hw3

Wenhua Bao 2512664
Xinrui Chen 2503398

July 10, 2020

1 Task1

1.0.1 a1

Ridge coefficient is a regulizer. To regularize the psesudo-inverse,so the the increase of W will be not very significant. When the features is too much and data is not enough, linear regression will get a big value of W. When data changes a little ,the output will change significantly.

1.0.2 a2

squared error loss function

$$J(\theta) = \sum_{i=1}^{m} \left(f_\theta \left(x^{(i)} \right) - y^{(i)} \right)^2$$

add ridge coefficientλ

$$J(\theta) = \frac{1}{2} \left[\sum_{i=1}^{n} \left(f_\theta(x)^{(i)} - y^{(i)} \right)^2 + \sum_{j=1}^{n} \lambda \theta_j^2 \right]$$

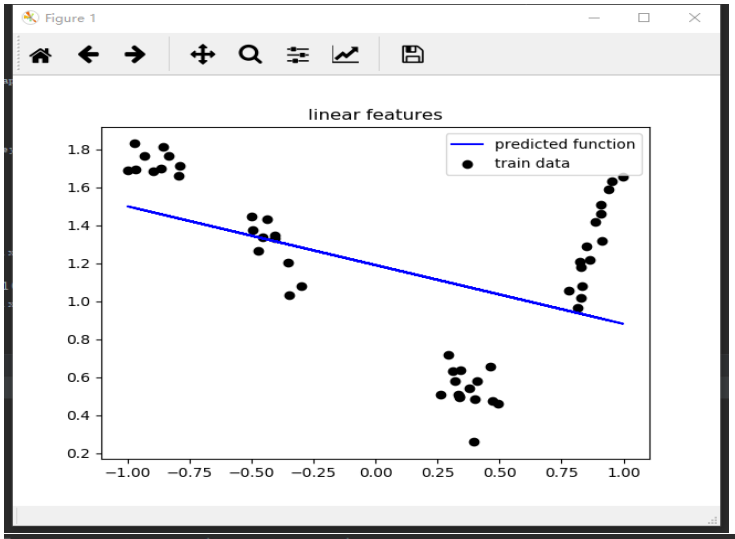
$$\begin{aligned} J(\theta) &= \frac{1}{2}(X\theta - Y)^\top (X\theta - Y) + \lambda \theta^\top \theta \\ &= \frac{1}{2}(X\theta - Y)^\top (X\theta - Y) + \lambda \theta^\top \theta \\ &= \frac{1}{2} \left(\theta^\top X^\top X\theta - \theta^\top X^\top Y - Y^\top X\theta + Y^\top Y + \lambda \theta^\top \theta \right. \end{aligned}$$

$$\begin{aligned} &\frac{\partial J(\theta)}{\partial \theta} = X^\top X\theta - X^\top Y + \lambda\theta = 0 \\ &\theta = \left(X^\top X + \lambda I \right)^{-1} X^\top Y \end{aligned}$$

1.0.3 a3

data test RMSEs:0.3843532873748282

1.0.4 a4




```python
def linear_features(data, lam=0.01):
    x_array = np.c_[np.ones(data[:, 0].shape[0]), data[:, 0]]
    y_array = data[:, 1]
    x = np.mat(x_array)
    y = np.mat(y_array).reshape(-1, 1)
    w = np.linalg.inv(x.T * x + lam * np.eye(x.shape[1])) * x.T * y
    return x, y, w


def quadratic_features(data, lam=0.01):
    x0 = data[:, 0]
    x_array = np.c_[np.ones(x0.shape[0]), x0, np.square(x0)]
    y_array = data[:, 1]
    x_p = np.linspace(np.min(x_array.ravel()), np.max(x_array.ravel()), 10000).reshape((-1, 1))
    x_plot = np.c_[np.ones(x_p.shape[0]), x_p, np.square(x_p)]
    x = np.mat(x_array)
    y = np.mat(y_array).reshape(-1, 1)
    w = np.linalg.inv(x.T * x + lam * np.eye(x.shape[1])) * x.T * y
    return x_p, x_plot, x, y, w


def cubic_features(data, lam=0.01):

    x0 = data[:, 0]
    x_array = np.c_[np.ones(x0.shape[0]), x0, np.square(x0), np.power(x0, 3)]
    y_array = data[:, 1]
    x_p = np.linspace(np.min(x_array.ravel()), np.max(x_array.ravel()), 10000).reshape((-1, 1))
    x_plot = np.c_[np.ones(x_p.shape[0]), x_p, np.square(x_p), np.power(x_p, 3)]
    x = np.mat(x_array)
    y = np.mat(y_array).reshape(-1, 1)
    w = np.linalg.inv(x.T * x + lam * np.eye(x.shape[1])) * x.T * y
    return x_p, x_plot, x, y, w


def quartic_features(data, lam=0.01):

    x0 = data[:, 0]
    x_array = np.c_[np.ones(x0.shape[0]), x0, np.square(x0), np.power(x0, 3), np.power(x0, 4)]
    y_array = data[:, 1]
    x_p = np.linspace(np.min(x_array.ravel()), np.max(x_array.ravel()), 10000).reshape((-1, 1))
    x_plot = np.c_[np.ones(x_p.shape[0]), x_p, np.square(x_p), np.power(x_p, 3), np.power(x_p, 4)]
    x = np.mat(x_array)
    y = np.mat(y_array).reshape(-1, 1)
    w = np.linalg.inv(x.T * x + lam * np.eye(x.shape[1])) * x.T * y
    return x_p, x_plot, x, y, w
```

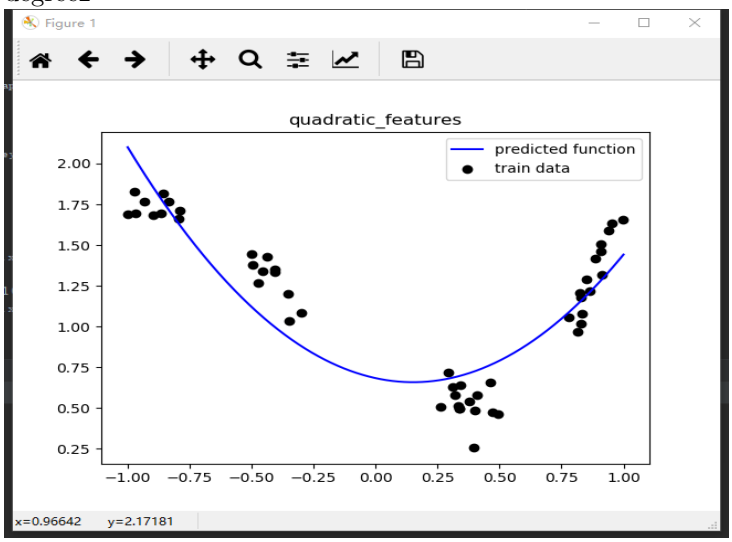
1.0.5 b1

```
polynomials of degrees 2:
EMSE of train data: 0.21201447265968615
EMSE of test data: 0.19770227360236917
polynomials of degrees 3:
EMSE of train data: 0.08706821295481748
EMSE of test data: 0.10253259510642491
polynomials of degrees 4:
EMSE of train data: 0.08701261306638178
EMSE of test data: 0.10034141482688069
```

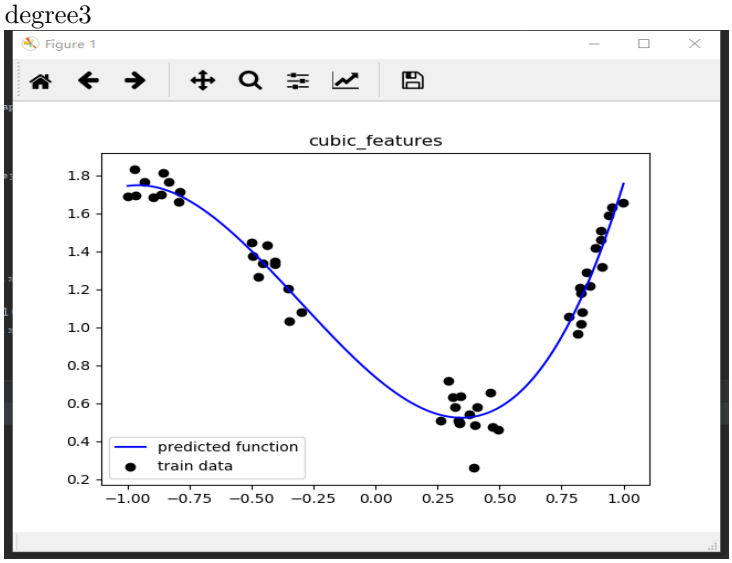
```
#1b
x_plot_raw_train, x_plot_train, x_train, y_train, w_train = quadratic_features(lin_reg_train)
EMSE_train = RMSE(y_train, x_train*w_train)
x_plot_raw_test,x_plot_test, x_test, y_test, w_test = quadratic_features(lin_reg_test)
EMSE_test = RMSE(y_test, x_test*w_test)
print('polynomials of degrees 2:')
print('EMSE of train data:', EMSE_train)
print('EMSE of test data:', EMSE_test)
plt.scatter(lin_reg_train[:, 0], lin_reg_train[:, 1], c='black', label='train data')
plt.plot(x_plot_raw_train, x_plot_train * w_train, c='blue', label='predicted function')
plt.title('quadratic_features')
plt.legend()
plt.show()

x_plot_raw_train, x_plot_train, x_train, y_train, w_train = cubic_features(lin_reg_train)
EMSE_train = RMSE(y_train, x_train*w_train)
x_plot_raw_test,x_plot_test, x_test, y_test, w_test = cubic_features(lin_reg_test)
EMSE_test = RMSE(y_test, x_test*w_test)
print('polynomials of degrees 3:')
print('EMSE of train data:', EMSE_train)
print('EMSE of test data:', EMSE_test)
plt.scatter(lin_reg_train[:, 0], lin_reg_train[:, 1], c='black', label='train data')
plt.plot(x_plot_raw_train, x_plot_train * w_train, c='blue', label='predicted function')
plt.title('cubic_features')
plt.legend()
plt.show()

x_plot_raw_train, x_plot_train, x_train, y_train, w_train = quartic_features(lin_reg_train)
EMSE_train = RMSE(y_train, x_train*w_train)
x_plot_raw_test,x_plot_test, x_test, y_test, w_test = quartic_features(lin_reg_test)
EMSE_test = RMSE(y_test, x_test*w_test)
print('polynomials of degrees 4:')
print('EMSE of train data:', EMSE_train)
print('EMSE of test data:', EMSE_test)
plt.scatter(lin_reg_train[:, 0], lin_reg_train[:, 1], c='black', label='train data')
plt.plot(x_plot_raw_train, x_plot_train * w_train, c='blue', label='predicted function')
plt.title('quartic_features')
plt.legend()
plt.show()
```

1.0.6 b2

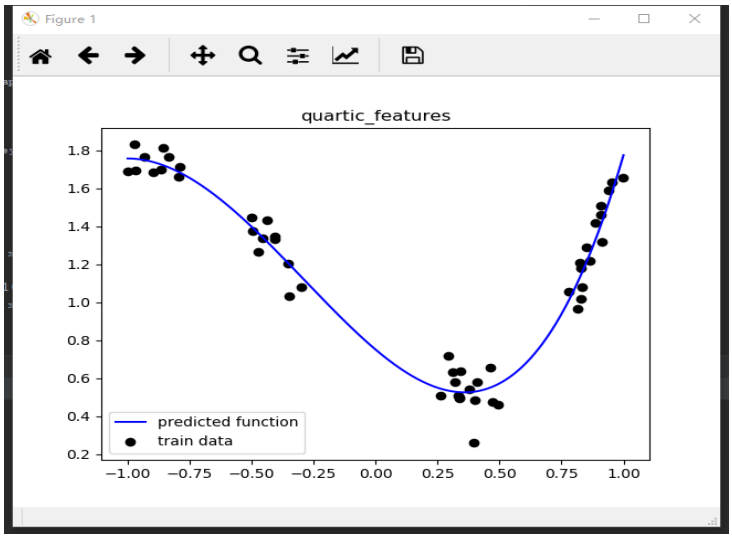
degree2



degree3



degree4



1.0.7 b3

The regression function in prediction from train data is linear.

1.0.8 c1

```
#train_1 = np.r_[x_data_train[0:10], x_data_train[20:50]]
train_1 = x_data_train[10:50]
train_2 = x_data_train[0:10]
#y_train_1 = np.r_[y_data_train[0:10], y_data_train[20:50]]
y_train_1 = y_data_train[10:50]
y_train_2 = y_data_train[0:10]
#creat model, save error
```

oder :train validation test
fold1:

2 0.22556753 0.15048197 0.22028206830068742
3 0.08428286 0.15048197 0.11179742139957746
4 0.08414773 0.10435305 0.11268232077846108
fold2:
2 0.1989927 0.26645353 0.2292217385755652
3 0.08601135 0.09440721 0.10875662754775296
4 0.08603288 0.09418586 0.1085676042605185
fold3:
2 0.20084105 0.25867888 0.22443805339442496
3 0.08881662 0.08738739 0.10848132887128964
4 0.08864758 0.08900698 0.10951896935966333
fold4:
2 0.22494425 0.16930346 0.21190796159995878
3 0.09214729 0.06459363 0.10969028524269515
4 0.09073245 0.07924134 0.10723300347728246

fold5:
2 0.19685787 0.27953923 0.20534393067167173
3 0.07978302 0.1138868 0.10725099232824602
4 0.07786331 0.12530459 0.10461065723562771

1.0.9 c2

The result don't fit my expectations, think because 1.the divided data is not average 2. the noise disturbed the fitting and leads to overfitting 3. the number of test data¿train data.

1.0.10 c3

I think we should degree 3, because the difference between 3 and 4 is little and maybe the noise makes a bais.

1.0.11 d1

$P(w|x, y) \sim$ Gaussian Distribution

1.0.12 d2

$p\left(y_* \mid x_*, x, y\right) \sim$ Gaussian Distribution

1.0.13 d3

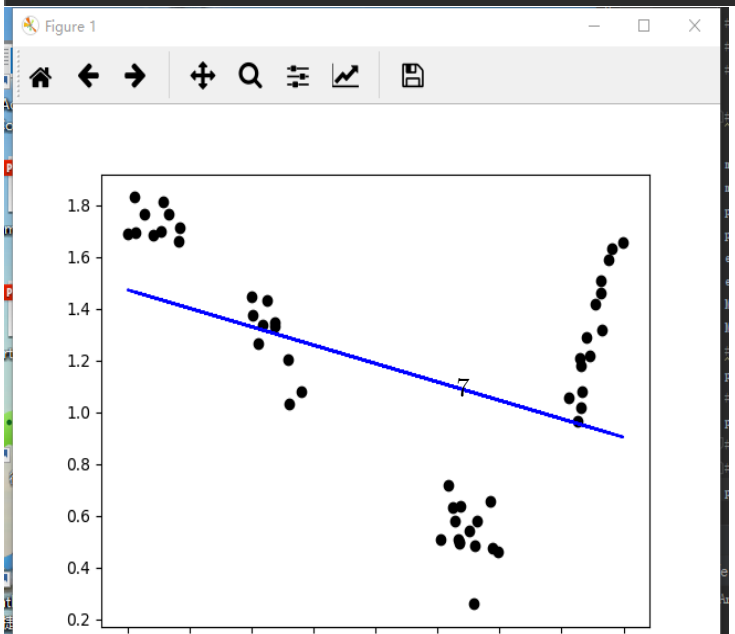
train :3.5386226788394106 test:4.588156582914009

1.0.14 d4

train: -31235.44383000205 test:-105117.53949070684

1.0.15 d5

```
#1d


model = BayesianRidge(alpha_1=0.01, lambda_1=0.01)
model.fit(x_pha,y_data_train.ravel())
pre_train = model.predict(x_pha)
pre_test = model.predict(x_pha_test)
error_train = np.sqrt(np.sum(np.power((y_data_train-pre_train),2))/50)
error_test = np.sqrt(np.sum((y_data_test-pre_test)**2)/100)
MLK_train = 50*np.log(1/np.sqrt(2*np.pi)/0.1)-np.sum(np.power((y_data_train-pre_train),2))/2/0.01
MLK_test = 100*np.log(1/np.sqrt(2*np.pi)/0.1)-np.sum(np.power((y_data_test-pre_test),2))/2/0.01
#print(MLK_test)
plt.scatter(x_data_train, y_data_train, marker='o',color='black', label='train_data')
# plt.scatter(x_data_test, y_data_test, marker='*',color='green', label='test_data')
plt.plot(x_data_train, pre_train, c='blue')
#plt.xlabel('x')
#plt.ylabel('y')
plt.show()
```



```
#1d


model = BayesianRidge(alpha_1=0.01, lambda_1=0.01)
model.fit(x_pha,y_data_train.ravel())
pre_train = model.predict(x_pha)
pre_test = model.predict(x_pha_test)
error_train = np.sqrt(np.sum(np.power((y_data_train-pre_train),2))/50)
error_test = np.sqrt(np.sum((y_data_test-pre_test)**2)/100)
MLE_train = 50*np.log(1/np.sqrt(2*np.pi)/0.1)-np.sum(np.power((y_data_train-pre_train),2))/2/0.01
MLE_test = 100*np.log(1/np.sqrt(2*np.pi)/0.1)-np.sum(np.power((y_data_test-pre_test),2))/2/0.01
#print(MLE_test)
plt.scatter(x_data_train, y_data_train, marker='o',color='black', label='train_data')
#plt.scatter(x_data_test, y_data_test, marker='*',color='green', label='test_data')
plt.plot(x_data_train, pre_train, c='blue')
#plt.xlabel('x')
#plt.ylabel('y')
plt.show()
```

1.0.16 d6

linear regression is the fit of data point , but the bayesian is the fit of data distribution propability.

1.0.17 e1

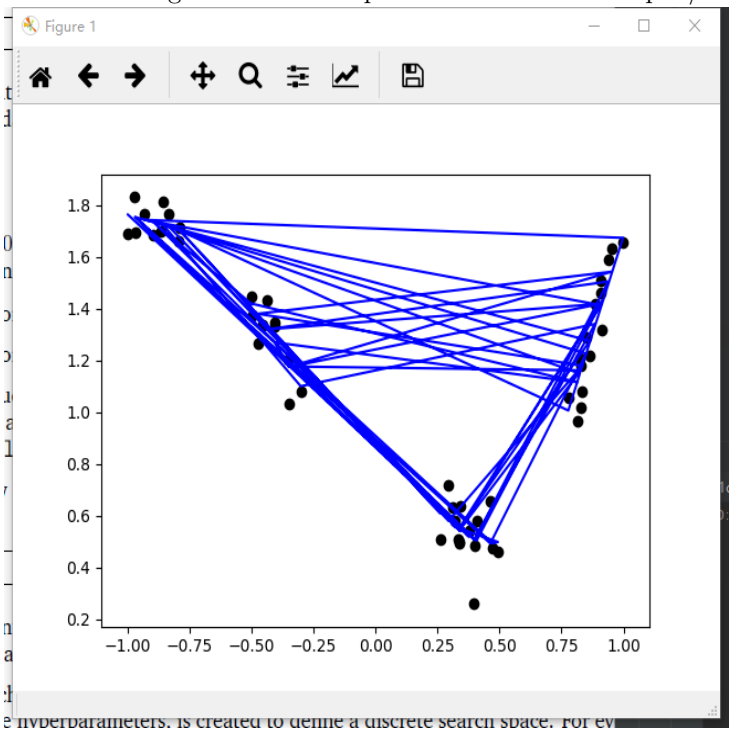
train:4.573687487696011 test:12.294027090210921

1.0.18 e2

train:-52227.36075977815 test: -755577.1458182211

1.0.19 e3

SE features regulize the X sample matrix lambata = alpha/beta



1.0.20 e4

2 Task2

2.1 2a

Generative models learn prior distribution to derive posterior distribution then get classification, but discriminative models learn posterior distribution to get

classification. discriminative models: Logistical Regression generative models: Bayesian Analysis Discriminative models is easier to learn, because it doesn't need to learn conditional probability and directly to learn posterior.

2.2 2b

there's 19 samples being misclassified.

```
from sklearn import discriminant_analysis

def plot_LDA(converted_X, y):
    '''
    draw the LDA transformed data
    :param converted_X: data set after Lda
    :param y: the label
    :return: None
    '''
    import matplotlib.pyplot as plt
    colors = 'rgb'
    markers = 'o*s'
    for target, color, marker in zip([1, 2, 3], colors, markers):
        pos = (y == target).ravel()
        X = converted_X[pos, :]
        plt.scatter(X[:, 0], X[:, 1], color=color, marker=marker,
                    label="Label %d" % target)
    plt.legend(loc="best")
    plt.suptitle("After LDA")
    plt.show()


import numpy as np

x_data = np.loadtxt("ldaData.txt")
y_data = np.zeros(137)
print(x_data.shape, y_data.shape)
for i in range(50):
    y_data[i]=1
for i in range(50,93):
    y_data[i]=2
for i in range(93,137):
    y_data[i]=3
lda = discriminant_analysis.LinearDiscriminantAnalysis()
lda.fit(x_data, y_data)
pre = lda.predict(x_data)
num = 0
for i in range(137):
    if y_data[i] != pre[i]:
        num = num+1
print(num)
converted_X = np.dot(x_data, np.transpose(lda.coef_)) + lda.intercept_
plot_LDA(converted_X, y_data)
plot_LDA(x_data, y_data)
```

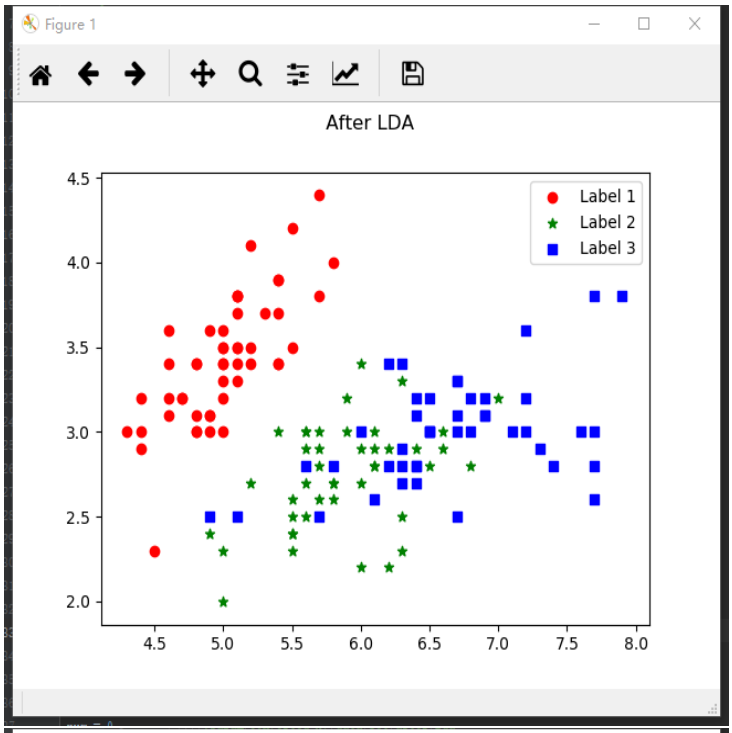


Figure 1
After LDA
Label 1
Label 2
Label 3
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0

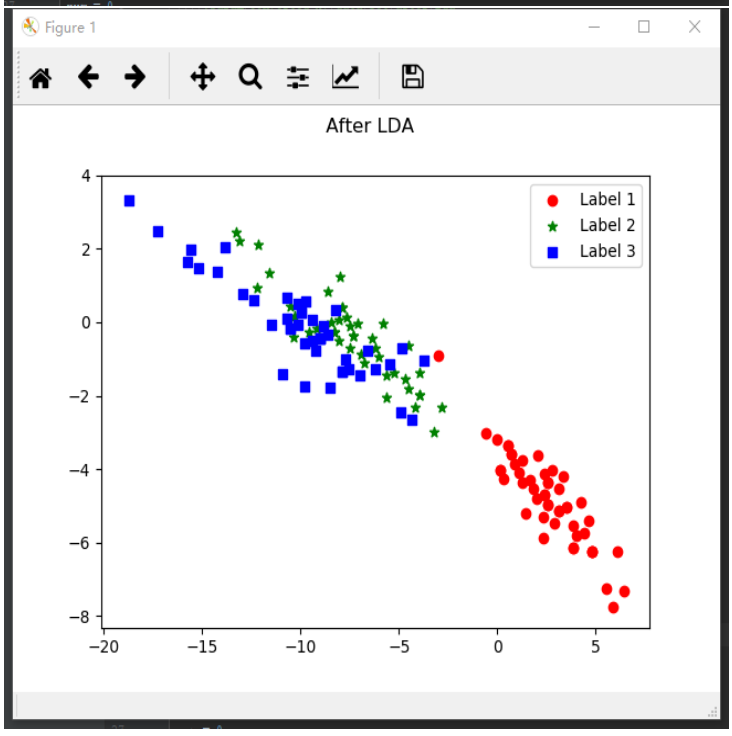


Figure 1
After LDA
Label 1
Label 2
Label 3
4
2
0
−2
−4
−6
−8
−20
−15
−10
−5
0
5

Task 3: Principal Component Analysis

3a)

Normalizing can convert the original data into pure,dimensionless values and remove the unit restriction on data, then it is easy to compare and weight indicator among data with different units or scales. After Normalizing the data is easy to handle. One Advantage is that the convergence speed of the model improves. For example, if there is only two features, and the range of component 1is 1000 and the range of component 2 is 5. The speed of iteration is very low during the optimization. But after normalizing the speed will be faster. Another advantage is that the accuracy of the model improves. In the last example, when we need to calculate about the distance, the second feature make much less contribute to the result than the first one.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
# get data
iris = np.loadtxt("./dataSets/iris.txt", delimiter=',')
iris_set = iris[:, 0:4]
iris_label = iris[:, 4]
# Normalizing
iris_norm = (iris_set-np.mean(iris_set, axis=0))/np.std(iris_set, axis=0)
print('mean:', np.mean(iris_norm, axis=0))
print('var:', np.var(iris_norm, axis=0))
```

3b)

We need only two components to in order to explain at least 95% of the dataset variance. From the picture below, we can know that the cumulative variance of first two components is 95.8010%.

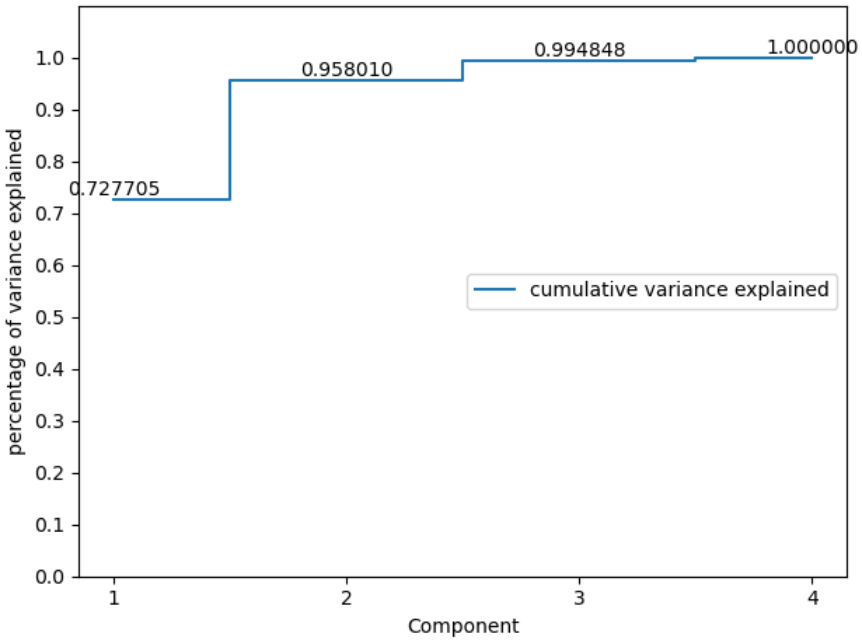


Figure 1: cumulative variance explained

Code:

```
#PCA
cov_matrix = np.cov(iris_norm, rowvar=0)
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
eigenvalues_sorted = np.sort(eigenvalues)[-1::-1]
total = np.sum(eigenvalues_sorted)
var_explained = np.zeros(4)
for i, item in zip(range(4), eigenvalues_sorted):
   var_explained[i] = item/total
cum_var_explained = np.cumsum(var_explained)
x = ['1','2','3','4']
plt.step(x, cum_var_explained, where='mid', label='cumulative variance explained')
for a, b in zip(x, cum_var_explained):
   plt.text(a, b, '%f' % b, ha='center', va='bottom')
plt.ylim((0, 1.1))
plt.yticks(np.arange(0,1.1,0.1))
plt.xlabel('Component')
plt.ylabel('percentage of variance explained')
plt.legend(loc='center right')
plt.show()
```

3c)

From the picture below, we can know that the data can be clearly distinguished by using 2 components, specially, through component1 the data is distinguished. Two components can explain 95% of the dataset variance. After PCA we spared two feature, i.e., 50% data, but the data can be still good distinguished and only 5% dataset variance is lost.

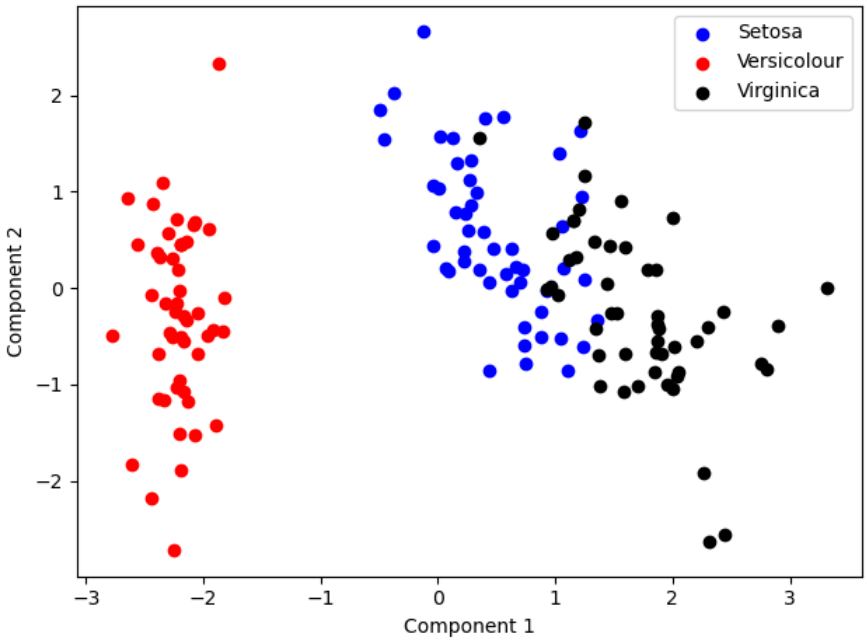


Figure 2: lower dimensional projection

Code:

```
vector_n = eigenvectors[:,0:2]
iris_lowDim = iris_norm@vector_n
name =['Setosa', 'Versicolour', 'Virginica']
print(iris_lowDim.shape)
for lab, col in zip(range(3), ('blue', 'red', 'black')):
    plt.scatter(iris_lowDim[iris_label==lab, 0], iris_lowDim[iris_label==lab, 1],
        label=name[lab], color=col)
plt.xlabel('Component 1')
plt.ylabel('Component 2')
plt.legend()
plt.show()
```

3d)

N. of components	x1	x2	x3	x4
1	0.22925036	0.18006108	0.29805579	0.31692192
2	0.11261513	0.19281695	0.13314032	0.12717403
3	0.07919279	0.23615959	0.13302896	0.1257637
4	0.04925365	0.23658978	0.1322316	0.10227491

Code:

```
# reconstruct
def nrmse(eigvec,n):
   vec_n = eigvec[:, 0:n]
   lowDim = iris_norm @ vec_n
   std = np.std(iris_set, axis=0)
```

```
    iris_reconstruct = lowDim@vec_n.T+np.mean(iris_set, axis=0)
    k = np.max(iris_set, axis=0)-np.min(iris_set, axis=0)
    return np.sqrt(np.mean(np.square(iris_reconstruct-iris_set), axis=0))/k
for n in range(4):
    print(nrmse(eigenvectors, n))
```

3e)

1. Explain the difference between PCA and ZCA whitening.

PCA whitening: It ensures the variance of each dimension of the data is 1.
ZCA whitening: It ensures that the variance of each dimension of the data is the same. It does a rotation operation on the basis of PCA whitening to make the data after whitening closer to the original data

2. State the equation(s) to compute the ZCA whitening parameters, given the data.

There are m samples, and the feature dimension of each sample is n. We have a sample set feature matrix X of n rows and m columns. And then we calculate Zero-average each row of X. The covariance matrix of X:

$$\Sigma = \frac{1}{m} X X^T \tag{1}$$

Calculate the eigenvalues and corresponding eigenvectors of the covariance matrix:

$$\frac{1}{m} X X^T = U A U^T \tag{2}$$

Rotate data

$$X_{rot} = U^T X \tag{3}$$

Then we get the PCA whitening

$$X_{PCAwhite,i} = \frac{X_{rot,i}}{\sqrt{\lambda_i + e}}, e = 1.0e^{-5} \tag{4}$$

ZCA whitening:

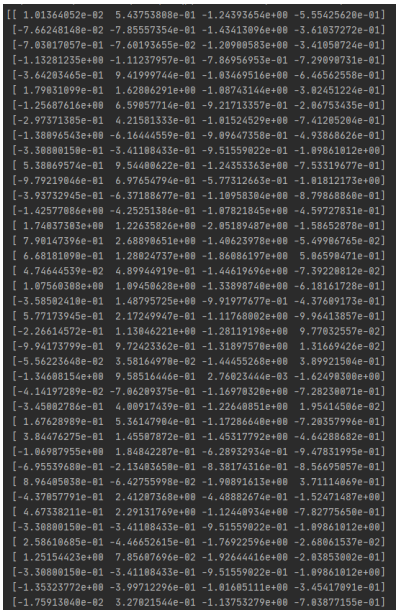
$$X_{ZCAwhite,i} = U X_{PCAwhite,i} \tag{5}$$

3. State the equation(s) to whiten a (new) data example x, given the ZCA parameters.
4. Compute and report the ZCA whitening parameters for the unnormalized IRIS data (including numerical values!).

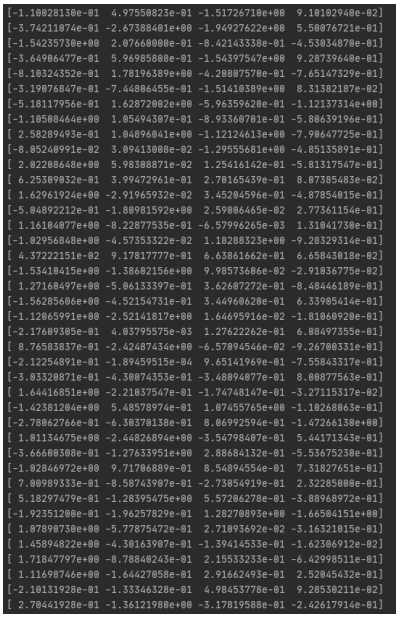
```
import numpy as np
# get data
iris = np.loadtxt("./dataSets/iris.txt", delimiter=',')
iris_set = iris[:, 0:4]
iris_label = iris[:, 4]


def ZCA_whitening(x):
    avg = np.mean(x, axis=0)
    x = x - avg
    sigma = np.cov(x, rowvar=0)
    u,s,v = np.linalg.svd(sigma)
    xRot = x@u
    xPCAwhite = xRot / np.sqrt(s + 1e-5)
    xZCAwhite = (u@xPCAwhite.T).T
```

```
    return xPCAwhite,xZCAwhite


xPCAwhite, xZCAwhite = ZCA_whitening(iris_set)
print(xZCAwhite)
```

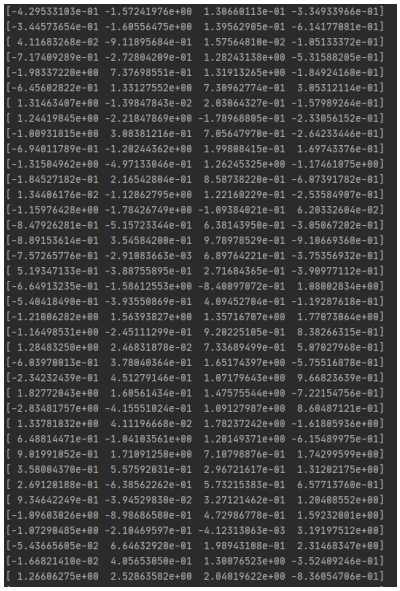
result:



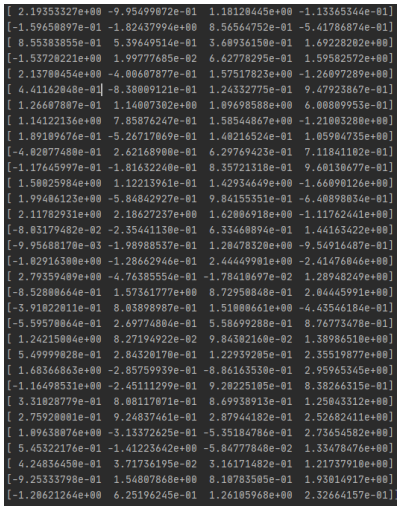
(a) data1.



(b) data2.



(c) data3.



(d) data4.

Figure 3: ZCA

3f)

In general, PCA is suitable for linear dimensionality reduction of the data. Kernel Principal Component Analysis enables non-linear dimensionality reduction of data and is used to deal with linear distinguishable data set.The basic idea of KPCA is that for a input matrix X, by using a nonlinear mapping we map all the samples in X to a high or even infinite dimensional space to making it linearly distinguishable, and then process the data in this high-dimensional space using PCA.

These are the steps for the implementation of the kernel PCA algorithm:
1. Choose a kernel mapping $k(\mathbf{x}_m, \mathbf{x}_n)$.
2. Obtain$\mathbf{K}$ based on training data $\{\mathbf{x}_n, (n = 1, \cdots, N)\}$.
3. Solve eigenvalue problem of $\mathbf{K}$ to get λ_i and $\mathbf{a}_i$.
4. For each given data point $\mathbf{x}$, obtain its principal components in the feature space: $(f(\mathbf{x}) \cdot \phi_i) = \sum_{n=1}^{N} a_n^{(i)} k(\mathbf{x}, \mathbf{x}_n)$
5. Do whatever processing (e.g., feature selection, classification) in the feature space.
(Reference: http://fourier.eng.hmc.edu/e161/lectures/kernelPCA/node4.html)

Limits: KPCA coats more time than PCA