Algorithm of K-means

Given samples X = {x1,x2,...,xn}, where for each xi represents a sample, the purpose of k means is arranging those samples into k different classes.
It use the squared Euclidean distance as the distance between samples:

$$d(x_i, x_j) = \sum_{k=1}^{m} (x_{ki} - x_{kj})^2$$
$$= \|x_i - x_j\|^2$$

The loss function is defined as the sum of the distance of different samples to its centroids.

$$W(c) = \sum_{l=1}^{k} \sum_{C(i)=l} \|(xi - xl')\|^2$$

The purpose of K means is optimise this loss function: $C* = argminW(C)$

In realistic, we use iteration to solve this optimisation, because this may be a NP hard problem.
The iteration steps are:

input: X
output: cluster of X
steps: (1) initialise: random select k samples as initial centroids
(2) calculate all distances between X and initial centroids, arranging each samples into its nearest class
(3) calculate new centroids, which is the mean value of each feature
(4) if the iterations converge or satisfy the condition. Stop and output cluster

The time will spend O(mnk).

Algorithm of PCA

Given samples X = {x1,x2,...,xn}, where for each xi represents a sample:

$$X = [\ x_1 \quad x_2 \quad \cdots \quad x_n\ ] = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix} \qquad (1$$

Then we can get covariance S:

行：相关性
主：方差

$$S = [s_{ij}]_{m \times m}$$

实际X = 对X转置

(注1) $s_{ij} = \frac{1}{n-1} \sum_{k=1}^{n} (x_{ik} - \bar{x}_i)(x_{jk} - \bar{x}_j), \quad i,j = 1,2,\cdots,m \qquad (1$

And its Correlation matrix R:

样本相关矩阵 R 为 (归一化)

归一化
$\frac{x - \bar{x}_i}{std}$;
$std = \sqrt{var(x)}$

$$R = [r_{ij}]_{m \times m}, \quad r_{ij} = \frac{s_{ij}}{\sqrt{s_{ii} s_{jj}}}, \quad i,j = 1,2,\cdots,m$$

Our purpose is finding the eigenvalues and eigenvectors of R, because these eigenvectors are **orthogonal**, so these values and vectors can be seen as the most important part of getting matrix's effective informations. Which means they can reduce redundancy information or noise.

Arranging the eigenvalues from the largest to smallest.

$$\lambda 1 \geq \lambda 2 \geq ...$$

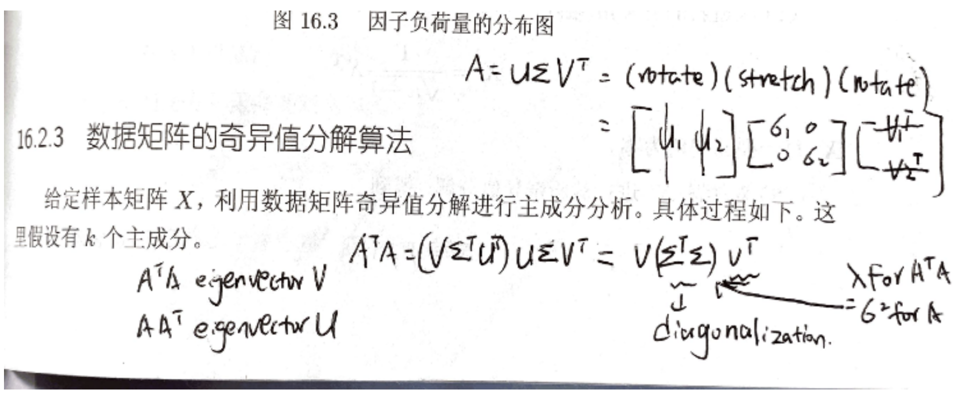
Then the correlated eigenvectors becomes the main component of X:

$$V = (v1, v2, ..., vk)$$

Thus we can get Y(k, n) = V.T(k, m) * X(m, n)
The Y is the target matrix after dimension reduced.

However, in real world, we cannot guarantee these eigenvectors are orthogonal, so we ofter use **SVD** .

This is my study note (please ignore the Chinese language part):



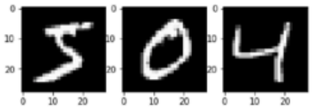
We can know that, for our X, we can use X.T *X to get the eigenvectors.

Suppose X' = 1/sqrt(n-1) * X.T
Then X'T.X' = 1/(n-1) * X * X.T = R(of X)
So the problem change into finding the SVD of X', where X' = U*sigma*V.T
Then the first k raws of V.T compose the k numbers of principal components.
Thus we can get Y(k, n) = V.T(k, m) * X(m, n)

Experiments:

Preprocessing the data:
One thing I want to discuss here is the image performance after using standard scaler:

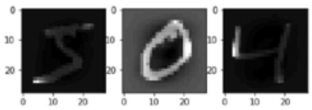
```
ax0.imshow(train_features[0].reshape(28,28),cmap='gray')
ax1.imshow(train_features[1].reshape(28,28),cmap='gray')
ax2.imshow(train_features[2].reshape(28,28),cmap='gray')
plt.show()
```



standard scaler

```
In [11]: from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
train_features=scaler.fit_transform(train_features)
test_features=scaler.fit_transform(test_features)
```

```
In [12]: fig,axs = plt.subplots(1,3)
ax0,ax1,ax2 = axs.ravel()
ax0.imshow(train_features[0].reshape(28,28),cmap='gray')
ax1.imshow(train_features[1].reshape(28,28),cmap='gray')
ax2.imshow(train_features[2].reshape(28,28),cmap='gray')
plt.show()
```

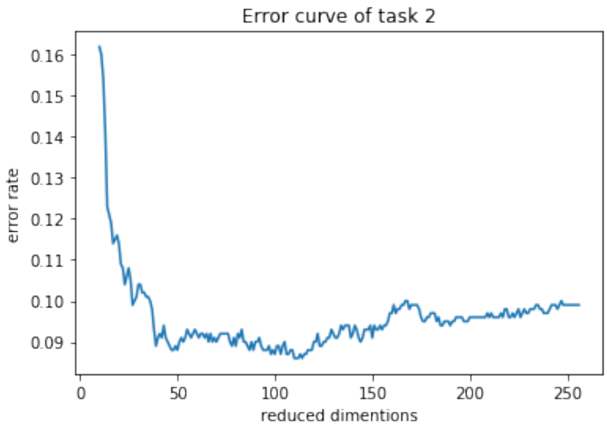


From the picture we can see that the structure of each number still remains, it seems that the only change is the shade of colour, which mainly because standard scaler does not change the distribution of data, it only narrow the data into -1 and +1.

Task 1:
In task 1 I build 2 models, one is the traditional PCA and the other using SVD. Because I want to see whether they have the similar trend and their performance.

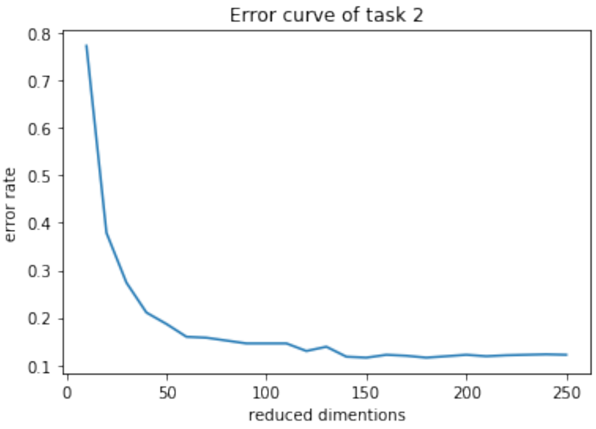
Task2:

This picture is the PCA using SVD, we can see some informations from it.
(1) The error curve sharp decreased from 10 to 50, and it becomes steady flat after 50, which can be explained that the front 50 eigenvectors are very important that can not be ignored. The eigenvalues that after 50 are much more smaller and we cannot get too much informations from them.
(2) The curve slightly arise during 100 to 250 and then becomes stable (because I checked dimensions in [512, 784], and their error is [0.102, 0.103])




```
In [16]: error2 = []
for i in [512,784]:
    temp = PCA(train_features.T,i)
    train_x = temp.fit(train_features.T)
    test_x = temp.fit(test_features.T)
    error = KNN(train_x,train_label,test_x).error(test_label)
    error2.append((error,i))
print(error2)
[(0.102, 512), (0.103, 784)]
```

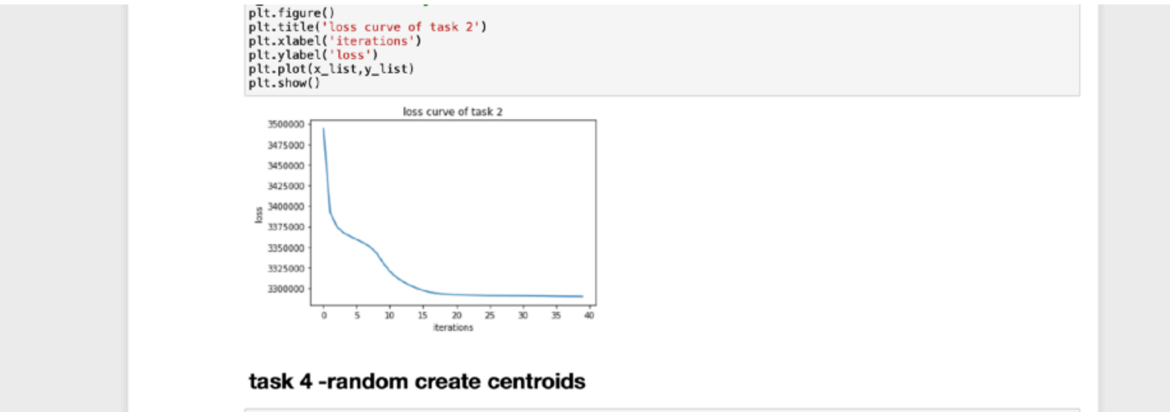
Compared to SVD, the traditional PCA performs similar:



This picture's dimensions interval is 10, so it seems more smooth, but both PCA and SVD have similar trend and all of their errors get narrowed around 0.1.

Task 3:

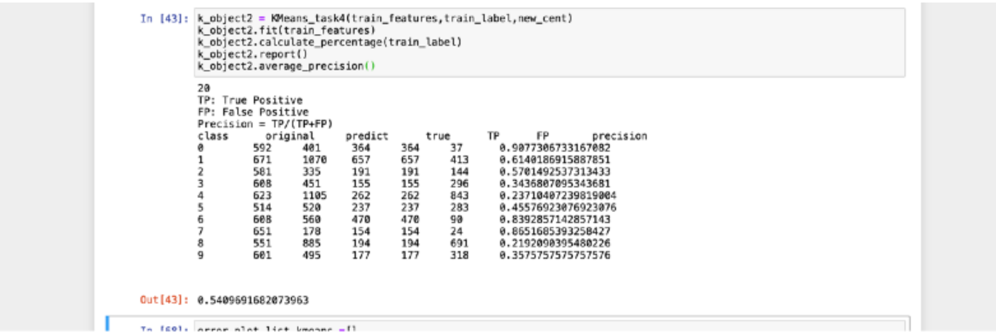
My iteration sets to 100, and the loss curve looks like:



```
plt.figure()
plt.title('loss curve of task 2')
plt.xlabel('iterations')
plt.ylabel('loss')
plt.plot(x_list,y_list)
plt.show()
```


As we can see, the loss value decreases dramatically before the first 20 iterations, and then becomes stable. This implies that the curve has ability to converge.
But if updating the centroids can not reduce the loss anymore, then theoretically, all the samples are almost classified correctly, then the accuracy should high.

Task4:
Without using PCA, K means performs like:



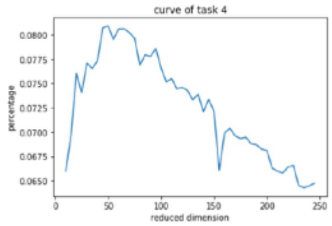
```
In [43]: k_object2 = KMeans_task4(train_features,train_label,new_cent)
         k_object2.fit(train_features)
         k_object2.calculate_percentage(train_label)
         k_object2.report()
         k_object2.average_precision()

20
TP: True Positive
FP: False Positive
Precision = TP/(TP+FP)
class     original     predict     true     TP     FP     precision
0         592     401     364     364     37      0.9077306733167082
1         671     1070    657     657     413     0.6140186915887851
2         581     335     191     191     144     0.5701492537313433
3         608     451     155     155     296     0.3436807095343681
4         623     1105    262     262     843     0.2371040723981904
5         514     520     237     237     283     0.4557692307692307
6         608     560     470     470     90      0.8392857142857143
7         651     178     154     154     24      0.8651685393258427
8         551     885     194     194     691     0.2192090395480226
9         601     495     177     177     318     0.3575757575757576

Out[43]: 0.5409691682073963
```

With PCA, K means performs like:
Why the curve arise, and then decrease?
I guess... when the dimensions are smaller than 50, the important informations are not sufficient,
So the curve increases when dimensions arise.

```python
plt.plot(x_list,y_list)
plt.show()
```

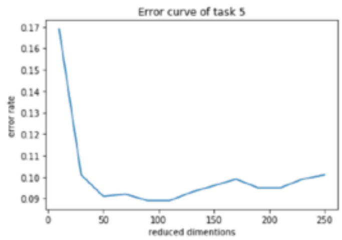


Task 5:

The original image is composed with 0 and 1, so I use generate 256 0 and 1 randomly and add them to the original data. Lets call it new data

Then I use the standard scaler to the new data for PCA,
Then, I get PCA picture:

```python
In [38]: y_list = [i[0] for i in error_plot_list]
x_list = [i[1] for i in error_plot_list]
plt.figure()
plt.title('Error curve of task 5')
plt.xlabel('reduced dimentions')
plt.ylabel('error rate')
plt.plot(x_list,y_list)
plt.show()
```



We can find that this curve is extremely similar to the task 2- curve. They all have a sharp decreasing before 50 and then becomes stable around 0.1.
The reason of that is PCA can decrease noise. The main idea of PCA is getting the unique(the most important) informations, replacing the redundant informations and abandoning rubbish informations. Thus PCA can do quite well in reducing rubbish features.
Besides, the SVM looks like:

```python
# predict test dataset

y_pred_with_scale = clf.predict(X_test)

print("Prediction report with scale:")
print(classification_report(y_test, y_pred_with_scale))
```

```
Prediction report with scale:
              precision    recall  f1-score   support

         0.0       0.97      1.00      0.99        99
         1.0       0.95      0.98      0.97       113
         2.0       0.98      0.94      0.96        94
         3.0       0.96      0.88      0.92       108
         4.0       0.96      0.95      0.95        93
         5.0       0.92      0.94      0.93        96
         6.0       0.96      0.97      0.97       101
         7.0       0.96      0.92      0.94       103
         8.0       0.95      0.92      0.93        99
         9.0       0.88      0.98      0.92        94

    accuracy                           0.95      1000
   macro avg       0.95      0.95      0.95      1000
weighted avg       0.95      0.95      0.95      1000
```

We can see that the accuracy is 0.95, which is quite high. It implies that SVM can also performs well when dealing with noise, which is also called **outliers**. As we have discussed in the former assignment, it is the points that near the SVM boundary important, the outliers can hardly influence the boundary, thus SVM can also performs well even with noise.