EXERCISE 3

STATISTICAL MACHINE LEARNING

Giray Salgir and Gökberk Gül

Wednesday 30th June, 2021

1 Linear Regression

1.a

1) Ridge regression is used to prevents overfitting of testing data while adding a small bias to the estimation, prediction of testing data is less erroneous.

2) In order to find the parameters of the equation, we will used the close form solution of the parameter estimation which is explained in the lecture. Instead of applying gradient descent to the datas, we take the gradiend of the cost function and obtain the closed form solution.

$$\hat{\mathbf{w}} = \left(\hat{\mathbf{X}}\hat{\mathbf{X}}^{\mathsf{T}} + \lambda \mathbf{I}\right)^{-1} \hat{\mathbf{X}}\mathbf{y}$$

Figure 1: Closed Form solution of the gradient descent algorithm

In this algorithm $\hat{X}$ is the values of the input datas inserted in corresponding basis functions. λ is the ridge coefficient and y values are the corresponding outputs of the datas. Following Python function does this operation.

```
def linear_ridge_regression(lamda,dataset):
    x_hat = np.ones([2,len(dataset)])
    x_hat[0,:] = dataset[:,0]
    x_hat_transpose = np.transpose(x_hat)
    y = np.zeros([len(dataset),1])
    y = dataset[:,1]
    identity = np.identity(2)
    estimate1 = np.linalg.inv(np.matmul(x_hat,x_hat_transpose)+lamda*identity)
    estimate2 = np.matmul(estimate1,x_hat)
    estimate3 = np.matmul(estimate2,y)
    x = np.linspace(-1,1,500)
    y_estimate = estimate3[0]*x + estimate3[1]
    return x,y_estimate,estimate3
```

After using the function following values of the w(parameter) vector is found;

$$w(0) = -0.3094680556628061$$

$$w(1) = 1.1910947846925701$$

This means that our line has a slope of -0.30 and has a y-intercept as 1.19.

3) In order to find the RMSE of the testing and training data following python script is written.

```
def root_mean_squared_error(f,dataset):
    rmse = 0
    for i in range(len(dataset)):
        y_est = f(dataset[i,0])
        squared_distance = y_est - dataset[i,1]
        squared_distance = squared_distance*squared_distance
        rmse = rmse + squared_distance
    rmse = math.sqrt(rmse/len(dataset))
    return rmse
```

We applied this function to our dataset and following results are obtained.

$$RMSEfortraininglinear : 0.41217801567361073$$

$$RMSEfortestinglinear : 0.3842881699259788$$

As it can be seen from the datas, testing has better performance than the training datas which is probably the affect of ridge coefficient.

4)

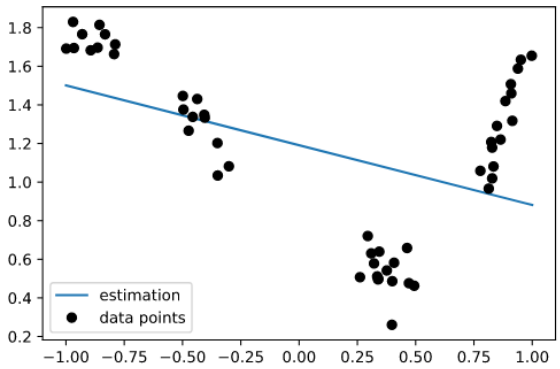

Figure 2: Linear Ridge Regression

1.b

In order to simulate polynomial feature projection of order 2,3 and 4 we write 3 different methods which have different basis functions so that we can estimate parameters. Following Python codes are the methods of this.

```
def linear_ridge_regression_second_order(lamda,dataset):
    x_hat = np.ones([3,len(dataset)])
    x_hat[0,:]=dataset[:,0]*dataset[:,0]
    x_hat[1,:] = dataset[:,0]
    x_hat_transpose = np.transpose(x_hat)
    y = np.zeros([len(dataset),1])
    y = dataset[:,1]
    identity = np.identity(3)
    estimate1 = np.linalg.inv(np.matmul(x_hat,x_hat_transpose)+lamda*identity)
    estimate2 = np.matmul(estimate1,x_hat)
    estimate3 = np.matmul(estimate2,y)
    x = np.linspace(-1,1,500)
    y_estimate = estimate3[0]*(x)*(x) + estimate3[1]*x + estimate3[2]
    return x,y_estimate

def linear_ridge_regression_third_order(lamda,dataset):
    x_hat = np.ones([4,len(dataset)])
    x_hat[0,:]=dataset[:,0]*dataset[:,0]*dataset[:,0]
    x_hat[1,:] = dataset[:,0]*dataset[:,0]
    x_hat[2,:] = dataset[:,0]
    x_hat_transpose = np.transpose(x_hat)
    y = np.zeros([len(dataset),1])
    y = dataset[:,1]
    identity = np.identity(4)
    estimate1 = np.linalg.inv(np.matmul(x_hat,x_hat_transpose)+lamda*identity)
    estimate2 = np.matmul(estimate1,x_hat)
    estimate3 = np.matmul(estimate2,y)
    x = np.linspace(-1,1,500)
    y_estimate = estimate3[0]*(x)*(x)*(x) + estimate3[1]*x*x + estimate3[2]*x + estimate3[3]
    return x,y_estimate

def linear_ridge_regression_fourth_order(lamda,dataset):
    x_hat = np.ones([5,len(dataset)])
    x_hat[0,:]=dataset[:,0]*dataset[:,0]*dataset[:,0]*dataset[:,0]
    x_hat[1,:] = dataset[:,0]*dataset[:,0]*dataset[:,0]
    x_hat[2,:] = dataset[:,0]*dataset[:,0]
    x_hat[3,:] = dataset[:,0]
    x_hat_transpose = np.transpose(x_hat)
    y = np.zeros([len(dataset),1])
    y = dataset[:,1]
    identity = np.identity(5)
    estimate1 = np.linalg.inv(np.matmul(x_hat,x_hat_transpose)+lamda*identity)
    estimate2 = np.matmul(estimate1,x_hat)
    estimate3 = np.matmul(estimate2,y)
```

```
x_sort = np.linspace(-1,1,500)
y_estimate = estimate3[0]*(x_sort)*(x_sort)*(x_sort)*(x_sort) + estimate3[1]*x_sort*x_sort*x_
return x_sort,y_estimate
```

These methods uses the same formula but now it takes square, power of three or four depending on the basis functions. We will now look at the RMSE and regression lines of these polynomials.

For Polynomial order 2:

Regression Line is the following.

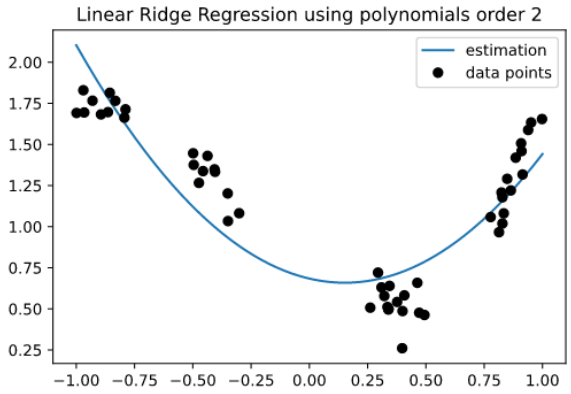

Figure 3: Linear Ridge Regression with polynomial degree 2

We will now look at the RMSE value of training and testing data.

$$RMSE for training polyomial order 2 : 0.2120144652937695$$

$$RMSE for testing polynomial order 2 : 0.21687121736099577$$

For Polynomial order 3:

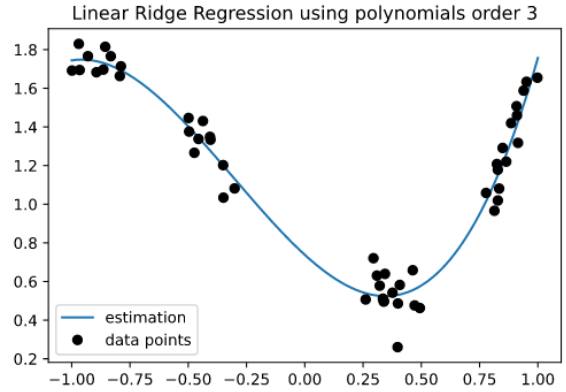

Figure 4: Linear Ridge Regression with polynomial degree 3

We will now look at the RMSE value of training and testing data.

$$RMSE for training polyomial order 3 : 0.08706846203963327$$

$$RMSE for testing polynomial order 3 : 0.10835624231161368$$

For Polynomial order 4:

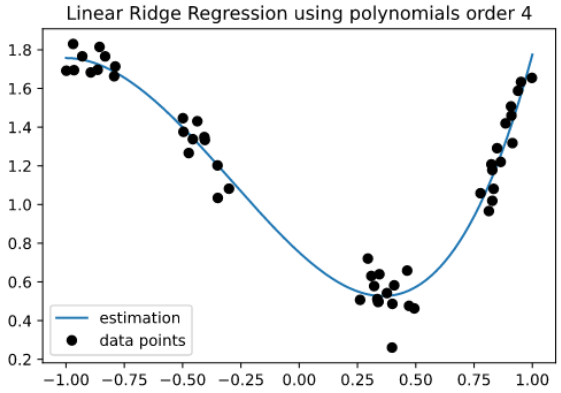

Figure 5: Linear Ridge Regression with polynomial degree 4

/

We will now look at the RMSE value of training and testing data.

$$RMSE for training polyomial order 4 : 0.08701290017643422$$

$$RMSE for testing polynomial order 4 : 0.10666047984567603$$

3) We call this method linear because the parameters are linear.

1.c

	First Order Polynomial	Second Order Polynomial	Third Order Polynomial	Fourth Order Polynomial
Average Train RMSE	0.3341	0.1549	0.069	0.0665
Average Validation RMSE	0.5505	0.2634	0.1185	0.1378
Average Test RMSE	0.4328	0.2681	0.1418	0.1546

Figure 6: Cross Validation

/

According to the table above, results met my expectations because our data is more suitable for higher order polynomials rather than first and second order. We can observe that as the degree of polynomial increases the avergare RMSE decreases for both validation, train and test sets.

Using the values from table I would choose third order polynomial because it has the lowest RMSE values among others but fourth order has also very low RMSE values compared to others. Because third order has slightly low RMSE, it would be a better choice for the given data.

1.d

1) Posterior Distrubution is the following;

$$p(w|X, y, \alpha, \beta) = p(y|w, X, \beta) * N(w|0, \alpha^{-1})$$

where N is the normal distrubution

2) Predictive Distrubution is the following;

$$p\left(y_* \left| \mathbf{x}_*, \mathbf{X}, \mathbf{y}\right.\right) = \mathcal{N}\left(\mathbf{y}_* \left| \mu\left(\mathbf{x}_*\right), \sigma^2\left(\mathbf{x}_*\right)\right.\right)$$

$$\mu\left(\mathbf{x}_*\right) = \phi^T\left(\mathbf{x}_*\right)\left(\frac{\alpha}{\beta}\mathbf{I} + \Phi\Phi^\mathsf{T}\right)^{-1}\Phi^\mathsf{T}\mathbf{y}$$

$$\sigma^2\left(\mathbf{x}_*\right) = \frac{1}{\beta} + \phi^\mathsf{T}\left(\mathbf{x}_*\right)\left(\alpha\mathbf{I} + \beta\Phi\Phi^\mathsf{T}\right)^{-1}\phi\left(\mathbf{x}_*\right)$$

Figure 7: Predictive Distrubution

/

This image is taken from the Regression Slides from the SML class.

3) Using the Bayesian model, following RMSE values for train and test datas is found;

$$RMSE for training : 0.27949358097$$

$$RMSE for testing : 0.19957389375$$

We can observe that when we use Bayesian approach RMSE decreases compared to linear ridge regression.

4)Average log-likelihood of the train and test data is the following;

$$log - likelihood for training : -1485.903845$$

$$log - likelihood for testing : -1465.937458$$

It is observed that log-likelihood of the testing data is better than the training data

5)

6) Linear regression uses frequentist aproach of the data in which the occurance of the data is used to predict the data. There is no prior knowledge about the parameters. However, in Bayesian Linear regression we know prior knowledge about the parameters. Thus using these prior beliefs we predict the posterior distrubition of the parameters.

1.e

1.f

2 Linear Classifications

2.a

In the generative model, we are interested in the underlying probability distribution $p(x, y)$ where x is our input and y is our label. Then, we use $p(y)$ and likelihood $P(x|y)$ to apply

Bayes rule and estimate $P(y|x)$. An example for this would be Naive Bayes Classifier. In the discriminative case, we are interested in directly finding the probability $P(x|y)$ with our data. An example is Logistic Regression. Generally speaking, learning discriminative models are easier since we only draw a boundary between classes and not try to estimate the underlying probability distributions, which requires prior knowledge/estimate about the data.

2.b

According to the lecture slides, Fisher's linear discriminant is Bayes optimal if the class-conditional distributions are equal, with diagonal covariance, equivalent to Linear Discriminant Analysis (LDA). We formalize our model with below assumptions.

$$p(Y = y|X = x) = \frac{f_1(x)\pi_x}{P(X)}$$

$$f_k(x) = \frac{1}{(2\pi)^2|\mathbf{\Sigma_k}|^{1/2}} exp(-\frac{1}{2}(x-\mu_k)^T \mathbf{\Sigma_k}^{-1}(x-\mu)) \quad \textit{Input is Gaussian}$$

$$\textit{Assume } \Sigma_k = \Sigma$$

$$\textit{We compare } f_k(x)\pi_k \textit{ with } f_n(x)\pi_n$$

$$\textit{Finding log-decision boundary reduces to}$$

$$h^*(x) = \textit{argmax}_k \delta_k(x)$$

$$\delta_k(x) = x^T\mathbf{\Sigma}^{-1}\mu_k - \frac{1}{2}\mu_k^T\mathbf{\Sigma}^{-1}\mu_k + \log\pi_k$$

$$\pi_k = \frac{N_k}{N}$$

$$\mu_k = \frac{1}{N_k}\sum_{i \epsilon y_i = k} x_i$$

$$\Sigma_k = \frac{1}{N_k - K}\sum_{i \epsilon y_i = k}(x_i - \mu_k)(x_i - \mu_k)^T$$

$$\mathbf{\Sigma} = \frac{\sum_{i=1}^{K} N_i \Sigma_i}{N}$$

Our code for learning the parameters for the given dataset is

```
def find_parameters(dataset, class_seperator, number_of_classes):
    classes = []
    mus = []
    pis = []
    final_cov = 0
    for i in range(number_of_classes):
        if i == 0:
            gauss_class = dataset[0:class_seperator[i],:]
        elif i == (number_of_classes - 1):
```

```
            gauss_class = dataset[class_seperator[i-1]:,:]
        else:
            gauss_class = dataset[class_seperator[i-1]:class_seperator[i],:]
        mu_temp = np.mean(gauss_class, axis=0)
        cov_temp = np.cov(gauss_class, rowvar=0)
        pi_temp = len(gauss_class)/len(dataset)
        final_cov += cov_temp*len(gauss_class)
        mus.append(mu_temp)
        pis.append(pi_temp)
        classes.append(gauss_class)

    final_cov = final_cov/len(dataset)
    return classes, mus, final_cov, pis
```

Here, the input *class_seperator* is just holding the index where the data of the new classes begin. For our case it is $class\_seperator = [50, 93]$. Then, we make the classification for single input with

```
def classify(data, sigma, mus, pis):
    likelihood = np.zeros(len(mus))
    for i in range(len(mus)):
        likelihood_temp = (data.dot(np.linalg.inv(sigma))).dot(mus[i])
        - 1/2*(mus[i].dot(np.linalg.inv(sigma))).dot(mus[i]) + np.log(pis[i])
        likelihood[i] = likelihood_temp
    return np.argmax(likelihood)
```

Plots for the originial dataset and our LDA classified data can be found in Figures 8 and 9. Our number of misclassified samples are 19 out of 137.

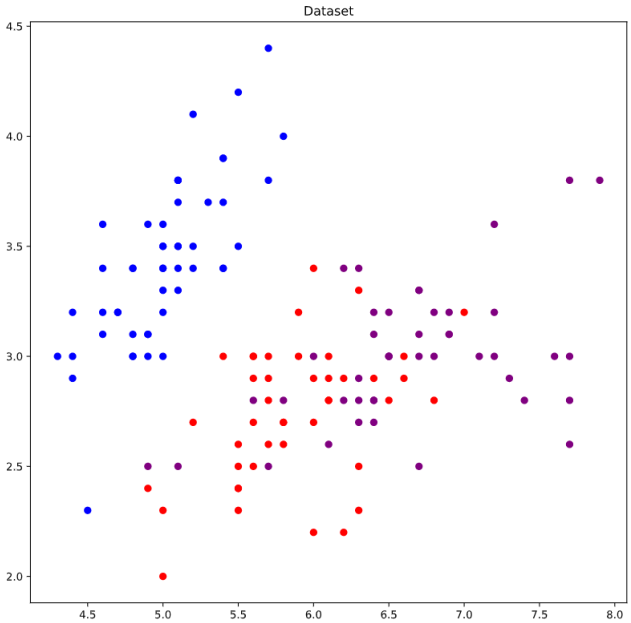

Figure 8: Original Dataset

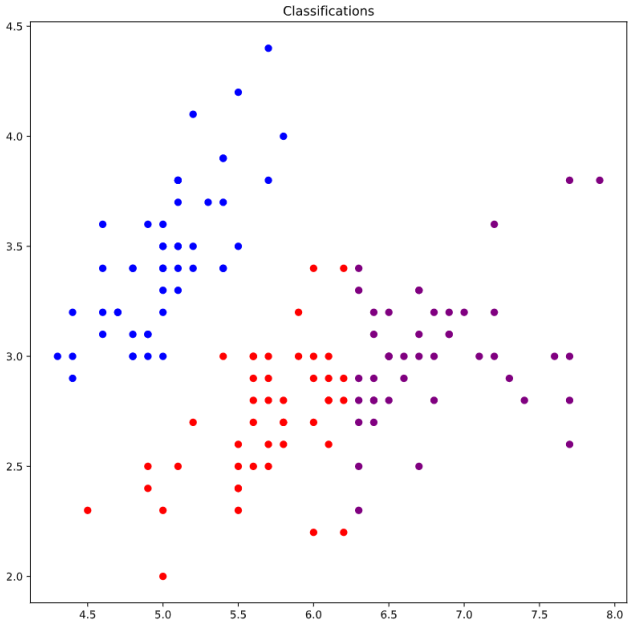

Figure 9: LDA Classified Data

3 Principal Component Analysis

3.a

Think of dataset with two of features are weight in kg and length in mm. When we compute the variance, numerically, length feature might have higher variance. However, when we normalize both of the features, we might see that weight feature has more variance to it since we can now relatively compare them. That is why normalizing the dataset is crucial before applying the machine learning algorithms. Calculation for normalization and code for our normalization is given below where X is only the features of the dataset, without the labels.

$$x_{normalized} = \frac{x - \mu_x}{\sigma_x}$$

```
means = np.mean(X, axis=0)
variances = np.var(X, axis=0)
X_normalized = (X - means)/np.sqrt(variances)
```

3.b

PCA aims to find the variances of components (features) from dataset, and take the first K component with largest variances to represent the data while transforming our space from N dimentional to K dimentional. PCA algorithm is;

Normalize dataset (Code given in part a

$\boldsymbol{\Sigma} = \boldsymbol{X}^T\boldsymbol{X}$ *(Covariance Matrix*

Find eigen values λ *and eigen vectors* v *from* $\boldsymbol{\Sigma}$

Sort eigen-pairs wrt. eigen values

Calculate cumulative variance of K components as $\frac{\sum_{i=1}^{K}\lambda_i}{\sum_{j=1}^{N}\lambda_j}$

Generate transition matrix $\boldsymbol{W}$ *from first K eigen vectors*

$$\boldsymbol{X_{new}} = \boldsymbol{XW}$$

Code for our PCA algorithm is given below.

```
def PCA(X, cumulative_var = 95, no_of_components = 4, use_cum_var = True, plot = False):
    # Compute and sort eigenpairs
    cov_mat = (X.T).dot(X)/len(X)
    eig_vals, eig_vecs = np.linalg.eig(cov_mat)
    eig_vals_sort_indices = eig_vals.argsort()
    eig_vals = eig_vals[eig_vals_sort_indices[::-1]]
    eig_vecs = eig_vecs[eig_vals_sort_indices[::-1], :]
```

```
# Compute cumulative variance
total = sum(eig_vals)
eig_vars = (eig_vals/total)*100
cum_vars = np.zeros(len(eig_vals))
cum_vars[0] = eig_vars[0]
for i in range(1, len(eig_vals)):
    cum_vars[i] = cum_vars[i-1] + eig_vars[i]

if plot:
    plt.plot(np.arange(4) + 1, cum_vars)
    plt.xlabel('Number of Components')
    plt.ylabel('Cumulative Variance')

# Find how many components to take
if (use_cum_var == True):
    k = np.argmax(cum_vars >= cumulative_var)
else:
    k = no_of_components - 1

# Project
W = eig_vecs[:, 0:k+1]
X_new = X.dot(W)

return X_new, W
```

Cumulative Variance vs Number of Components taken plot is given in Figure 10

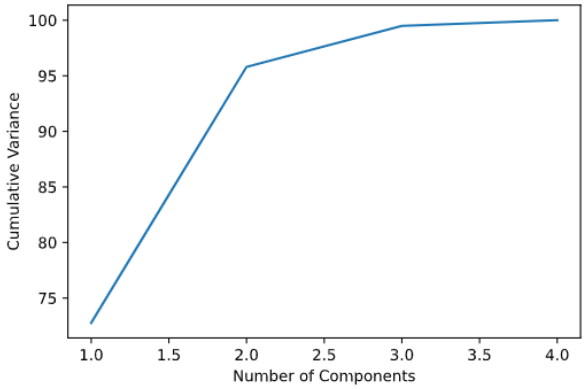

Figure 10: LDA Cumulative Variance vs Number of Components Data

For our case, more than 95% of the variance is preserved while taking the first 2 components.

3.c

As explained in the previous section, we only need 2 components to explain the 95% of variance of dataset. We apply the transformation by simply multiplying $\boldsymbol{XW}$. The resulting scatter plot is

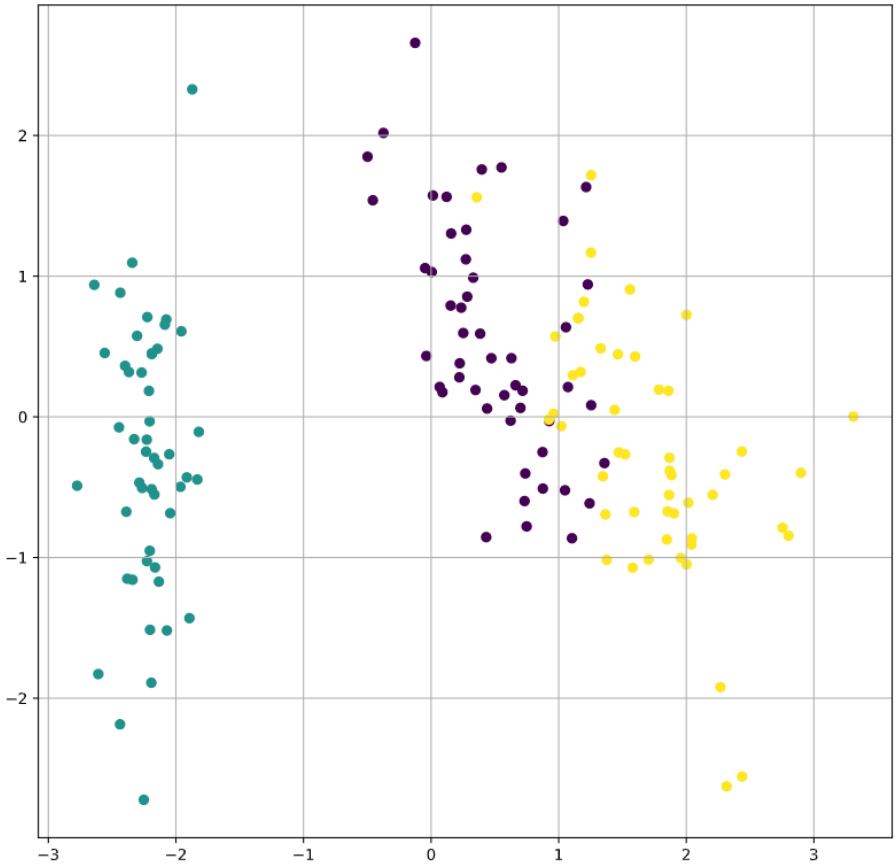

Figure 11: Scatter Plot of PCA Applied IRIS Dataset

Code for applying PCA is already discussed in part b. We have observed that the true nature of data is still intact, however now we are able to represent with only 2 components. If one were to apply classification algorithms to the PCA applied dataset, it would still be successful.

3.d

NRMSE values for each component can be calculated as

$$NRMSE_k = \frac{RMSE_k}{\sigma_k} = \frac{1}{\sigma_k}\frac{\sqrt{\sum_{i=1}^{N}(x_{ki} - \hat{x_{ki}})^2}}{N}$$

For the reconstruction of original data, we use the formula

$$X_{reconstructed} = XW^T\sqrt{\Sigma} + \mu$$

Below is our code for applying PCA for 1-4 components, then reconstructing the data and calculating the NRMSE with the original data. Code outputs 4x1 vector for each calculation corresponds to the error level for each component (feature).

```
X_1, W_1 = PCA(X_normalized, no_of_components = 1, use_cum_var = False, plot = False)
X_2, W_2 = PCA(X_normalized, no_of_components = 2, use_cum_var = False, plot = False)
X_3, W_3 = PCA(X_normalized, no_of_components = 3, use_cum_var = False, plot = False)
X_4, W_4 = PCA(X_normalized, no_of_components = 4, use_cum_var = False, plot = False)

X_1_recons = X_1.dot(W_1.T)*np.sqrt(variances) + means
NMRSE_1 = np.sum((np.square(X - X_1_recons)/len(X)), axis=0)/np.sqrt(variances)
X_2_recons = X_2.dot(W_2.T)*np.sqrt(variances) + means
NMRSE_2 = np.sum((np.square(X - X_2_recons)/len(X)), axis=0)/np.sqrt(variances)
X_3_recons = X_3.dot(W_3.T)*np.sqrt(variances) + means
NMRSE_3 = np.sum((np.square(X - X_3_recons)/len(X)), axis=0)/np.sqrt(variances)
X_4_recons = X_4.dot(W_4.T)*np.sqrt(variances) + means
NMRSE_4 = np.sum((np.square(X - X_4_recons)/len(X)), axis=0)/np.sqrt(variances)
```

Results are given in the table below. As it can be seen, the more components we use, the less information we lose.

No of Components	x_1	x_2	x_3	x_4
1	0.1697801	0.3449039	0.02912477	0.05231732
2	0.06438868	0.00386749	0.02840389	0.0493189
3	0.00116743	0.00013723	0.0232601	0.00429639
4	0	0	0	0

3.e

1) In PCA whitening the redundancy in the dataset is reduced by reducing the dimensionality of the dataset.However, in ZCA whitening the the dimensionality of the data remains same and apply an other technique to reduce redundancy.

2) We need three parameters, namemly U,S,Σ and ϵ. U is any orthagonal matrix such that; U*U^t = I(Identity). We can also find Σ using the following equation; 1/m *$\sum_{i=1}^{m} x^i * x^i t$. S matrix is a diagonal matrix whose dimensions are the eigenvalues of the Σ. Finally ϵ is given for numerical stability. As we can see using the input values we can find the parameters of the ZCA whitening using the input values.

3.f

Kernel PCA uses kernel functions to project the non-linear dataset to a higher dimensional space. In Kernel PCA in order to decrease the dimensinonality of the space, we first increase it to a

higher dimensional space and than decrease it to a lower dimensional space. Its main limitation is the computational complexity because extrating the principal component in kernal PCA takes too much time compared to the regular PCA.