545 ],
       [ 2.90401653,  4.61220411],
       [ 3.23197916,  4.93989405],
       [ 1.24792268,  4.93267846],
......中间省略部分数据集
       [ 4.8255341 ,  2.77961664],
       [ 6.11768055,  2.85475655],
       [ 0.94048944,  5.71556802]])}
```

接下来是 dataset:

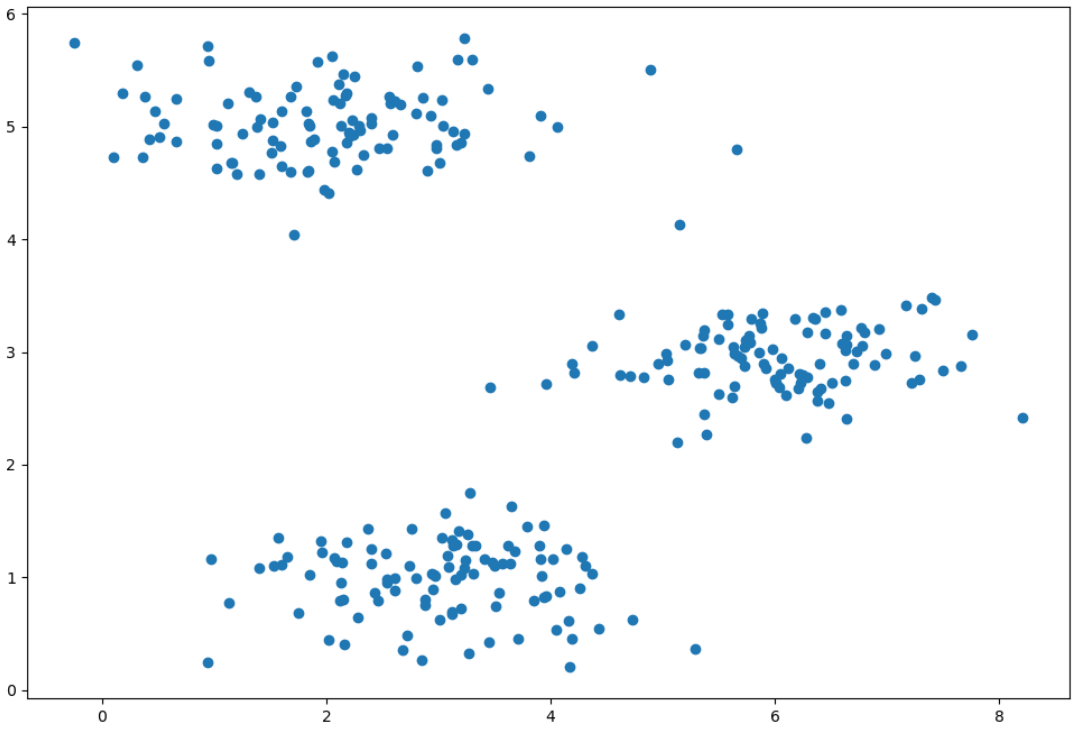
```
           X1        X2
0    1.842080  4.607572
1    5.658583  4.799964
2    6.352579  3.290854
3    2.904017  4.612204
4    3.231979  4.939894
```

```
..         ...        ...
295   7.302787   3.380160
296   6.991984   2.987067
297   4.825534   2.779617
298   6.117681   2.854757
299   0.940489   5.715568
```

3.2.3　数据可视化

拿到数据集后，我们可以可视化一下，看一下数据集的样子：

```
def plot_scatter():
    data,dataset=load_dataset()
    plt.figure(figsize=(12,8))
    plt.scatter(dataset['X1'],dataset['X2'],cmap=['b'])
    plt.show()
```



3.2.4　寻找数据点的最近的聚类中心

我们这一步的目标是定义函数，用于寻找数据点的最近的聚类中心。

输入参数为数据集 X、初始聚类中心，返回一个一维数组，其长度与 X 的数据点个数相同，每个索引对应的值为该点对应的聚类中心。

```
def get_near_cluster_centroids(X,centroids):
    """
    :param X: 我们的数据集
    :param centroids: 聚类中心点的初始位置
    :return:
    """
    m = X.shape[0] #数据的行数
    k = centroids.shape[0] #聚类中心的行数，即个数
    idx = np.zeros(m) # 一维向量idx，大小为数据集中的点的个数，用于保存每一
                                          个X的数据点最小距离点的是哪个聚类
                                          中心
    for i in range(m):
        min_distance = 1000000
        for j in range(k):
            distance = np.sum((X[i, :] - centroids[j, :]) ** 2) # 计算数据
                                                  点到聚类中心距离代价的公
                                                  式，X中每个点都要和每个聚
                                                  类中心计算
            if distance < min_distance:
                min_distance = distance
                idx[i] = j # idx中索引为i，表示第i个X数据集中的数据点距离最
                                                      近的聚类中心的索引
    return idx # 返回的是X数据集中每个数据点距离最近的聚类中心
```

3.2.5　编写计算聚类中心函数

传入数据集 X、聚类中心索引 idx、聚类中心个数 k。

算法根据 idx 将 X 分为 k 个组，每个组有多个 X 内的数据集，计算其平均值，得到求平均值之后的聚类中心。返回值为求一次平均值之后的聚类中心。

本函数需要传入 idx 数组，故不能用于直接初始化聚类中心，只起到优化聚类中心的作用。

```
def compute_centroids(X, idx, k):
```

```
    """
    :param X: 数据集
    :param idx: 每个样本所属的类别
    :param k: 类别总数
    :return:
    """
    m, n = X.shape
    centroids = np.zeros((k, n)) # 初始化为k行n列的二维数组，值均为0，k为聚
                                    类中心个数，n为数据列数
    for i in range(k):
        indices = np.where(idx == i) # 输出的是索引位置
        centroids[i, :] = (np.sum(X[indices, :], axis=1) / len(indices[0]))
                                                .ravel()
    return centroids
```

3.2.6　K-means 迭代算法

传入参数数据集 X、初始化后的聚类中心、迭代次数。

每次迭代使用之前的两个函数，先计算出 idx 索引数组，再计算出聚类中心。

返回迭代多次后的 idx 和 cendroids。

```
def k_means(X, initial_centroids, max_iters):
    """
    :param X:
    :param initial_centroids: 初始聚类中心
    :param max_iters: 迭代次数
    :return:
    """
    m, n = X.shape
    k = initial_centroids.shape[0]
    idx = np.zeros(m)
    centroids_all = []
    centroids_all.append(initial_centroids)
    centroids = initial_centroids
    for i in range(max_iters):
        idx = get_near_cluster_centroids(X, centroids)
        centroids = compute_centroids(X, idx, k)
        centroids_all.append(centroids)
    return idx, np.array(centroids_all),centroids
```

3.2.7　调用执行算法

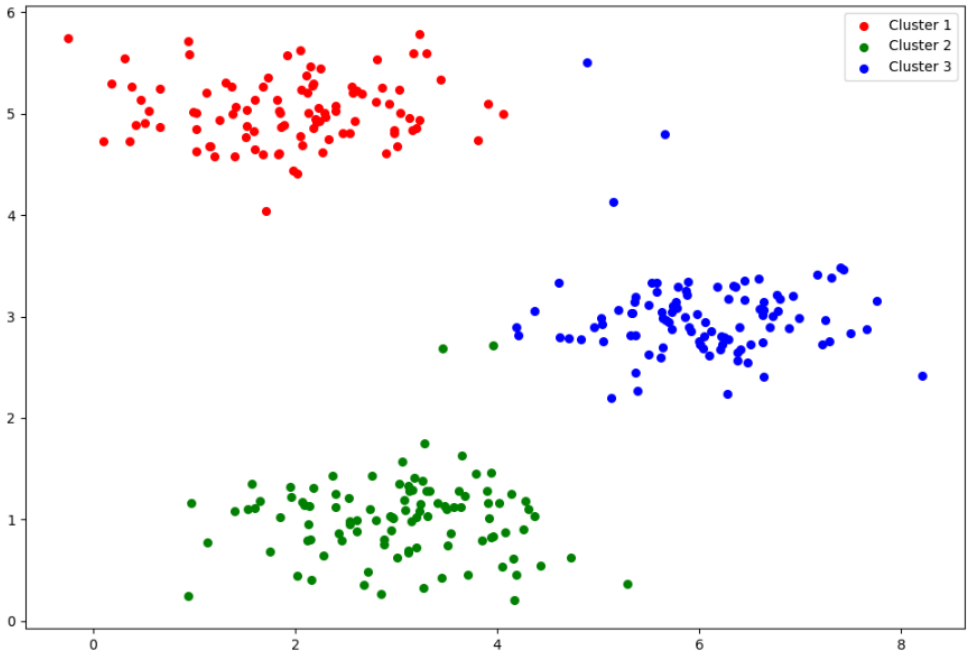
首先，我们初始化 centroids 为：

```
centroids_0=np.array([[3, 3], [6, 2], [8, 5]])
```

接着我们画出聚类图像：

```
def plot_classify_data(X,idx):
    cluster1 = X[np.where(idx == 0)[0], :]
    cluster2 = X[np.where(idx == 1)[0], :]
    cluster3 = X[np.where(idx == 2)[0], :]
    fig, ax = plt.subplots(figsize=(12, 8))
    ax.scatter(cluster1[:, 0], cluster1[:, 1], s=30, color='r', label='
                                        Cluster 1')
    ax.scatter(cluster2[:, 0], cluster2[:, 1], s=30, color='g', label='
                                        Cluster 2')
    ax.scatter(cluster3[:, 0], cluster3[:, 1], s=30, color='b', label='
                                        Cluster 3')
    ax.legend()
    plt.show()
```

```
idx, centroids_all,centroids=k_means(X,centroids_0,10)
plot_classify_data(X,idx)
```



我们想看一下聚类中心的变化情况：

```
def plot_data(centroids_all,idx):
    plt.figure(figsize=(12,8))
    cm=mpl.colors.ListedColormap(['r','g','b'])
    plt.scatter(dataset['X1'],dataset['X2'], c=idx,cmap=cm)
    plt.plot(centroids_all[:,:,0],centroids_all[:,:,1], 'kx--')
    plt.show()
```

结果为：

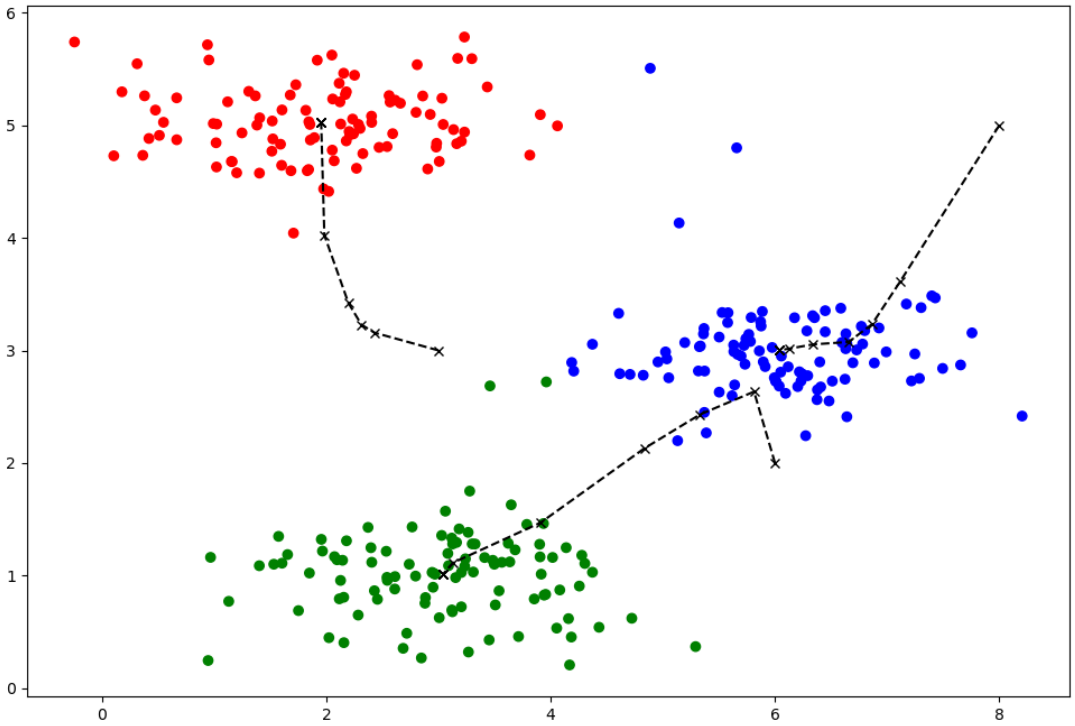
综上所述，结果清晰明了。

在以上操作中，使用的初始聚类中心均为自己找的聚类中心，直接以数组形式进行定义。

实际上，聚类中心应在数据集 X 中随机选择 k 个样本带你作为聚类中心。

3.2.8　设计初始化聚类中心函数

```
def init_centroids(X, k):
    m, n = X.shape
    init_centroids = np.zeros((k, n))
    idx = np.random.randint(0, m, k)
    for i in range(k):
        init_centroids[i, :] = X[idx[i], :]
    return init_centroids
```

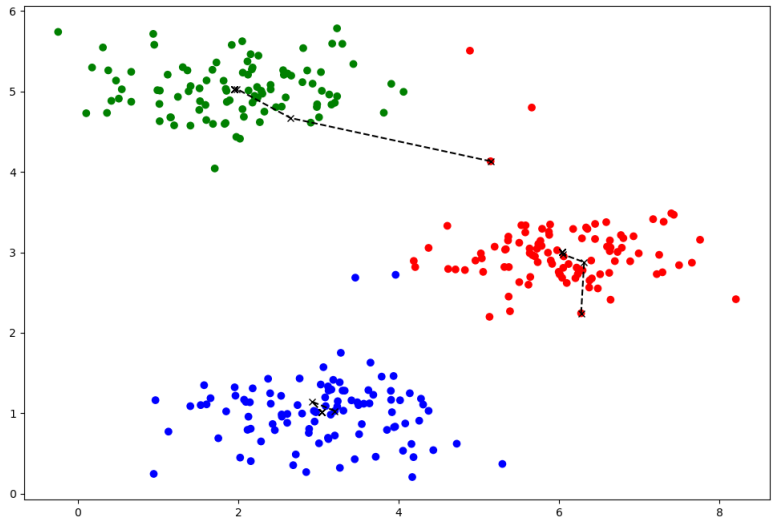


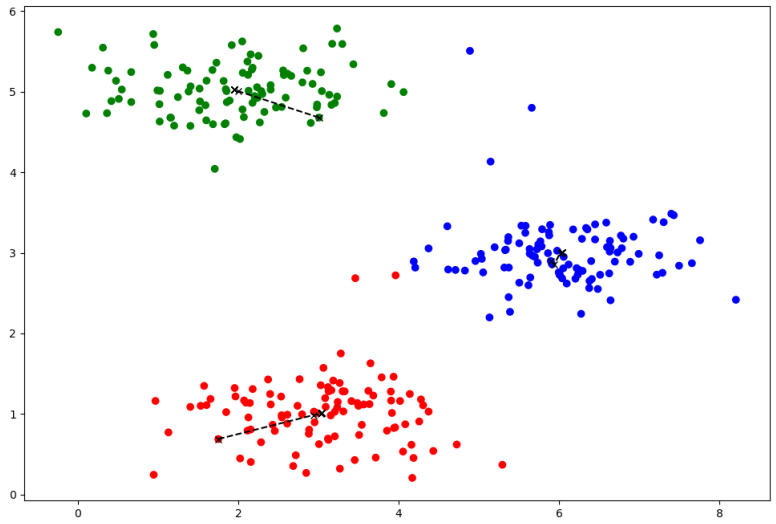
我们展示一下结果：

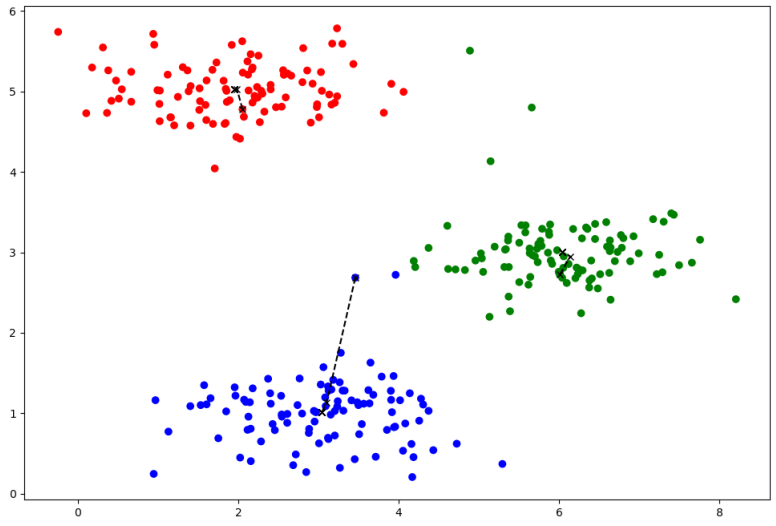
```
[[4.06069132 4.99503862]
 [2.55983064 5.26397756]
 [0.47647355 5.13535977]]
```

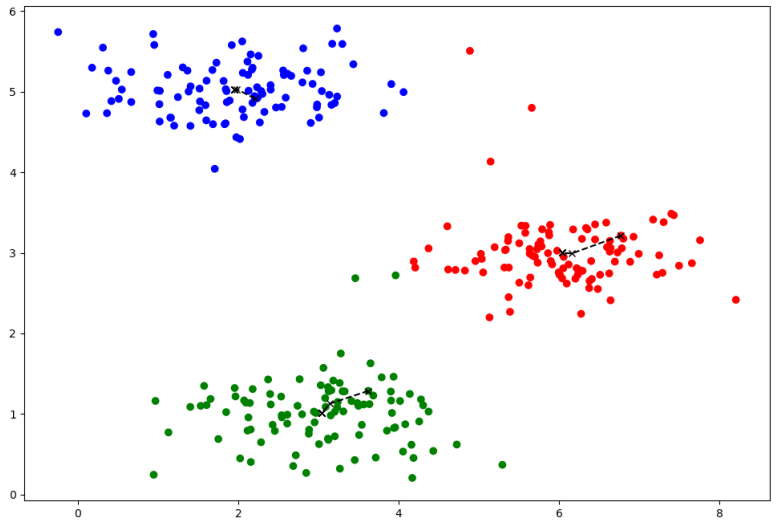
可视化图像为：

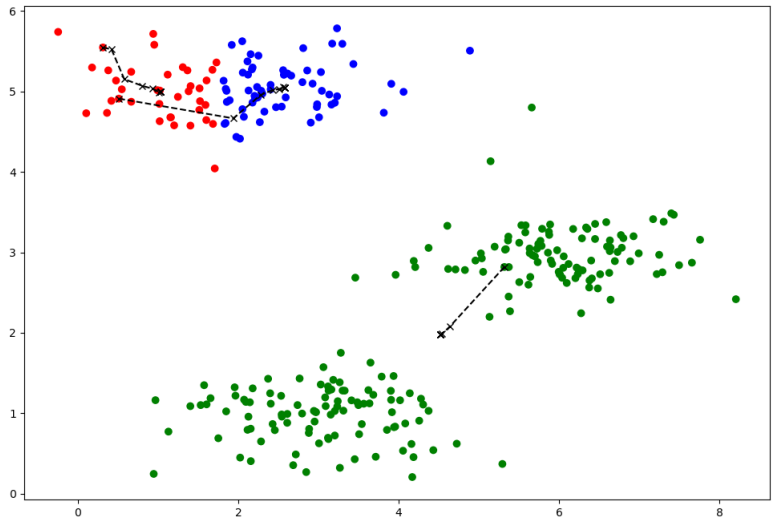
```
for i in range(5):
    idx, centroids_all, centroids = k_means(X, init_centroids(X, 3), 10)
    plot_data(centroids_all,idx)
```


0
1
2
3
4
5
6
0
2
4
6
8


0
1
2
3
4
5
6
0
2
4
6
8


0
1
2
3
4
5
6
0
2
4
6
8


0
1
2
3
4
5
6
0
2
4
6
8


0
1
2
3
4
5
6
0
2
4
6
8

3.3　k-means 算法对图片颜色进行聚类

RGB 图像，每个像素点值范围为 [0-255]

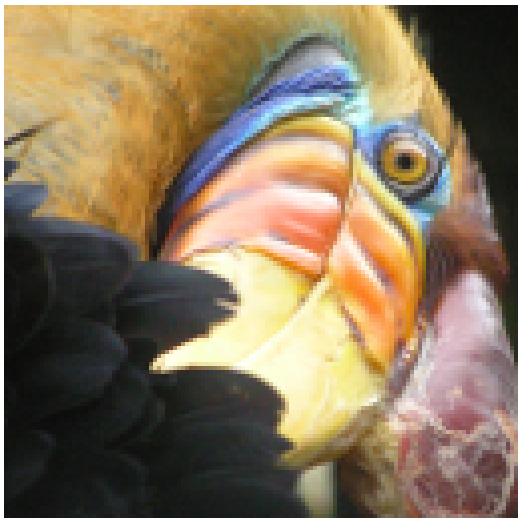
首先我们导入我们可能用到的包:

```
import matplotlib.pyplot as plt
from scipy.io import loadmat
from numpy import *
from IPython.display import Image
```

导入相应的 RGB 图像:

```
def load_picture():
    path='./data/bird_small.png'
    image=plt.imread(path)
    plt.imshow(image)
    plt.show()
```

我们看一下图片：



3.3.1　数据的归一化

这一步是相当有必要的

我们归一化的实现流程如下：

```
def normalizing(A):
    A=A/255.
    A_new=reshape(A,(-1,3))
    return A_new
```

我们看一下归一化后的数据集：

```
[[0.85882353 0.70588235 0.40392157]
 [0.90196078 0.7254902  0.45490196]
 [0.88627451 0.72941176 0.43137255]
 ...
 [0.25490196 0.16862745 0.15294118]
 [0.22745098 0.14509804 0.14901961]
 [0.20392157 0.15294118 0.13333333]]

(16384, 3)
```

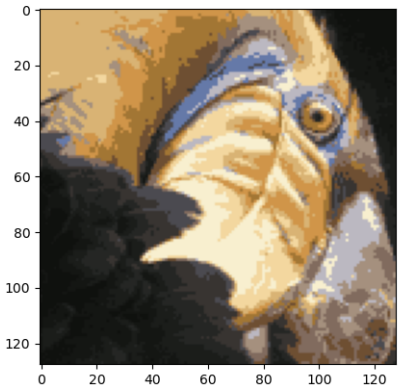
这里可以很明显的看到，数据集均变为了 0-1 之间，并且把三维数组转换成了二维数组。

至此，我们数据集的处理过程已经结束，我们给出 k-means 算法，过程与之前相同。

3.3.2　绘制压缩后的图像

```
def reduce_picture():
    initial_centroids = init_centroids(A_new, 16)
    idx, centroids = k_means(A_new, initial_centroids, 10)
    idx_1 = get_near_cluster_centroids(A_new, centroids)
    A_recovered = centroids[idx_1.astype(int), :]
    A_recovered_1 = reshape(A_recovered, (A.shape[0], A.shape[1], A.shape[2
                                          ]))
    plt.imshow(A_recovered_1)
    plt.show()
```

我们结果为：



第四章　PCA 算法

4.1　使用 PCA 对二维数据降维

4.1.1　PCA 算法

PCA 算法为主成分分析算法，在数据集中找到“主成分”，可以用于压缩数据维度。

我们将首先通过一个 2D 数据集进行实验，以获得关于 PCA 如何工作的直观感受，然后在一个更大的图像数据集上使用它。

PCA 算法的好处如下：

1. 使得数据集更易使用
2. 降低算法的计算开销
3. 去除噪声
4. 使得结果更易理解

4.1.2　PCA 实战项目

导入需要的包

```
import matplotlib.pyplot as plt
import matplotlib as mpl
from scipy.io import loadmat
from numpy import *
import pandas as pd
```

导入数据集

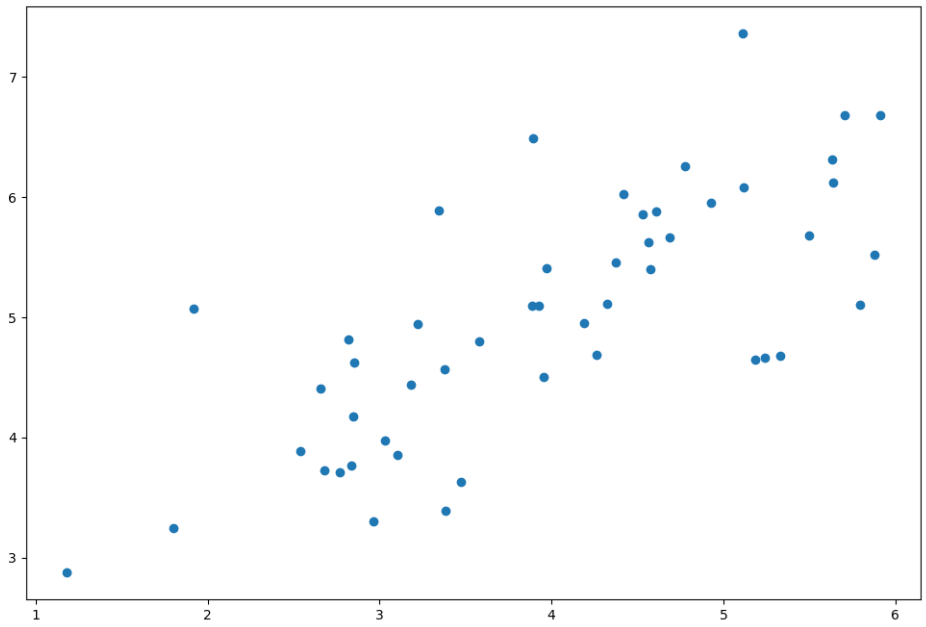
```
def load_dataset():
    path='./data/ex7data1.mat'
    two_dimension_data=loadmat(path)
```

```
    X=two_dimension_data['X']
    return X
```

绘制散点图

```
def plot_scatter(X):
    plt.figure(figsize=(12,8))
    plt.scatter(X[:,0],X[:,1])
    plt.show()
```

我们看一下可视化后的散点图：



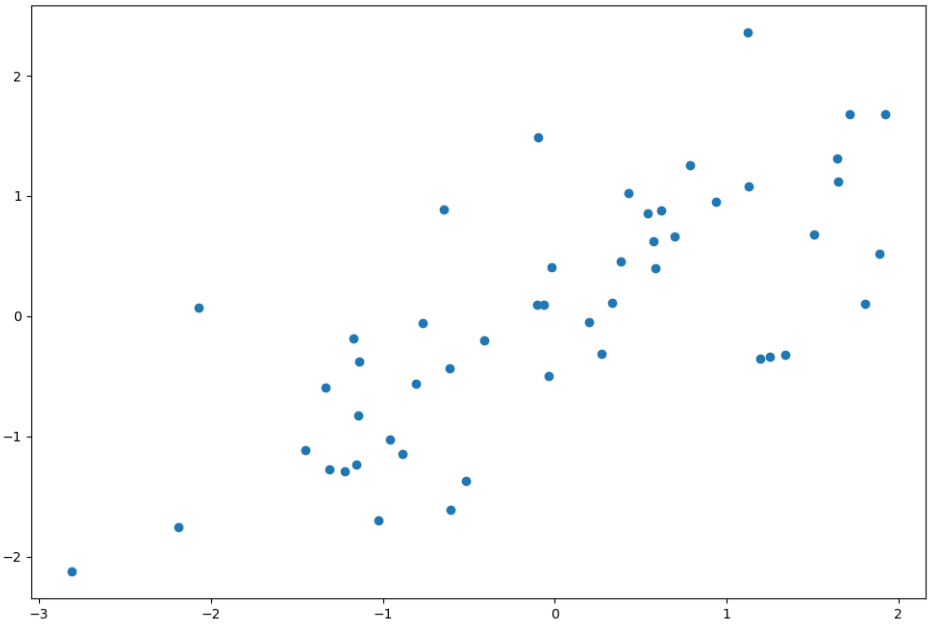
去均值化

$$X_{demean} = X - means$$

我们返回的结果是去均值化后的数据，并生成散点图：

```
def demean(X):
    X_demean=(X-mean(X,axis=0))
    plt.figure(figsize=(12,8))
```

```
    plt.scatter(X_demean[:,0],X_demean[:,1])
    plt.show()
    return X_demean
```



计算数据的协方差矩阵

$$C = \frac{1}{m} X^T X$$

```
def sigma_matrix(X_demean):
    sigma=(X_demean.T @ X_demean)/X_demean.shape[0]
    return sigma
```

```
[[1.34852518 0.86535019]
 [0.86535019 1.02641621]]
```

计算特征值、特征向量

```
def usv(sigma):
    u,s,v=linalg.svd(sigma)
    return u,s,v
```

对数据进行降维

对数据进行降维，降维后得到 Z，在二维数据中 Z 数据为一条直线点（一维）。

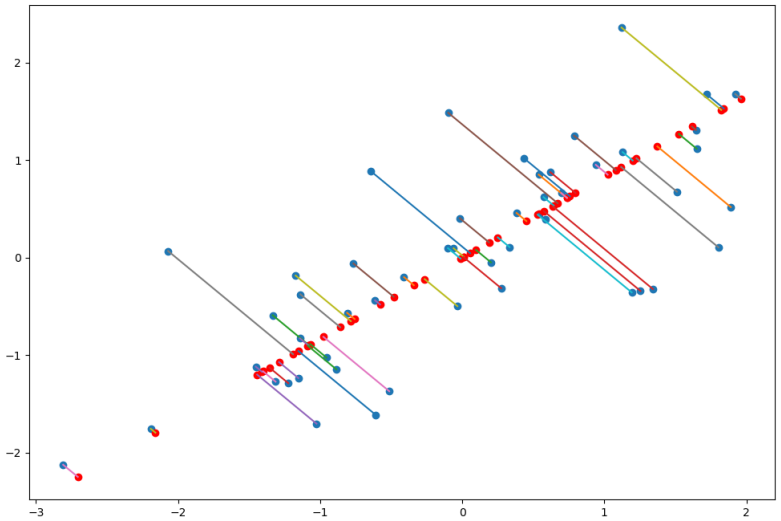
```
def project_data(X_demean, u, k):
    u_reduced = u[:,:k]
    z=dot(X_demean, u_reduced)
    return z
```

还原数据

```
def recover_data(z, u, k):
    u_reduced = u[:,:k]
    X_recover=dot(z, u_reduced.T)
    return X_recover
```

数据可视化

```
fig, ax = plt.subplots(figsize=(12, 8))
ax.scatter(X_demean[:,0],X_demean[:,1])
ax.scatter(list(X_recover[:, 0]), list(X_recover[:, 1]),c='r')
ax.plot([X_demean[:,0],list(X_recover[:, 0])], [X_demean[:,1],list(
                                        X_recover[:, 1])])
plt.show()
```



4.2 使用 PCA 对图像降维

在这部分练习中，我们将学习人脸图像上运行 PCA，看看如何在实践中使用它来减少维度。

4.2.1 导入需要用到的包

```
from numpy import *
from scipy.io import loadmat
import matplotlib.pyplot as plt
```

4.2.2 导入数据

```
faces_data = loadmat('data/ex7faces.mat')
print(faces_data)
X=faces_data['X']
print(X.shape)
```

```
(5000, 1024)
```

说明我们的数据集有 5000 个样本，每个样本有 1024 个特征。

4.2.3 可视化

我们可视化一下前 100 张人脸图像：

```
def plot_100_image(X):
    fig,ax=plt.subplots(nrows=10,ncols=10,figsize=(10,10))
    for c in range(10):
        for r in range(10):
            ax[c,r].imshow(X[10*c+r].reshape(32,32).T,cmap='Greys_r')
            ax[c,r].set_xticks([])
            ax[c,r].set_yticks([])
    plt.show()

plot_100_image(X)
```

结果如下图所示：

接下来我们应用 PCA 算法的步骤与之前在二维数据集上的步骤一致：

1. 去均值化
2. 计算协方差矩阵
3. 计算特征值和特征向量

我们接下来还原数据，这里选择只保留 100 个特征：

```
def recover_data(z, u, k):
    u_reduced = u[:,:k]
    X_recover=dot(z, u_reduced.T)
    return X_recover
X_recover=recover_data(z,u,100)
```

我们看一下最后降维后的图像：

我们对比两张图片，可以很明显的看出，第二张图片保留的特征较少，已经导致脸部有些模糊。

第五章　源代码

5.1　K-means 算法

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.io import loadmat
import matplotlib as mpl

def load_dataset():
    path='./data/ex7data2.mat'
    # 字典格式 : <class 'dict'>
    data=loadmat(path)
    # data.keys() : dict_keys(['__header__', '__version__', '__globals__',
                                            'X'])
    dataset = pd.DataFrame(data.get('X'), columns=['X1', 'X2'])
    return data,dataset

def plot_scatter():
    data,dataset=load_dataset()
    plt.figure(figsize=(12,8))
    plt.scatter(dataset['X1'],dataset['X2'],cmap=['b'])
    plt.show()

def get_near_cluster_centroids(X,centroids):
    m = X.shape[0] #数据的行数
    k = centroids.shape[0] #聚类中心的行数, 即个数
    idx = np.zeros(m) # 一维向量idx, 大小为数据集中的点的个数, 用于保存每一
                                            个X的数据点最小距离点的是哪个聚类
                                            中心

    for i in range(m):
```

```
        min_distance = 1000000
        for j in range(k):
            distance = np.sum((X[i, :] - centroids[j, :]) ** 2) # 计算数据
                                                    点到聚类中心距离代价的公
                                                    式，X中每个点都要和每个聚
                                                    类中心计算
            if distance < min_distance:
                min_distance = distance
                idx[i] = j # idx中索引为i，表示第i个X数据集中的数据点距离最
                                                        近的聚类中心的索引
    return idx # 返回的是X数据集中每个数据点距离最近的聚类中心

def compute_centroids(X, idx, k):
    m, n = X.shape
    centroids = np.zeros((k, n)) # 初始化为k行n列的二维数组，值均为0，k为聚
                                        类中心个数，n为数据列数
    for i in range(k):
        indices = np.where(idx == i) # 输出的是索引位置
        centroids[i, :] = (np.sum(X[indices, :], axis=1) / len(indices[0]))
                                                .ravel()
    return centroids

def k_means(X, initial_centroids, max_iters):
    m, n = X.shape
    k = initial_centroids.shape[0]
    idx = np.zeros(m)
    centroids_all = []
    centroids_all.append(initial_centroids)
    centroids = initial_centroids
    for i in range(max_iters):
        idx = get_near_cluster_centroids(X, centroids)
        centroids = compute_centroids(X, idx, k)
        centroids_all.append(centroids)
    return idx, np.array(centroids_all),centroids

def plot_data(centroids_all,idx):
    plt.figure(figsize=(12,8))
    cm=mpl.colors.ListedColormap(['r','g','b'])
    plt.scatter(dataset['X1'],dataset['X2'], c=idx,cmap=cm)
```

```
    plt.plot(centroids_all[:,:,0],centroids_all[:,:,1], 'kx--')
    plt.show()

def plot_classify_data(X,idx):
    cluster1 = X[np.where(idx == 0)[0], :]
    cluster2 = X[np.where(idx == 1)[0], :]
    cluster3 = X[np.where(idx == 2)[0], :]
    fig, ax = plt.subplots(figsize=(12, 8))
    ax.scatter(cluster1[:, 0], cluster1[:, 1], s=30, color='r', label='
                                        Cluster 1')
    ax.scatter(cluster2[:, 0], cluster2[:, 1], s=30, color='g', label='
                                        Cluster 2')
    ax.scatter(cluster3[:, 0], cluster3[:, 1], s=30, color='b', label='
                                        Cluster 3')
    ax.legend()
    plt.show()


def init_centroids(X, k):
    m, n = X.shape
    init_centroids = np.zeros((k, n))
    idx = np.random.randint(0, m, k)
    for i in range(k):
        init_centroids[i, :] = X[idx[i], :]
    return init_centroids

if __name__=='__main__':
    data,dataset = load_dataset()
    plot_scatter()
    X=data.get('X')
    centroids_0=np.array([[3, 3], [6, 2], [8, 5]])
    idx, centroids_all,centroids=k_means(X,centroids_0,10)
    plot_data(centroids_all, idx)
    plot_classify_data(X,idx)
    print(init_centroids(X, 3))
    for i in range(5):
        idx, centroids_all, centroids = k_means(X, init_centroids(X, 3), 10
                                                )
        plot_data(centroids_all,idx)
```

5.2　K-means 算法实现图像压缩

```
import matplotlib.pyplot as plt
from scipy.io import loadmat
from numpy import *
from IPython.display import Image
def load_picture():
    path='./data/bird_small.png'
    image=plt.imread(path)
    plt.imshow(image)
    plt.show()

def load_data():
    path='./data/bird_small.mat'
    data=loadmat(path)
    return data

def normalizing(A):
    A=A/255.
    A_new=reshape(A,(-1,3))
    return A_new

def get_near_cluster_centroids(X,centroids):
    m = X.shape[0] #数据的行数
    k = centroids.shape[0] #聚类中心的行数，即个数
    idx = zeros(m) # 一维向量idx，大小为数据集中的点的个数，用于保存每一个X
                                        的数据点最小距离点的是哪个聚类中
                                        心

    for i in range(m):
        min_distance = 1000000
        for j in range(k):
            distance = sum((X[i, :] - centroids[j, :]) ** 2) # 计算数据点到
                                                聚类中心距离代价的公式，X
                                                中每个点都要和每个聚类中
                                                心计算

            if distance < min_distance:
                min_distance = distance
                idx[i] = j # idx中索引为i，表示第i个X数据集中的数据点距离最
                                                近的聚类中心的索引
```

```
    return idx # 返回的是X数据集中每个数据点距离最近的聚类中心

def compute_centroids(X, idx, k):
    m, n = X.shape
    centroids = zeros((k, n)) # 初始化为k行n列的二维数组，值均为0，k为聚类
                                        中心个数，n为数据列数
    for i in range(k):
        indices = where(idx == i) # 输出的是索引位置
        centroids[i, :] = (sum(X[indices, :], axis=1) / len(indices[0])).
                                            ravel()
    return centroids

def k_means(A_1,initial_centroids,max_iters):
    m,n=A_1.shape
    k = initial_centroids.shape[0]
    idx = zeros(m)
    centroids = initial_centroids
    for i in range(max_iters):
        idx = get_near_cluster_centroids(A_1, centroids)
        centroids = compute_centroids(A_1, idx, k)
    return idx, centroids

def init_centroids(X, k):
    m, n = X.shape
    init_centroids = zeros((k, n))
    idx = random.randint(0, m, k)
    for i in range(k):
        init_centroids[i, :] = X[idx[i], :]
    return init_centroids

def reduce_picture():
    initial_centroids = init_centroids(A_new, 16)
    idx, centroids = k_means(A_new, initial_centroids, 10)
    idx_1 = get_near_cluster_centroids(A_new, centroids)
    A_recovered = centroids[idx_1.astype(int), :]
    A_recovered_1 = reshape(A_recovered, (A.shape[0], A.shape[1], A.shape[2
                                        ]))
    plt.imshow(A_recovered_1)
    plt.show()
```

```
if __name__=='__main__':
    load_picture()
    data=load_data()
    print(data.keys())
    A=data['A']
    print(A.shape)
    A_new=normalizing(A)
    print(A_new)
    print(A_new.shape)
    reduce_picture()
```

5.3　PCA 对二维数据降维

```
import matplotlib.pyplot as plt
import matplotlib as mpl
from scipy.io import loadmat
from numpy import *
import pandas as pd
"""
PCA对二维数据进行降维
"""

def load_dataset():
    path='./data/ex7data1.mat'
    two_dimension_data=loadmat(path)
    X=two_dimension_data['X']
    return X

def plot_scatter(X):
    plt.figure(figsize=(12,8))
    cm=mpl.colors.ListedColormap(['blue'])
    plt.scatter(X[:,0],X[:,1],cmap=cm)
    plt.show()

"""
对X去均值,并可视化图像
"""
```

```
def demean(X):
    X_demean=(X-mean(X,axis=0))
    plt.figure(figsize=(12,8))
    plt.scatter(X_demean[:,0],X_demean[:,1])
    plt.show()
    return X_demean

"""
计算协方差矩阵
"""
def sigma_matrix(X_demean):
    sigma=(X_demean.T @ X_demean)/X_demean.shape[0]
    return sigma

"""
计算特征值、特征向量
"""
def usv(sigma):
    u,s,v=linalg.svd(sigma)
    return u,s,v

def project_data(X_demean, u, k):
    u_reduced = u[:,:k]
    z=dot(X_demean, u_reduced)
    return z

def recover_data(z, u, k):
    u_reduced = u[:,:k]
    X_recover=dot(z, u_reduced.T)
    return X_recover

if __name__=='__main__':
    X=load_dataset()
    print(X)
    print('=='*50)
    print(X.shape)
    print('=='*50)
    plot_scatter(X)
    X_demean=demean(X)
```

```
    sigma=sigma_matrix(X_demean)
    print(sigma)
    print('=='*50)
    u, s, v=usv(sigma)
    print(u)
    print('=='*50)
    print(s)
    print('=='*50)
    print(v)
    print('=='*50)
    z = project_data(X_demean, u, 1)
    print(z)
    print('=='*50)
    X_recover = recover_data(z, u, 1)
    print(X_recover)
    print('=='*50)
    fig, ax = plt.subplots(figsize=(12, 8))
    ax.scatter(X_demean[:,0],X_demean[:,1])
    ax.scatter(list(X_recover[:, 0]), list(X_recover[:, 1]),c='r')
    ax.plot([X_demean[:,0],list(X_recover[:, 0])],[X_demean[:,1],list(
                                            X_recover[:, 1])])
    plt.show()
```

5.4　PCA 对图像降维

```
from numpy import *
from scipy.io import loadmat
import matplotlib.pyplot as plt
faces_data = loadmat('data/ex7faces.mat')
print(faces_data)
X=faces_data['X']
print(X[0].shape)
print(X.shape)

def plot_100_image(X):
    fig,ax=plt.subplots(nrows=10,ncols=10,figsize=(10,10))
    for c in range(10):
        for r in range(10):
```

```
            ax[c,r].imshow(X[10*c+r].reshape(32,32).T,cmap='Greys_r')
            ax[c,r].set_xticks([])
            ax[c,r].set_yticks([])
    plt.show()

plot_100_image(X)

def reduce_mean(X):
    X_reduce_mean=X-X.mean(axis=0)
    return X_reduce_mean
X_reduce_mean=reduce_mean(X)

def sigma_matrix(X_reduce_mean):
    sigma=(X_reduce_mean.T @ X_reduce_mean)/X_reduce_mean.shape[0]
    return sigma
sigma=sigma_matrix(X_reduce_mean)

def usv(sigma):
    u,s,v=linalg.svd(sigma)
    return u,s,v
u,s,v=usv(sigma)
print(u)

def project_data(X_reduce_mean, u, k):
    u_reduced = u[:,:k]
    z=dot(X_reduce_mean, u_reduced)
    return z

def recover_data(z, u, k):
    u_reduced = u[:,:k]
    X_recover=dot(z, u_reduced.T)
    return X_recover

z = project_data(X_reduce_mean, u, 100)

X_recover=recover_data(z,u,100)
plot_100_image(X_recover)
```