近邻成分分析（Neighbourhood Component Analysis，NCA）是由Jacob Goldberger和Geoff Hinton等大佬们在2005年发表的一项工作，属于度量学习（Metric Learning）和降维（Dimension Reduction）领域。其关键点可以概括为：任务是KNN Classification，样本相似度计算方法基于马氏距离（Mahalanobis Distance），参数选择方法为留一验证法（Leave One Out）。最后模型可以学习样本的低维嵌入表示（Embedding），既属于度量学习范畴，又是降维的过程。

近邻成分分析在度量学习和降维领域展现了很强大的本领，也为后来很多降维的工作起了引导性的作用。本文将详细介绍NCA的思想和数学形式，然后也会围绕如何快速实现NCA进行介绍，利用Python3.6实现，最后给出一些自己实验得到的结果，以便更加了解NCA的能力与缺陷。

下面是本文的主要内容：

NCA的主要思想、数学形式及求解
NCA的代码实现：四种方法
NCA在一些数据集上的表现

1 NCA的主要思想、数学形式及求解

1.1 NCA主要思想

首先，NCA中的Neighbourhood是指近邻，可以理解为相似的样本（这里默认距离小代表相似度高）。在降维和度量学习任务中，“邻居”是一个很重要的思想，很多算法都是围绕它进行推导的。NCA基于KNN Classification，所以，NCA是一个有监督的算法，必须给定数据和类别标签才可以进行训练。

如果想理解NCA，必须要先理解NCA的三个关键点，分别是：Stochastic KNN、Metric Learning和Leave one out。

NCA是有监督的，基于分类器Stochastic KNN；
NCA衡量近邻相似度的方法借助了Metric Learning；
NCA在训练过程中调参采用的是Leave one out验证法；

注意，虽然NCA借助使用了分类器KNN，但是NCA并不是做分类问题的，而是度量学习和降维。下面就用数学公式来形式化表述NCA做的事情。

1.2 NCA数学形式

给定数据样本 $\{(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)\}$ ， $x_i \in R^d$ 代表第 i 个数据样本，$y_i \in R$ 代表样本标签。考虑在KNN中，使用Leave one out交叉验证法，假设现在要预测第 i 个样本的标签，那么我们可以这么做：

计算样本 i 和其余所有样本之间的欧氏距离， $d_{ij} = ||x_i - x_j||_2$
选择距离最小的 k 个， $d_{ij_1}, d_{ij_2}, \cdots, d_{ij_k}$
利用这 k 个样本的标签进行投票得到预测结果， $Vote(y_{j_1}, y_{j_2}, \cdots, y_{j_k})$

上述过程是一般的KNN过程，那么这里先引入Stochastic 1-NN改进办法：

计算样本 i 的近邻分布：

$$p_{ij}=\frac{\exp\left(||x_i-x_j||_2^2\right)}{\sum_{k\neq i}\exp\left(||x_i-x_k||_2^2\right)},p_{ii}=0$$

2. 根据概率分布 $p_{ij},j\neq i,j\in[1,n]$ 采样得到一个样本 k，然后将第 i 个数据的样本预测为 y_k

从上面可以看出，第 i 个样本的真实标记为 y_i ，假如 $y_k=y_i$ ，那么预测正确。记 $C_i=\{j|y_j=y_i\}$ 表示与第 i 个样本类别一样的下标集合，那么利用上述 Stochastic 1-NN正确预测第 i 个样本标签的概率为：

$$p_i=\sum_{j\in C_i}p_{ij}$$

那么，对于所有的样本，优化目标为：

$$f=\sum_{i=1}^{n}p_i=\sum_{i=1}^{n}\sum_{j\in C_i}p_{ij}$$

到这里，不免产生一个问题，Stochastic 1-NN里面没有什么参数可以学习，给定样本集合最终的优化目标 f 就是一个确定值，目前为止还没真正构成一个优化问题。再鉴于使用欧氏距离计算会导致计算量特别大，并且维度空间特别高，这里再引入度量学习的思想，引入可学习的马氏距离。在进一步介绍之前，先看一下马氏距离的定义以及度量学习的定义。

1.3 马氏距离和度量学习

数据样本矩阵表示为 $X=[x_1;x_2;\cdots;x_n]^T$ ，这是以样本角度表示的，还可以以特征角度表示为 $X=[f_1;f_2;\cdots;f_d]$ 。假设样本间的协方差矩阵为 $S\in R^{d\times d}$ ，那么有：

$$S_{ij}=Cov(i,j)=\frac{1}{n}(f_i-mean(f_i))^T(f_j-mean(f_j))$$

马氏距离的定义为：

$$d(x_i,x_j)=\sqrt{(x_i-x_j)^TS^{-1}(x_i-x_j)}$$

这里简单介绍一下马氏距离的优点：

相当于是对数据中心化和标准化，数据中心化和标准化后的马氏距离与原来的马氏距离一致
由于是相当于中心化和标准化，所以不受特征单位的影响
可以推导出可学习的马氏距离，是度量学习的基础
考虑样本总体特性，一般来说，两个样本放入不同的总体，计算得到的马氏距离不相等

度量学习是基于可学习的马氏距离，也称伪（pseudo）马氏距离：

$$d_M(x_i,x_j)=\sqrt{(x_i-x_j)^TM(x_i-x_j)},\quad M\in S_+^d$$

其中 M 是半正定矩阵（Positive Semi-Definite, PSD），是可以学习的参数，称为度量。度量学习的目标就是通过一些约束（比如，must-link pair和must-not-link pair）进行优化矩阵 M，从而学习到一个距离度量。

由于 M 是半正定矩阵，那么存在 $A\in R^{k\times d},k<d,k\geq rank(M)$，满足 $M=A^TA$，那么：

$$d_M(x_i,x_j)=\sqrt{(x_i-x_j)^TA^TA(x_i-x_j)}=||Ax_i-Ax_j||_2$$

可以看出马氏距离做的事情，可以理解为先把数据降维到低维空间，然后再求欧氏距离。

1.4 NCA完整表达

下面回到NCA，通过引入距离度量得到下面的过程：

令 $A\in R^{k\times d}$ 为参数
计算样本 i 的近邻分布：

$$p_{ij}=\frac{\exp\left(||Ax_i-Ax_j||_2^2\right)}{\sum_{k\neq i}\exp\left(||Ax_i-Ax_k||_2^2\right)},p_{ii}=0$$

3. 优化目标：

$$f(A)=\sum_{i=1}^n p_i=\sum_{i=1}^n\sum_{j\in C_i}p_{ij}$$

到这儿，就介绍完了NCA的主要数学形式，再总结一下：

NCA是有监督的，需要提供样本标签
借鉴度量学习引入距离度量 A 来计算样本间距离
目标是使得Stochastic 1-NN的准确率最高
参数学习过程使用了Leave one out交叉验证的方法

1.5 NCA求解

那么下面的问题就是如何求解NCA？论文中直接使用梯度下降法，但由于目标不是凸的，所以只能得到局部最优解。至于目标为什么不是凸的，是因为：将马氏距离下的度量学习转换为基于样本投影矩阵的度量学习，会造成问题非凸。

下面来推导一下NCA优化目标的梯度，记 $g_{ij}=\exp\left(-\|Ax_i-Ax_j\|^2\right)$ ，那么有：

$$p_{ij}=\frac{g_{ij}}{\sum_{k\neq i}g_{ik}}$$

由：

$$\frac{\partial x^TD^TDx}{\partial D}=Dxx^T$$

得到：

$$\frac{\partial g_{ij}}{\partial A}=-2g_{ij}A\left(x_i-x_j\right)\left(x_i-x_j\right)^T$$

那么：

$$\frac{\partial p_{ij}}{\partial A}=\frac{1}{\left(\sum_{k\neq i}g_{ik}\right)^2}\left(\frac{\partial g_{ij}}{\partial A}\sum_{k\neq i}g_{ik}-g_{ij}\sum_{k\neq i}\frac{\partial g_{ik}}{\partial A}\right)$$

前半部分为：

$$\frac{1}{\left(\sum_{k\neq i}g_{ik}\right)^2}\frac{\partial g_{ij}}{\partial A}\sum_{k\neq i}g_{ik}=\frac{-2g_{ij}A\left(x_i-x_j\right)\left(x_i-x_j\right)^T}{\sum_{k\neq i}g_{ik}}=-2p_{ij}A\left(x_i-x_j\right)\left(x_i-x_j\right)^T$$

后半部分为：

$$\begin{aligned}\frac{1}{\left(\sum_{k\neq i}g_{ik}\right)^2}g_{ij}\sum_{k\neq i}\frac{\partial g_{ik}}{\partial A}&=p_{ij}\frac{-2\sum_{k\neq i}g_{ik}A\left(x_i-x_k\right)\left(x_i-x_k\right)^T}{\sum_{s\neq i}g_{is}}\\&=-2p_{ij}A\left(\sum_{k\neq i}p_{ik}\left(x_i-x_k\right)\left(x_i-x_k\right)^T\right)\end{aligned}$$

所以：

$$\frac{\partial p_{ij}}{\partial A}=-2p_{ij}A\left(\left(x_i-x_j\right)\left(x_i-x_j\right)^T-\sum_{k\neq i}p_{ik}\left(x_i-x_k\right)\left(x_i-x_k\right)^T\right)$$

目标梯度为：

$$\frac{\partial f}{\partial A}=\sum_{i=1}^{n}\sum_{j\in C_i}\frac{\partial p_{ij}}{\partial A}=-2A\sum_{i}\sum_{j\in C_i}p_{ij}\left((x_i-x_j)(x_i-x_j)^T-\sum_{k\neq i}p_{ik}(x_i-x_k)(x_i-x_k)^T\right)$$

化简为，这里记下面的梯度表达为“梯度#1”：

$$\frac{\partial f}{\partial A}=2A\sum_{i}\left(p_i\sum_{k\neq i}p_{ik}(x_i-x_k)(x_i-x_k)^T-\sum_{j\in C_i}p_{ij}(x_i-x_j)(x_i-x_j)^T\right)$$

求出梯度之后，利用梯度下降法优化即可得到NCA的训练结果。训练得到的映射矩阵 A 可以用来对数据进行降维，以便于降低数据维度。

这里再稍微提一下，对上面的梯度进行变形，以便于后续代码实现，再进一步化简：

$$\frac{\partial f}{\partial A}=2\sum_{i}\sum_{j}\left(p_ip_{ij}-pmask_{ij}\right)(Ax_i-Ax_j)(x_i-x_j)^T$$

其中，$pmask_{ij}=p_{ij},if\ j\in C_i,else\ 0$ 。记 $W_{ij}=p_ip_{ij}-pmask_{ij}$ ，有：

$$\frac{\partial f}{\partial A}=2\sum_{i}\sum_{j}W_{ij}(Ax_i-Ax_j)(x_i-x_j)^T=2\sum_{i}\sum_{j}W_{ij}(Ax_ix_i^T+Ax_jx_j^T-Ax_ix_j^T-Ax_jx_i^T)$$
$$=2(XA^T)^T\left(diag\left(sum(W,axis=0)\right)+diag\left(sum(W^T,axis=0)\right)-W-W^T\right)X$$

上式第三个等号是因为：

$$\sum_{i}\sum_{j}W_{ij}x_ix_j^T=X^TWX$$
$$\Rightarrow$$
$$\sum_{i}\sum_{j}W_{ij}Ax_ix_j^T=(XA^T)^TWX$$

又由：

$$\sum_{j}W_{ij}=\sum_{j}(p_ip_{ij}-pmask_{ij})=p_i\sum_{j}p_{ij}-\sum_{j}pmask_{ij}=p_i-\sum_{j\in C_i}p_{ij}=p_i-p_i=0$$

所以 $sum(W^T,axis=0)=sum(W,axis=1)=[0,0,\cdots,0]$ 。这里化简得到“梯度#2”：

$$\frac{\partial f}{\partial A}=2(XA^T)^T\left(diag\left(sum(W,axis=0)\right)-W-W^T\right)X$$

注意：对比两个梯度表达形式“#1”与“#2”，后面代码实现会利用。不同的形式对应的实现在运行时间与空间上差别很大。

2 NCA的代码实现：四种方法

下面介绍如何利用Python3.6来实现NCA，主要是如何快速高效地求梯度。先回顾一下两个梯度形式“#1”与“#2”：

$$\frac{\partial f}{\partial A} = 2A\sum_i \left(p_i \sum_{k \neq i} p_{ik}(x_i - x_k)(x_i - x_k)^T - \sum_{j \in C_i} p_{ij}(x_i - x_j)(x_i - x_j)^T \right)$$

$$\frac{\partial f}{\partial A} = 2(XA^T)^T \left(diag\left(sum(W, axis = 0)\right) - W - W^T \right) X$$

先来简单分析一下，在“#1”中，梯度计算是基于两层 for 循环进行的，所以可以得到最简单的实现方法，就是两层 for 循环得到梯度，在Python中利用 for 循环会很慢，所以我们还可以使用矩阵来加速操作，第一种矩阵加速是对“#1”的第二层 for 循环里面的部分使用矩阵来实现，但是会占用很多内存，第二种是利用“#2”所表达的梯度形式，实现起来很快，又不占太多内存。但是上面的实现都是利用普通的梯度下降实现的，当然还可以使用更有效的优化方法来实现，借助了scipy里面提供的一些优化方法来实现。下面详细介绍代码实现。

代码实现详见我的github：

lxcnju/Neighbourhood-Component-Analysis
github.com

2.1 实现方法一：两层 for 循环

首先，最简单地就是利用两层 for 循环遍历得到梯度：

```
# gradients
gradients = np.zeros((self.high_dims, self.high_dims))
# for i
for i in range(self.n_samples):
    k_sum = np.zeros((self.high_dims, self.high_dims))        # first_part
    k_same_sum = np.zeros((self.high_dims, self.high_dims))   # second_part
    # for k
    for k in range(self.n_samples):
        out_prod = np.outer(X[i] - X[k], X[i] - X[k])
        k_sum += prob_mat[i][k] * out_prod
        if Y[k] == Y[i]:
            k_same_sum += prob_mat[i][k] * out_prod
    gradients += prob_row[i] * k_sum - k_same_sum
gradients = 2 * np.dot(gradients, self.A)
```

在Python中，两层 for 循环的运算速度很慢，复杂度为 $O(n^2)$ ，n为样本数目。上面的代码实现是最naive的方法，所以在github里面命名为nca_naive.py。

2.2 实现方法二：矩阵操作加速

因为 for 循环速度太慢，所以优化一部分能用矩阵操作来进行得到的计算。下面我们先看看如何尽可能地使用矩阵操作，考虑下面问题，给定样本矩阵 $\boldsymbol{X}$ ，如何求第 i 个样本与所有样本的差呢，即 $\left[\boldsymbol{x_i}-\boldsymbol{x_1};\boldsymbol{x_i}-\boldsymbol{x_2};\cdots;\boldsymbol{x_i}-\boldsymbol{x_n}\right]^T$ ，当然利用 for 循环来写是可以的，但是在Python中利用广播（broadcast）操作则很容易 计算：

```
xik = X[i] - X
```

上面的很简单，那么假设要求样本距离矩阵呢？即 $\boldsymbol{D_{ij}}=||\boldsymbol{x_i}-\boldsymbol{x_j}||_2^2$ 。怎么用矩阵来实现？

当然我们可以利用两层 for 循环来实现，当然也可以用一层 for 循环，然后利用上面的广播来实现：

```
for i in range(n):
    dist_mat[i,:] = np.sum((X[i] - X) ** 2, axis = 1)
```

如何把两层 for 循环都去掉呢，都用矩阵操作？当然如果能想到下面的转换：

$$\boldsymbol{D_{ij}}=||\boldsymbol{x_i}||_2^2+||\boldsymbol{x_j}||_2^2-2\boldsymbol{x_i^T}\boldsymbol{x_j}$$

那么就有：

```
# distance matrix
sum_row = np.sum(low_X ** 2, axis = 1)
xxt = np.dot(low_X, low_X.transpose())
dist_mat = sum_row + np.reshape(sum_row, (-1, 1)) - 2 * xxt
```

如果不转换的话可以直接用矩阵实现吗？答案是可以的，下面是解决方法：

```
# X[None,:,:] shape = (1, n, d)
# X[:, None, :] shape = (n, 1, d)
# X[None,:,:] - X[:,None,:] shape = (n, n, d)
# sum of square from axis = 2
dist_mat = np.sum((X[None,:,:] - X[:,None,:]) ** 2, axis = 2)
```

同样地，在求梯度的过程中也可以尽可能地将 for 循环变为矩阵操作，可以提高一点速度，这里并没有减少时间复杂度，只是在实现层面上加速了一点而已。最后实现的梯度部分代码为：

```
# gradients
part_gradients = np.zeros((self.high_dims, self.high_dims))
for i in range(self.n_samples):
    xik = X[i] - X
    prod_xik = xik[:, :, None] * xik[:, None, :]
    pij_prod_xik = pij_mat[i][:, None, None] * prod_xik
    first_part = pi_arr[i] * np.sum(pij_prod_xik, axis = 0)
```

```
        second_part = np.sum(pij_prod_xik[Y == Y[i], :, :], axis = 0)
        part_gradients += first_part - second_part
    gradients = 2 * np.dot(part_gradients, self.A)
```

需要注意的是这种实现的办法会增加内存占用，比如上面介绍求距离矩阵时，会在内存产生一个(n, n, d)的矩阵，假若n 和 d都很大，那么内存开销就会产生Memory Error。因此只适用于小数据集。

2.3 实现方法三：利用梯度“#2”计算

上面介绍的利用梯度“#1”实现的方法一个太慢，一个太占内存。利用“#2”则会改善很多，下面直接附上代码：

```
# gradients
weighted_pij = pij_mat_mask - pij_mat * pi_arr[:, None]      # (n_samples, n_samples)
weighted_pij_sum = weighted_pij + weighted_pij.transpose()   # (n_samples, n_samples)
np.fill_diagonal(weighted_pij_sum, -weighted_pij.sum(axis = 0))
gradients = 2 * (low_X.transpose().dot(weighted_pij_sum)).dot(X).transpose()
```

可以看出，完全是利用矩阵操作，没有任何循环 操作，所以时间和空间都大大改善了。

2.4 实现方法四：利用梯度“#2” + Scipy优化包

上面的三种方法都要自己实现梯度下降，当然理想的是用Scipy里面的优化包来实现，我们只需要把cost 和 gradient 传给优化包就可以了，优化的过程我们就不用去干预了。借助scipy.optimize提供的minimize，fmin_cg来实现，前者是梯度下降，后者是坐标下降。最终的代码为：

```
# fit by gradient descent
# optimizer params
optimizer_params = {'method' : 'L-BFGS-B',
                    'fun' : self.nca_cost,
                    'args' : (X, Y),
                    'jac' : True,
                    'x0' : A.ravel(),
                    'options' : dict(maxiter = self.max_steps),
                    'tol' : self.tol}
opt = minimize(**optimizer_params)
```

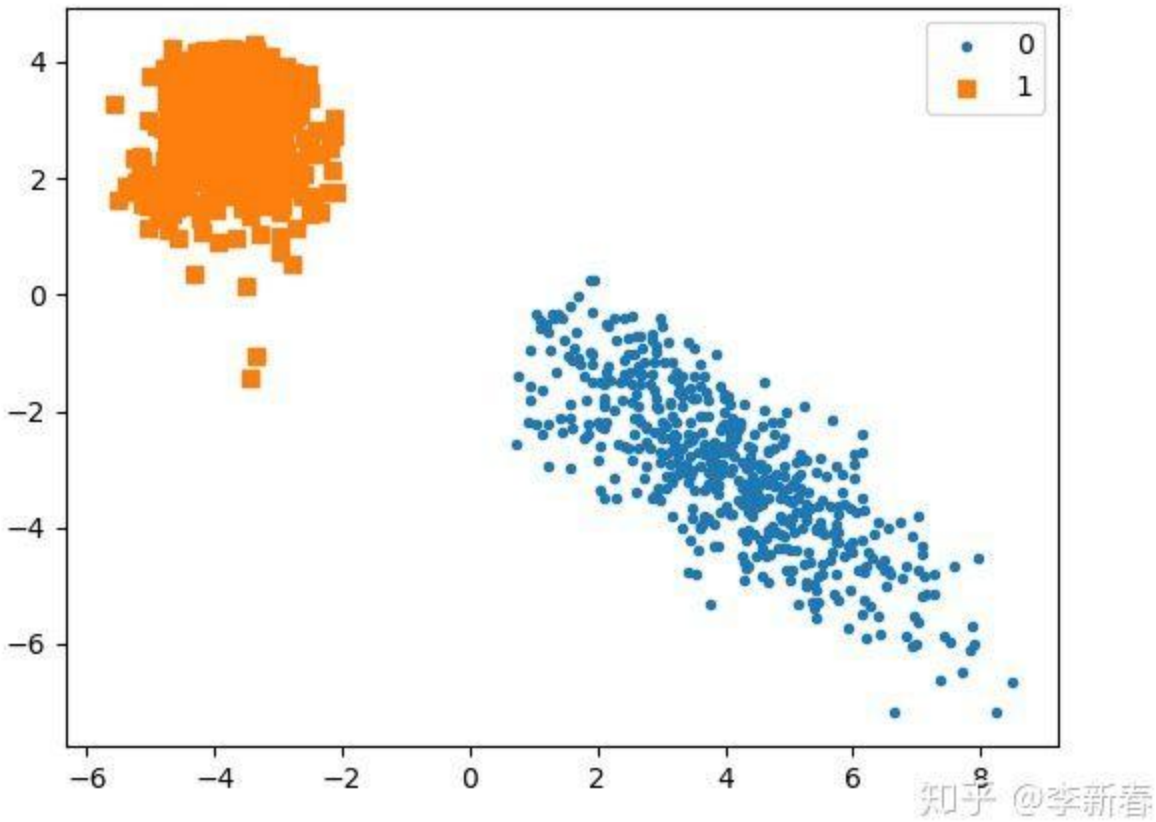
下面给出坐标下降的实现：

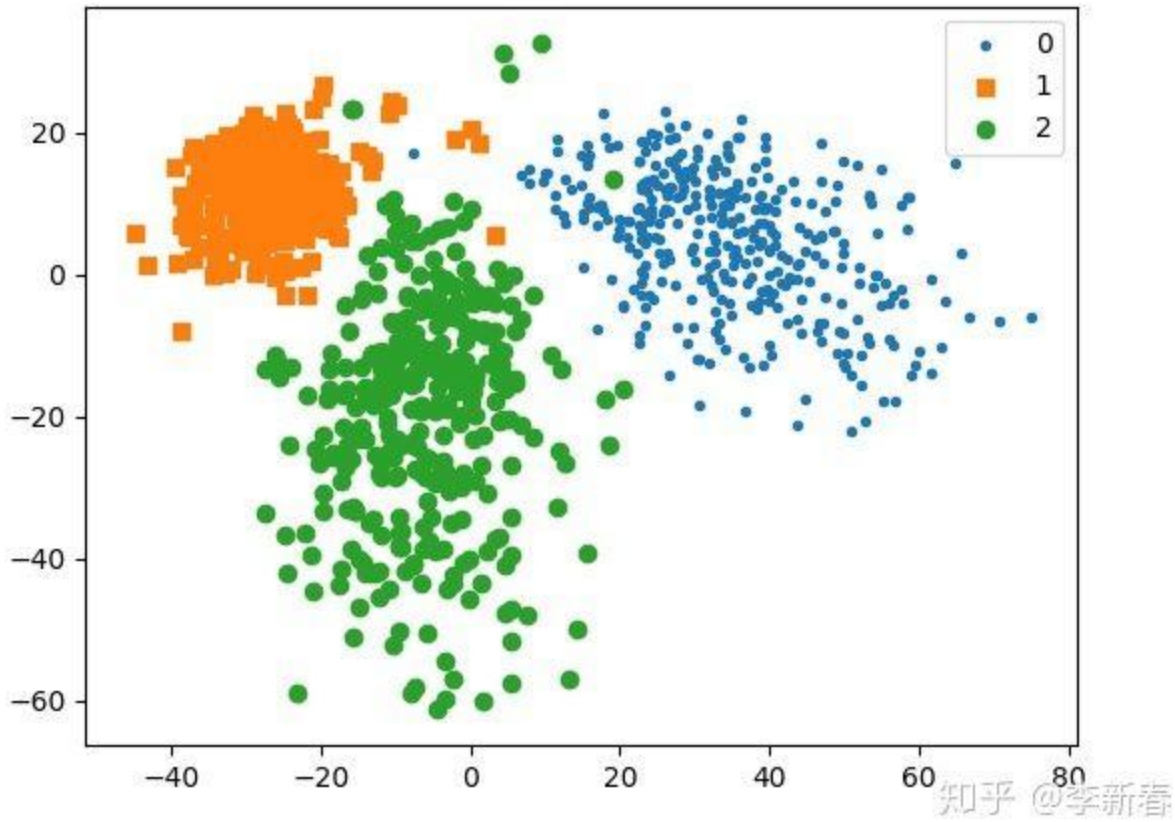
```
# fit by coordinate descent
def costf(A):
    f, _ = self.nca_cost(A.reshape((self.high_dims, self.low_dims)), X, Y)
    return f
def costg(A):
    _, g = self.nca_cost(A.reshape((self.high_dims, self.low_dims)), X, Y)
    return g
self.A = fmin_cg(costf, A.ravel(), costg, maxiter = self.max_steps)
```

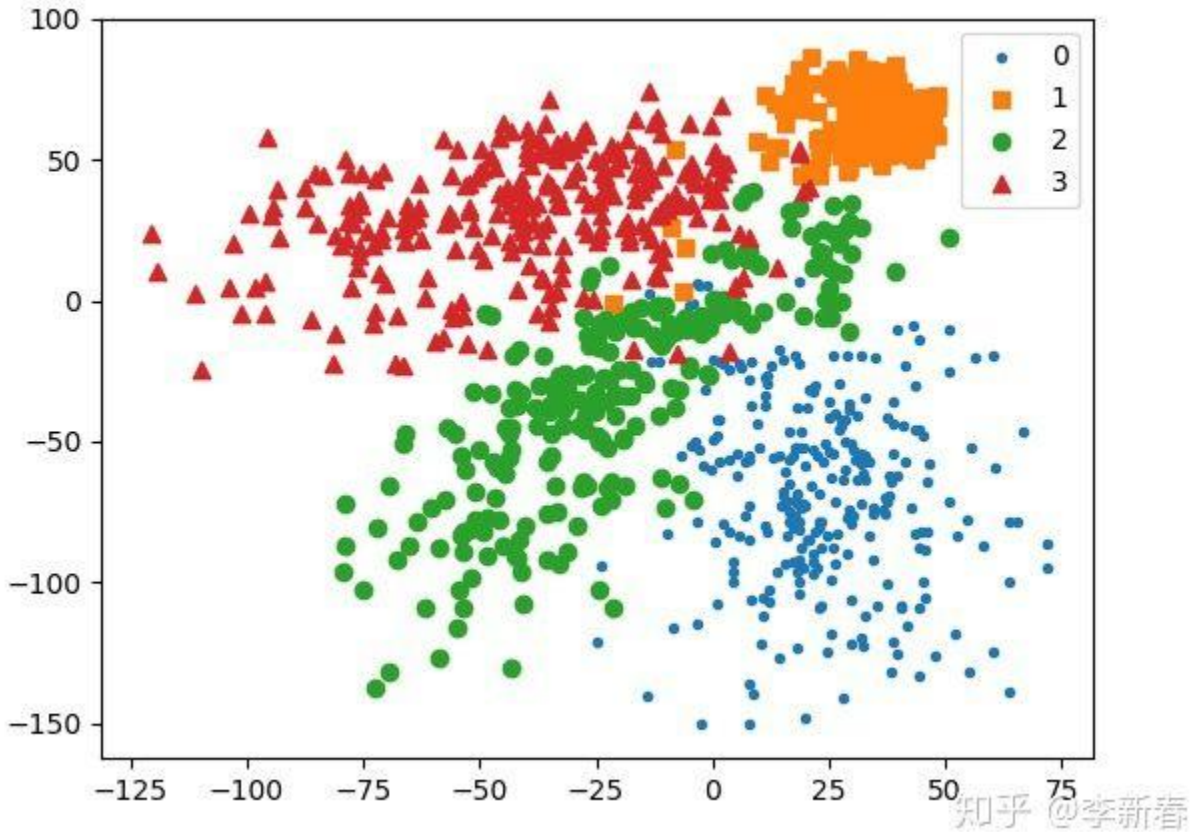
上面给出了几种实现，下面给出的实验结果都是基于nca_scipy.py来实现的，采用的是梯度下降。

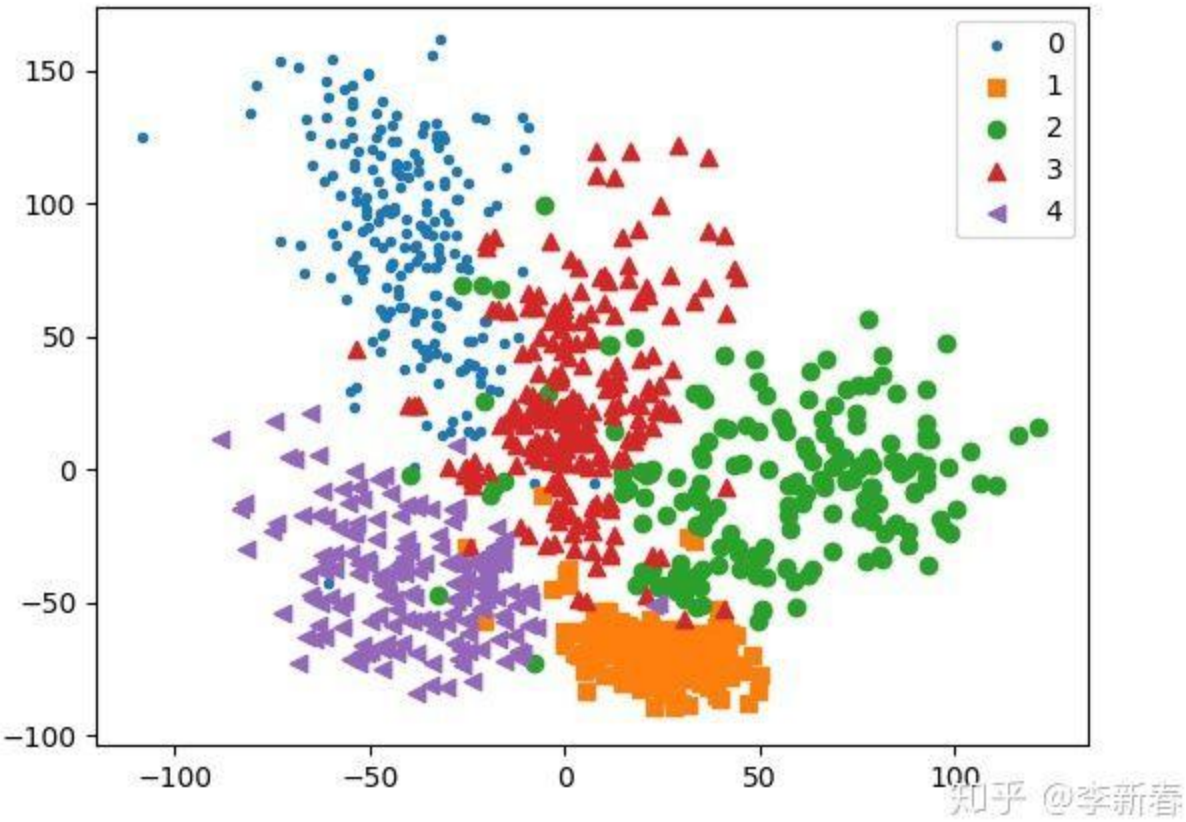
3 NCA在一些数据集上的表现

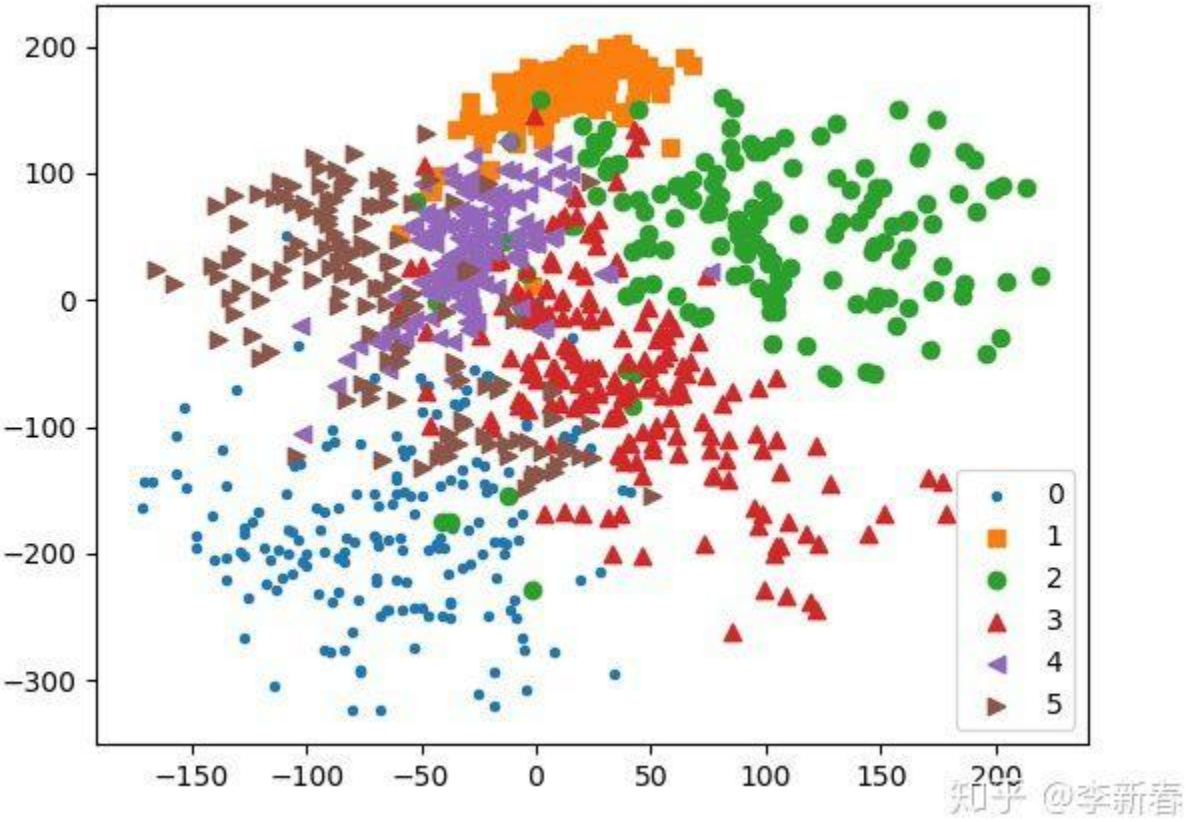
下面给出一些结果图片,前面9张是在mnist上面的结果，选取的数字类别数目分别为2至9，由于原图是784维的，所以先使用PCA降维到100维，然后再使用NCA降维到2维；后面三张是直接使用NCA在digits，breast_cancer和iris数据集降维到2维上得到的结果。

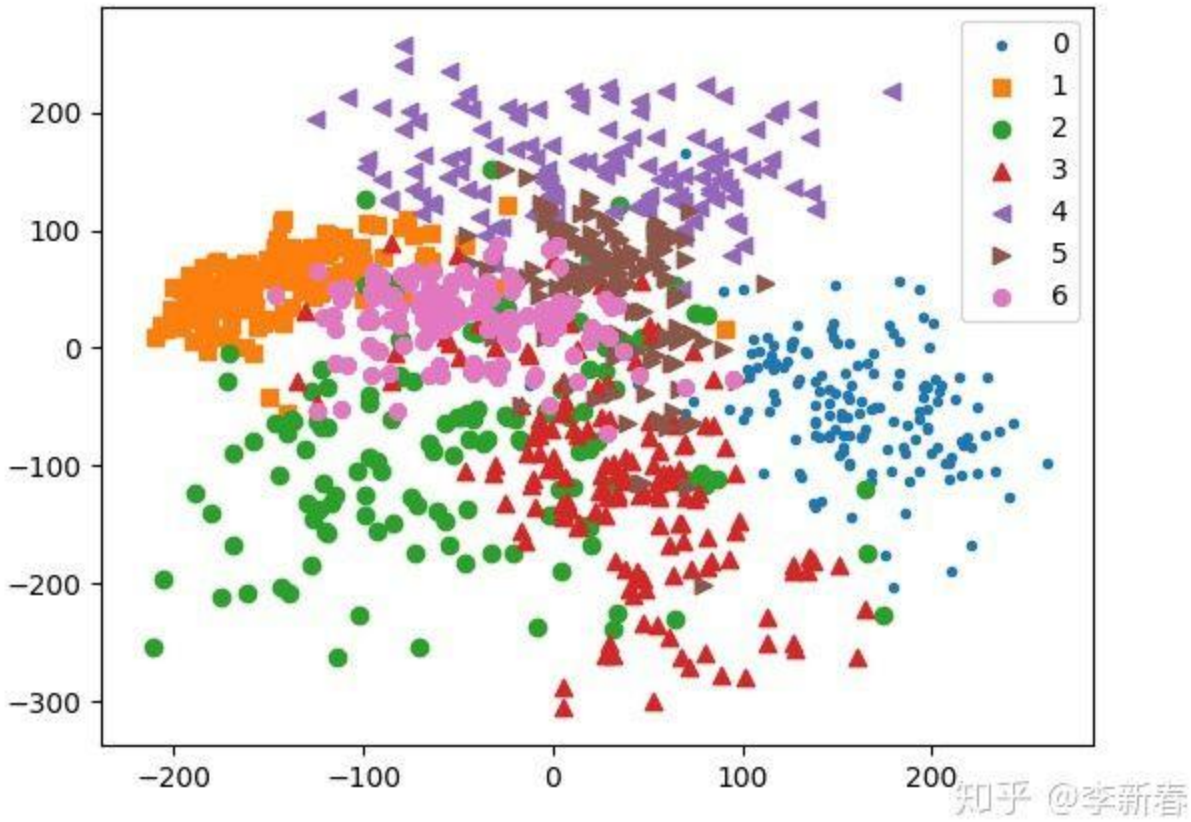


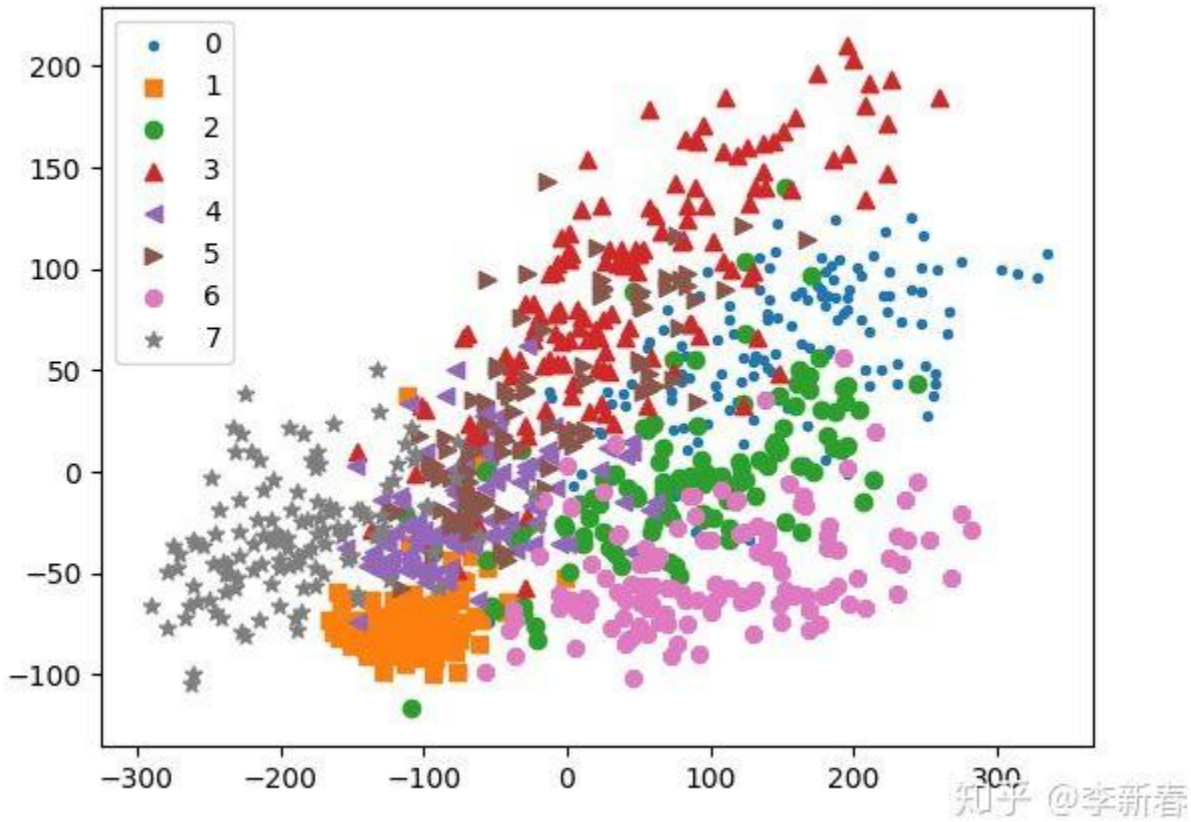

0
1
2
20
0
−20
−40
−60
−40
−20
0
20
40
60
80
知乎 @李新春

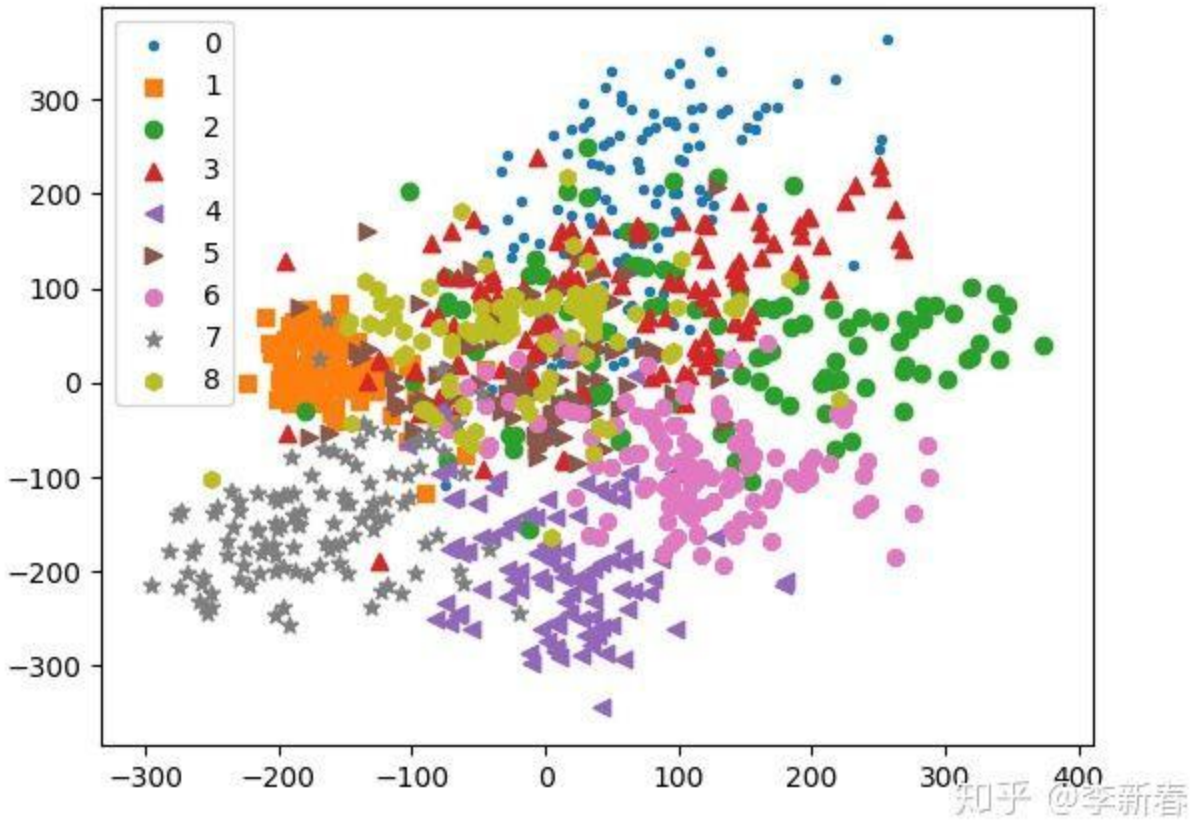

0
1
2
3
100
50
0
−50
−100
−150
−125
−100
−75
−50
−25
0
25
50
75
知乎 @李新春

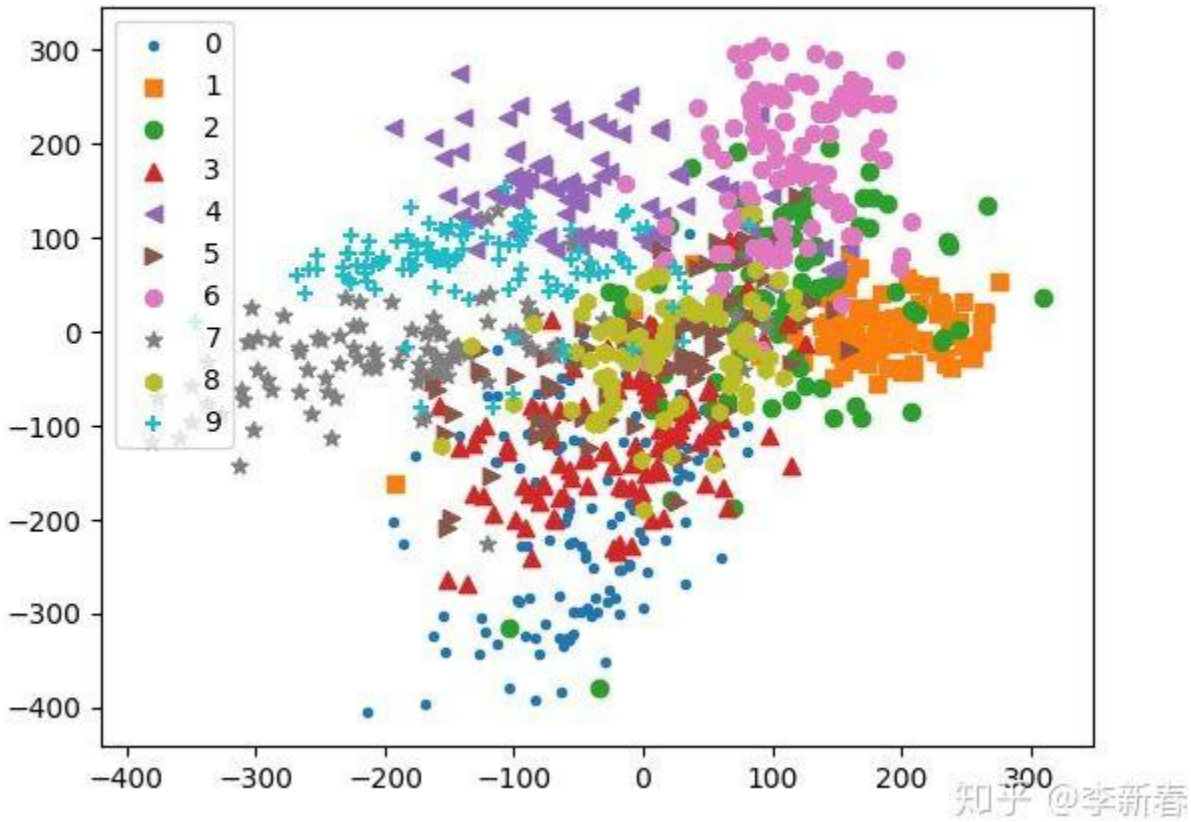

0
1
2
3
4
150
100
50
0
−50
−100
−100
−50
0
50
100
知乎 @李新春

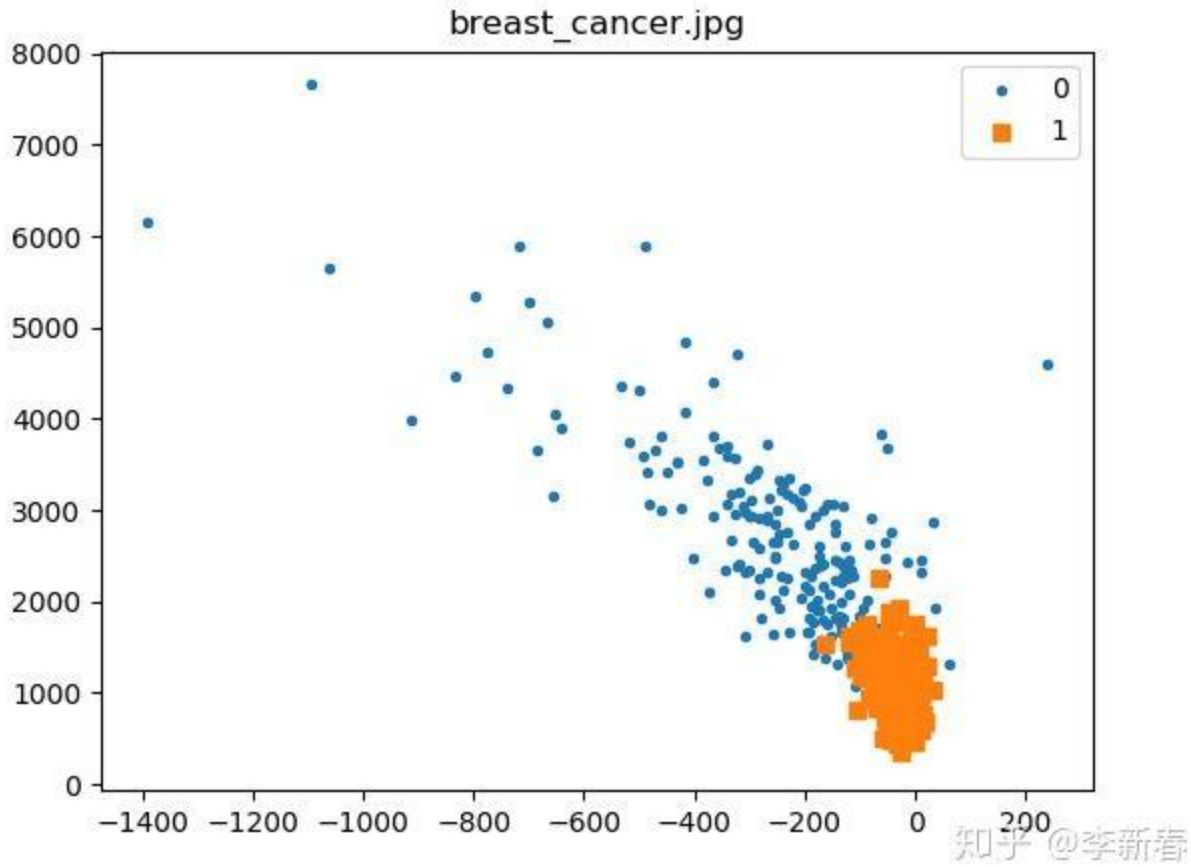

0
1
2
3
4
5
200
100
0
−100
−200
−300
−150
−100
−50
0
50
100
150
200
知乎 @李新春

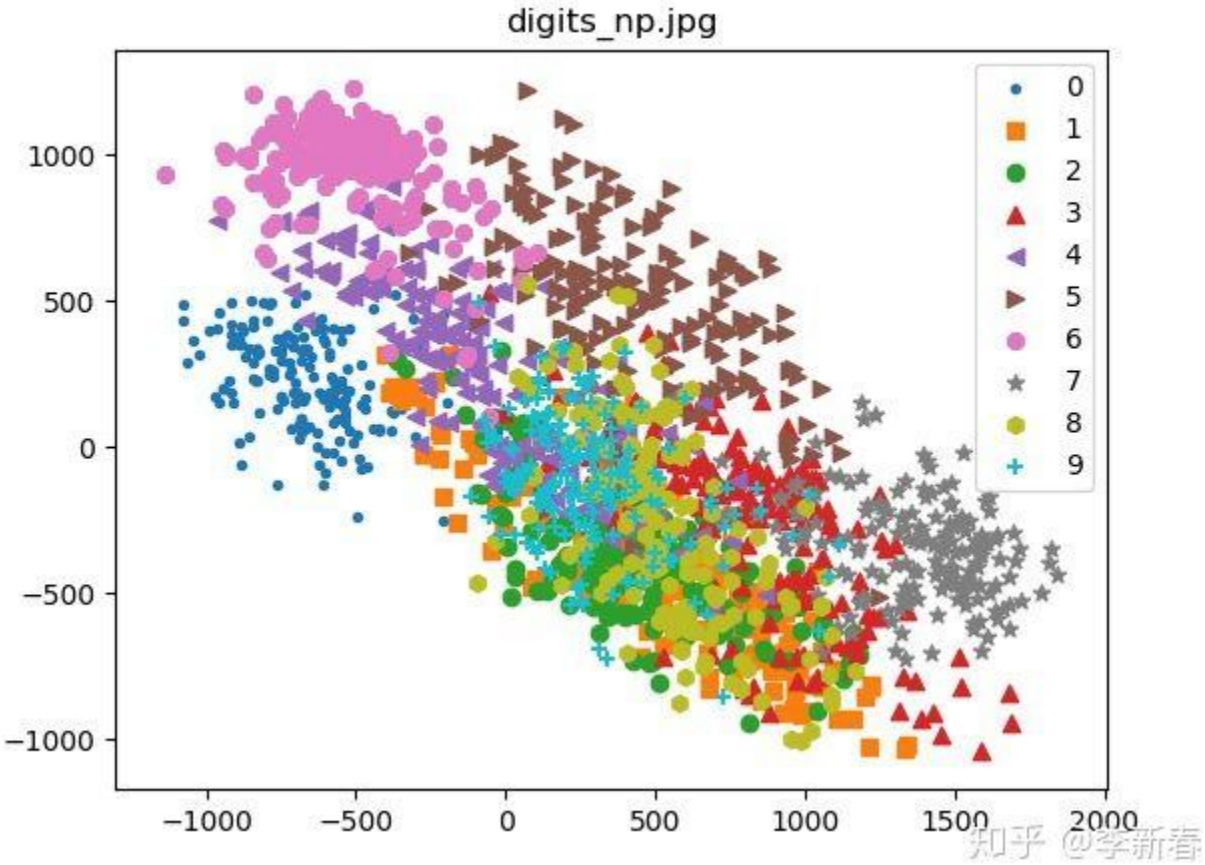

0
1
2
3
4
5
6
200
100
0
−100
−200
−300
−200
−100
0
100
200
知乎 @李新春

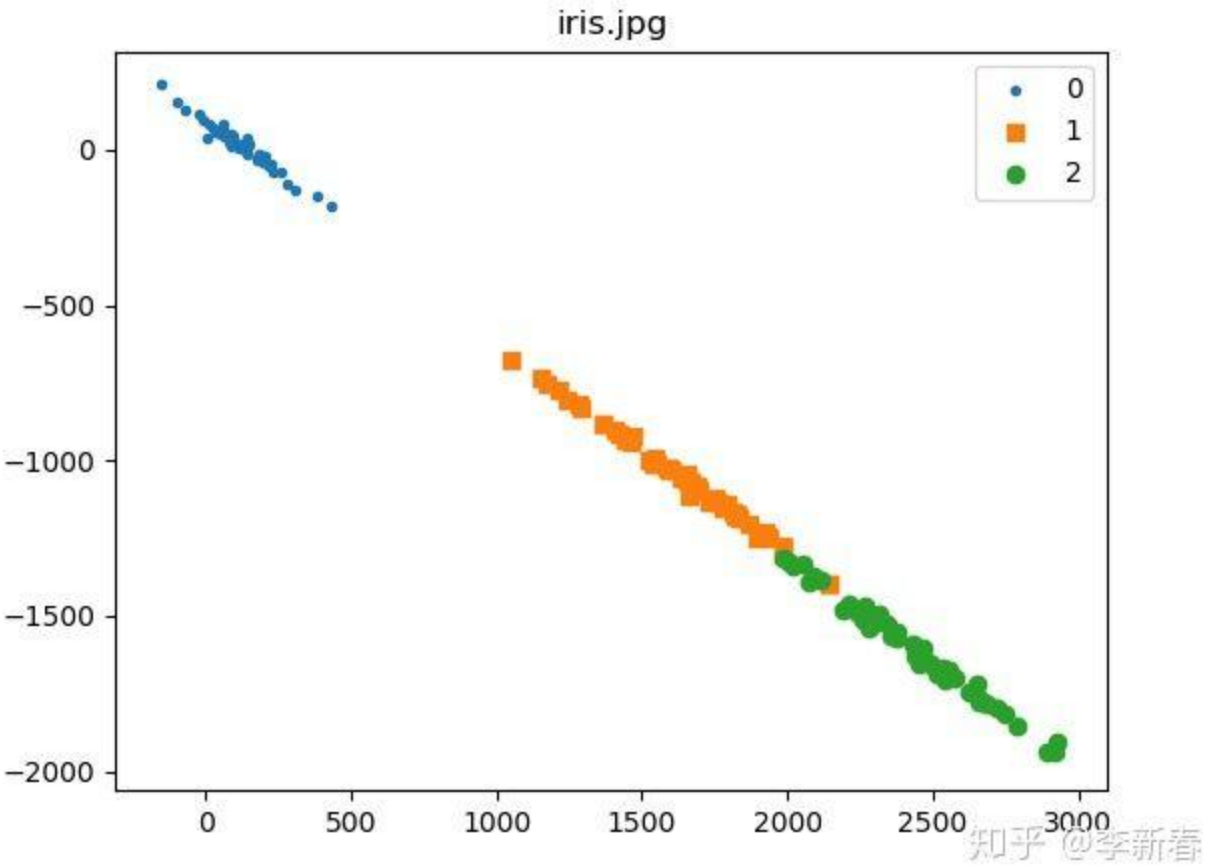

0
1
2
3
4
5
6
7
200
150
100
50
0
−50
−100
−300
−200
−100
0
100
200
300
知乎 @李新春


0
1
2
3
4
5
6
7
8
300
200
100
0
−100
−200
−300
−300
−200
−100
0
100
200
300
400
知乎 @李新春


0
1
2
3
4
5
6
7
8
9
300
200
100
0
−100
−200
−300
−400
−400
−300
−200
−100
0
100
200
300
知乎 @李新春


breast_cancer.jpg
8000
7000
6000
5000
4000
3000
2000
1000
0
−1400
−1200
−1000
−800
−600
−400
−200
0
200
0
1
知乎 @李新春


digits_np.jpg
1000
500
0
−500
−1000
−1000
−500
0
500
1000
1500
2000
0
1
2
3
4
5
6
7
8
9
知乎 @李新春


iris.jpg
0
1
2
0
−500
−1000
−1500
−2000
0
500
1000
1500
2000
2500
3000
知乎 @李新春