Supervised Learning (COMP0078) – Coursework 1

Bowen Ding (21118015)
Zhengxiang Wei (21181074)

November 15, 2021

Abstract

This is the answer report for Supervised Learning (COMP0078) Coursework 1. Answers are arranged in the order of Part I, Part II and Part III. All the code are attached at the end of this report with a single jupyter notebook file attached in the submission of work.

Contents

Introduction

In this submission, our coding is done with jupyter notebook, using python 3.9 as the kernel language. As required, we have implemented the Regression and $k-NN$ by writing our own functions and classes. To clarify that no packages that was not allowed were used in our work, here is a list of packages we used in this coursework:

- Numpy, for mathematical processing with data such as calculating, sorting and etc.;
- Numba.njit, for accelerating the speed of running the code in part II with lots for loops;
- matplotlib.pyplot, to give plots required in the coursework;

1 Part I

In this section question 1-5 would be answered.

1.1 Linear Regression

Considering a known data set of $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_1), ..., (\boldsymbol{x}_m, y_m)\}$,where $\boldsymbol{x_i} = (x_{i1}, x_{i2}, ..., x_{in})^T$ is a input sample point with n dimensions while corresponding y_i is a real number, which is the label of $\boldsymbol{x_i}$. It is shown in data set D, m points with labels are sampled independently and from an identical distribution. In linear regression with basis functions, with the help of basis functions($\phi_1(\boldsymbol{x}), ..., \phi_N(\boldsymbol{x})$, where $\phi_{\boldsymbol{i}}(\boldsymbol{x}) : R^n \to R$), we can define a feature map $\boldsymbol{\phi}(\boldsymbol{x}) = (\phi_1(\boldsymbol{x}), ..., \phi_k(\boldsymbol{x}))^T$ and mapped data matrix $A \in R^{m \times k}$:

$$\mathrm{A} := \begin{bmatrix} \phi(\boldsymbol{x}_1)^T \\ \vdots \\ \phi(\boldsymbol{x_m})^T \end{bmatrix} = \begin{bmatrix} \phi_1(\boldsymbol{x}_1) & ... & \phi_k(\boldsymbol{x}_1) \\ \vdots & ... & \vdots \\ \phi_1(\boldsymbol{x}_m) & ... & \phi_k(\boldsymbol{x}_m) \end{bmatrix}$$

The main idea of linear regression is to find function $f(\boldsymbol{x}_i) = w^T \boldsymbol{\phi}(\boldsymbol{x}_i)$ which is a linear combination of $\phi_j(\boldsymbol{x}_i), j = 1...k$ to fit y_i. We can calculate regression coefficients by the formula below:

$$w = \left(A^T A\right)^{-1} A^T y$$

where $y = (y_1, y_2, ..., y_m)^T$

Question 1

In this question, we have the data set $\{(1,3), (2,2), (3,0), (4,5)\}$ after applying k-order polynomial basis functions, we get mapped data matrix A:

$$A = \begin{bmatrix} \boldsymbol{x}_1{}^0 & \boldsymbol{x}_1{}^1 & \cdots & \boldsymbol{x}_1{}^{\boldsymbol{j}} & \cdots & \boldsymbol{x}_1{}^{\boldsymbol{k}-1} \\ \boldsymbol{x}_2{}^0 & \boldsymbol{x}_2{}^1 & \cdots & \boldsymbol{x}_2{}^{\boldsymbol{j}} & \cdots & \boldsymbol{x}_2{}^{\boldsymbol{k}-1} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \boldsymbol{x_i}{}^0 & \boldsymbol{x_i}{}^1 & \cdots & \boldsymbol{x_i}{}^{\boldsymbol{j}} & \cdots & \boldsymbol{x_i}{}^{\boldsymbol{k}-1} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \boldsymbol{x_m}{}^0 & \boldsymbol{x_m}{}^1 & \cdots & \boldsymbol{x_m}{}^{\boldsymbol{j}} & \cdots & \boldsymbol{x_m}{}^{\boldsymbol{k}-1} \end{bmatrix}$$

(a) We superimpose the curves with polynomial basis of k=1,2,3,4 respectively to fit over four data points.

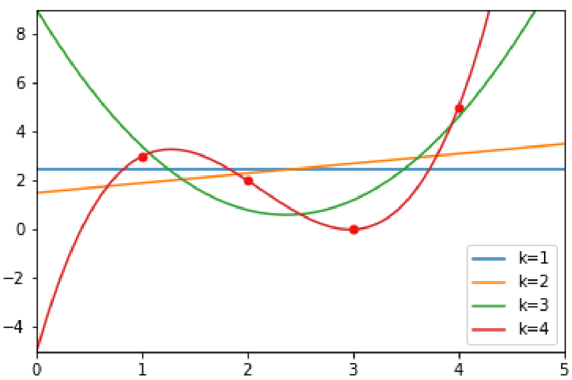

Figure 1: Four different fitting curves with polynomial basis of order=1,2,3,4

(b) The equations corresponding to the curves fitted for k=1,2,3 are as below:

- k=1, $f(x) = 2.5$
- k=2, $f(x) = 1.5 + 0.4x$
- k=3, $f(x) = 9 - 7.1x + 1.5x^2$

(c) We use $mse = \frac{1}{m}\sum_{i=1}^{m}\left(y_i - \sum_{j=1}^{k} w_j \phi_j(x_i)\right)^2$ to measure loss. The lower the value of mse, the better the algorithm fits.

- k=1, mse=3.25
- k=2, mse=3.05
- k=3, mse=0.80
- k=4, mse=5.29×10^{-24}

Question 2

In question2, we explore the overfitting phenomena. Firstly, we generate sample $\boldsymbol{x}_i$ uniformly at random from the interval $[0, 1]$ 30 times and apply $g_\sigma(\boldsymbol{x}_i) := \sin^2(2\pi\boldsymbol{x}_i) + \varepsilon$ to generate training set, where $\varepsilon \sim N(0, \sigma^2)$ serves as noise . The training set is as below:

$$S_{0.07,30} = \{(\boldsymbol{x}_1, g_{0.07}(\boldsymbol{x}_1)), (\boldsymbol{x}_2, g_{0.07}(\boldsymbol{x}_2)), ..., (\boldsymbol{x}_{30}, g_{0.07}(\boldsymbol{x}_{30}))\}$$

(a) i. The graph of the function $\sin^2(2\pi x)$ and scatter of samples in training set.

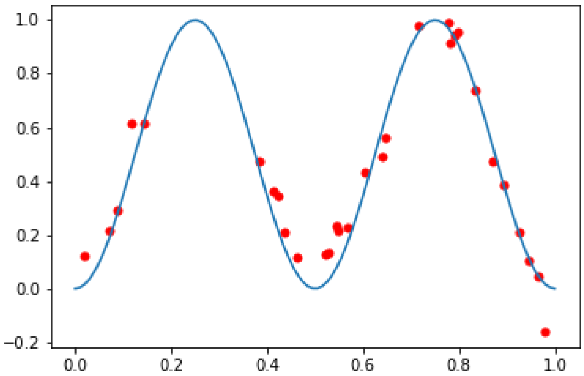

Figure 2: $\sin^2(2\pi x)$ and training set samples

(a) ii. We fit the training set with a polynomial basis of dimension k=2,5,10,14,18

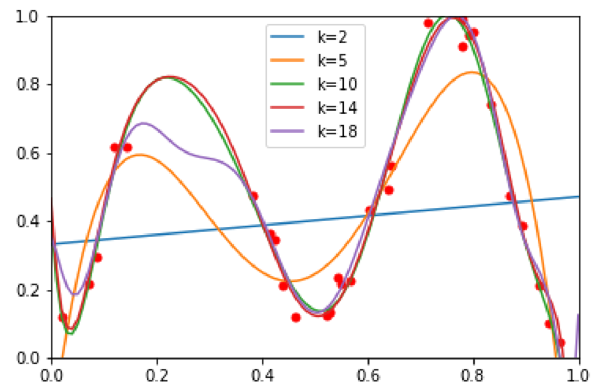

Figure 3: Polynomial fit curves of k=2,5,10,14,18

(b) Using $\ln(mse)$ to measure loss in the training set against polynomial dimensions.

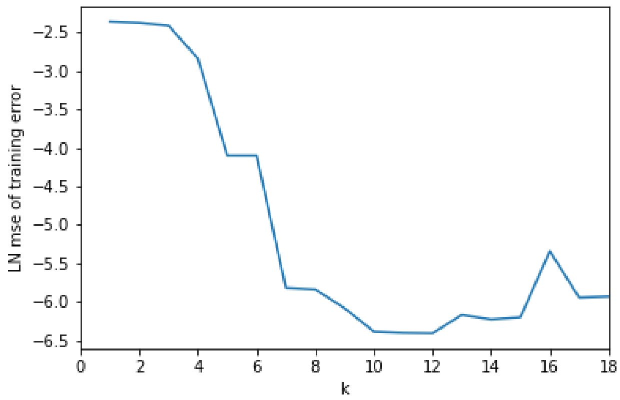

Figure 4: Training set loss against k=1...18

(c) In order to test whether the model begins to fit the noise rather than fitting the function, with model's complexity(dimension of polynomial) increasing, we generate a test set T with 1000 points:

$$T_{0.07,1000} = \{(\boldsymbol{x}_1, g_{0.07}(\boldsymbol{x}_1)), (\boldsymbol{x}_2, g_{0.07}(\boldsymbol{x}_2)), ..., (\boldsymbol{x}_{1000}, g_{0.07}(\boldsymbol{x}_{1000}))\}$$

Then we use $\ln(mse)$ to measure loss in the test set against polynomial dimensions.

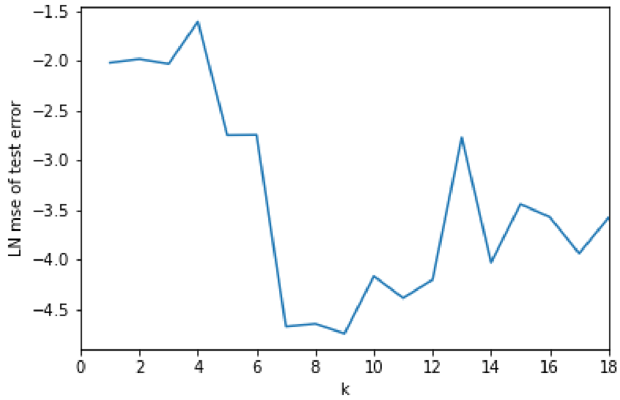

Figure 5: Test set loss against k=1...18

(d) We repeat (b) and (c) 100 times and use ln(average results of a 100 runs) to measure the loss both in training set and test set.

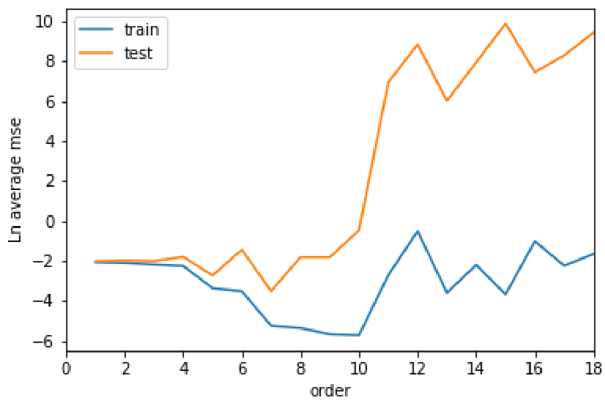

Figure 6: ln(average mse of a 100 runs) for training and test set

Question 3

In this case, the mapped data matrix A could be:

$$A = \begin{bmatrix} \sin(\pi \boldsymbol{x}_1) & \sin(2\pi \boldsymbol{x}_1) & \cdots & \sin(j\pi \boldsymbol{x}_1) & \cdots & \sin(k\pi \boldsymbol{x}_1) \\ \sin(\pi \boldsymbol{x}_2) & \sin(2\pi \boldsymbol{x}_2) & \cdots & \sin(j\pi \boldsymbol{x}_2) & \cdots & \sin(k\pi \boldsymbol{x}_2) \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \sin(\pi \boldsymbol{x}_i) & \sin(2\pi \boldsymbol{x}_i) & \cdots & \sin(j\pi \boldsymbol{x}_i) & \cdots & \sin(k\pi \boldsymbol{x}_i) \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \sin(\pi \boldsymbol{x}_m) & \sin(2\pi \boldsymbol{x}_m) & \cdots & \sin(j\pi \boldsymbol{x}_m) & \cdots & \sin(k\pi \boldsymbol{x}_m) \end{bmatrix}$$

From the picture below, we can see $\ln(mse)$ to measure loss in the training set and test set against polynomial dimensions for one round.

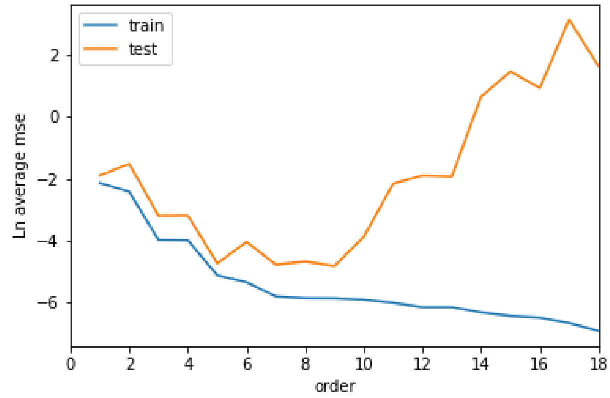

Figure 7: ln(mse) for training and test set, $\sin(k\pi x)$ basis functions

We also repeat 100 rounds,

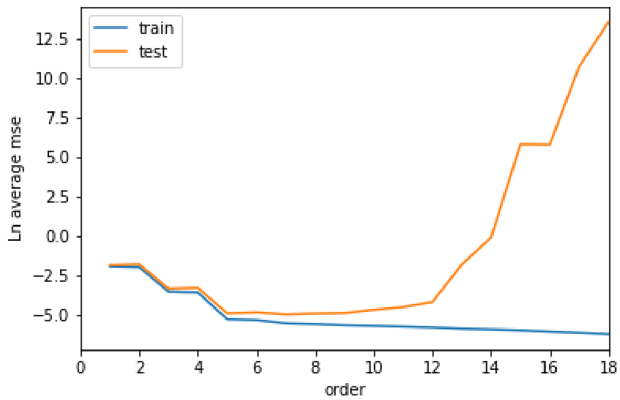

Figure 8: ln(average mse of 100 runs) for training and test set, $\sin(k\pi x)$ basis functions

1.2 Filtered Boston housing and kernels

Question 4

Compare the performance among different regression methods. We split the data set into training set (2/3 of total data set) and test set (1/3 of total data set), and perform regression over 20 runs and calculate the average mse of every runs.

a. Perform naive regression on the training set and calculate the mse on the training and test sets.

In the case of Naive Regression. The basis function of this case is $\boldsymbol{\phi}(\boldsymbol{x}) = \{1\}$, which means mapped matrix A is :

$$A = \begin{bmatrix} 1 \\ 1 \\ ... \\ 1 \end{bmatrix} \in R^m$$

- MSE on the training set: 82.87
- MSE on the test set: 86.92

b. Give a simple interpretation of the constant function in a.
Since,

$$w = \left(A^T A\right)^{-1} A^T y$$

Hence,

$$w = \frac{1}{m}\sum_{i=1}^{m} y_i$$

$$predict = Aw = \begin{bmatrix} \frac{1}{m}\sum_{i=1}^{m} y_i \\ \frac{1}{m}\sum_{i=1}^{m} y_i \\ ... \\ \frac{1}{m}\sum_{i=1}^{m} y_i \end{bmatrix}$$

We can find that the constant function in (a) is

$$f(\boldsymbol{x}_i) = \frac{1}{m}\sum_{i=1}^{m} y_i$$

which is the mean of the label of inputted samples.
c. Linear Regression with single attributes.
The fit function of this case could be

$$f(\boldsymbol{x}_i) = w_0 + w_1 x_{ij}, i = 1...m, j = 1...12$$

where x_{ij} means the value of i-th sample point's j-th attribute.

- attribute1: MSE train 70.53869529 MSE test 73.00817819
- attribute2: MSE train 71.96287247 MSE test 75.48257325
- attribute3: MSE train 63.25866244 MSE test 66.65348237
- attribute4: MSE train 80.61347746 MSE test 82.74184527
- attribute5: MSE train 67.55052457 MSE test 70.96812434
- attribute6: MSE train 43.44303058 MSE test 43.13931893
- attribute7: MSE train 70.54381803 MSE test 75.17310599
- attribute8: MSE train 77.51229332 MSE test 81.32391081
- attribute9: MSE train 70.9495549 MSE test 73.43982526
- attribute10: MSE train 64.83087928 MSE test 66.95009183
- attribute11: MSE train 61.58601723 MSE test 63.76119662
- attribute12: MSE train 37.28030312 MSE test 40.30809575

d. Linear Regression using all attributes.

- MSE train: 21.631067081403906
- MSE test: 21.791834593295146

1.3 Kernelised ridge regression

Considering that we have l examples, each sample point has n attributes, so we can try to find a weight of ridge regression by optimisation:

$$w^* = arg \min_{w \in R^n} \frac{1}{l}\sum_{i=1}^{l}\left(x_i^T w - y_i\right)^2 + \gamma w^T w$$

For a given kernel function K define the kernel matrix $\boldsymbol{K}$:

$$K_{i,j} := K(\boldsymbol{x}_i, \boldsymbol{x}_j), i, j = 1...l$$

The dual optimisation formulation after kernelization is

$$\alpha^* = arg \min_{\alpha \in R^l} \frac{1}{l}\sum_{i=1}^{l}\left(\sum_{j=1}^{l}\alpha_j K_{i,j} - y_i\right)^2 + \gamma \alpha^T K \alpha$$

We may solve α^* by:

$$\alpha^* = (K + \gamma l I_l)^{-1} y$$

and predict y by:

$$y_{test} = \sum_{i=1}^{l} \alpha_i^* K(\boldsymbol{x}_i, \boldsymbol{x}_{test})$$

Question 5

In this exercise, we perform kernel ridge regression on the data set with Gaussian kernel,

$$K(\boldsymbol{x_i}, \boldsymbol{x}_j) = \exp\left(-\frac{\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2}{2\sigma^2}\right)$$

We still hold 2/3 of total data for training and 1/3 for testing.
a. We perform kernel ridge regression on the training set using five-fold cross-validation to find the best pair of and ($\gamma \in \left[2^{-40}, 2^{-39}, ..., 2^{-26}\right], \sigma \in \left[2^{7}, 2^{7.5}, ..., 2^{12.5}, 2^{13}\right]$) ,whose test error of five-fold cross-validation is the lowest.

- best gamma: 2^{-33}
- best sigma: 2^{10}

b. We plot the "five-fold cross validation error" as a function of γ *and* σ

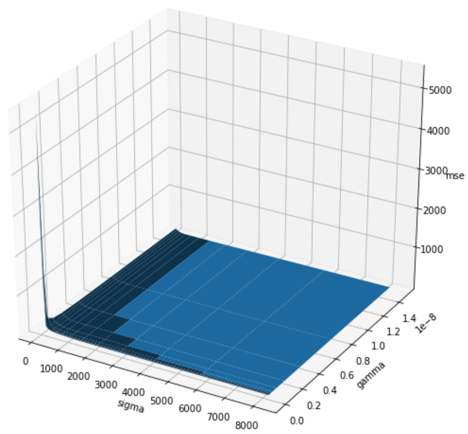

Figure 9: 5-fold cross validation mse of each pair of γ *and* σ

c.The MSE on training and test sets for the best and :

- MSE train: 22.634913880379973
- MSE test: 24.163437363863615

d. Comparison among different methods.(Average among 20 runs)

Method	MSE train	MSE test
Naive Regression	86.75±4.84	80.07±9.54
Linear Regression (attribute 1)	73.63±4.57	68.81±9.10
Linear Regression (attribute 2)	75.65±4.53	69.54±8.96
Linear Regression (attribute 3)	67.63±4.56	59.29±9.05
Linear Regression (attribute 4)	83.57±4.69	78.97±9.31
Linear Regression (attribute 5)	71.34±4.63	64.83±9.15
Linear Regression (attribute 6)	44.73±3.99	41.82±8.05
Linear Regression (attribute 7)	75.1±5.06	67.58±10.06
Linear Regression (attribute 8)	81.62±5.17	74.95±10.11
Linear Regression (attribute 9)	74.17±4.96	68.59±9.84
Linear Regression (attribute 10)	68.34±5.00	61.55±9.97
Linear Regression (attribute 11)	63.98±3.53	60.64±6.78
Linear Regression (attribute 12)	39.57±2.68	36.66±5.19
Linear Regression (all attributes)	22.49±2.07	23.77±4.68
Kernel Ridge Regression	8.44±1.65	12.15±2.14

2 Part II

To solve the questions in part II, we need to write a knn classifier first. Here we defined a knn classifier by sorting **euc_distance**, to improve its efficiency, we added **numba njit** (just in time compiler, no python mode) to enable fast compiling.

2.1 k-Nearest Neighbors

2.1.1 Generating the data

Question 6

In this question need to produce a figure of k-nn map trained from the sample data. The selected k for this example figure here is $k = 3$ and the test points is 100. To do this we need to complete the following tasks in this answer to question 6:

- Draw 100 random data uniformly from $[0, 1]^2$, name it X_Hs, which is the feature of 100 random points as the training set we draw; along with 100 random data uniformly from $0, 1$, call it y_Hs, which is the label of 100 random points as the training set we draw. Those data would be inherited by answers to **question 7** and **question 8**, as the p_H distribution for following generating of train and test sets.
- Do k-NN predictions for the points on the set $[0, 1]^2$, in order to give label to each pixel on the background of the image plotted. This is done by meshgriding $[0, 1]^2$ by single pixel size of 0.001, which yields 1000000 pixels on the background to be labelled by the $k$nn-classifier with respect to the 100 training points generated in step 1 above.
- Plot the generated image, with 2 parts, firstly the background pixels to give a result of the k-NN classifier, next plot the 100 train points on the map. The result is shown below.

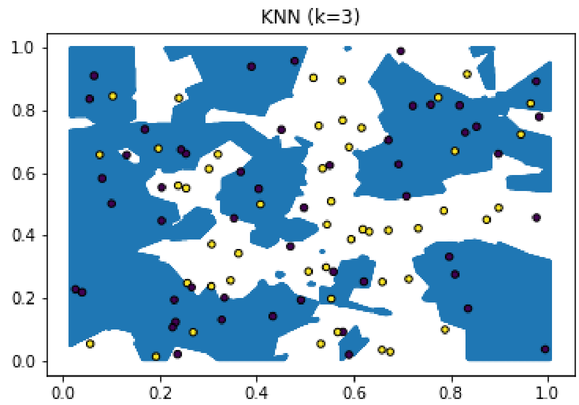

Figure 10: This figure is the visualization of generated hypothesis h_S, v with $|S| = 100$ and $k = 3$. The white area is the mapping to 0 and the blue area is the mapping to 1, the corresponding centres are yellow and dark brown.

2.1.2 Estimated generalization error of k-NN as a function of k

Question 7

In this question there are 2 parts to be completed, firstly a figure would be given with respect to protocal A in the question listed. Secondly some comments would be given for this figure. This two parts could be combined and separated as the following tasks:

- Generate the random points following the instruction given in Question 7, as 80 percents of points would be drawn and given following the generated distribution p_H in question 6. While the other 20 percents points would be given by generating uniformly from $0, 1$ as the noise in the data. This task is full-filled by writing a function called "**generating_points**" in the coding.
- Calculate the generalization error in the test sets. This is done by writing function called "**do_knn_test**", each iteration of this function would produce a data containing the error calculated by comparing results of test labels and predicted labels. This test would be done for k from 1 to 49, each k would iterate 100 times and give the mean error of each 100 runs.
- Plot the figure. A figure with generalization error vs. k would be given. Comments would be attached in the description under the figure.

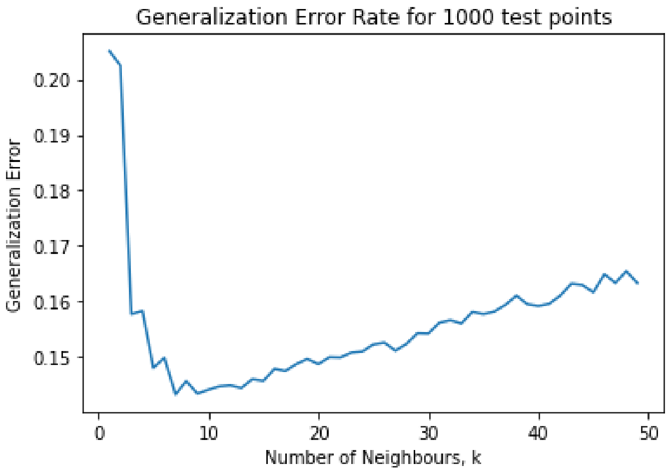

Figure 11: This figure is the visualization of generalization error vs. corresponding number of k.

- It could be seen that the value of generalization error was relatively high at $k = 1$, this is believed due to 1-NN Classifier would cause over-fitting, which caused the noise data in training set has been considered into the prediction. With k increasing to 10, the trend is sharply decreasing since the effect of over-fitting is decreasing.
- However, after k reached 10, the error would increase again with increment of k, that is because the majority of training and test points are generated from the distribution set in Question 6, which is 3-NN. So when k has exceeds too far from 3, larger k would lead to under-fitting, which means that too much details were lost in larger k-NN prediction

2.1.3 Determine the optimal k-NN as a function of the number of training points (m)

Question 8

This question could be separated as the following tasks, which is similar to question 7:

- Generate the random points following the instruction given in Question 7, as 80 percents of points would be drawn and given following the generated distribution p_H in question 6. While the other 20 percents points would be given by generating uniformly from $0, 1$ as the noise in the data. This task is full-filled by writing a function called "**generating_points**" in the coding.
- Calculate the generalization error in the test sets. This is done by writing function called "**do_knn_test**", each iteration of this function would produce a data containing the error calculated by comparing results of test labels and predicted labels. This test would be done for k from 1 to 49, each k would iterate 100 times and give the mean error of each 100 runs. Minimum error would yield the selection of optimal ks for each iteration of m training points.
- Plot the figure. A figure with generalization error vs. k would be given. Comments would be attached in the description under the figure, same as done in question 7.

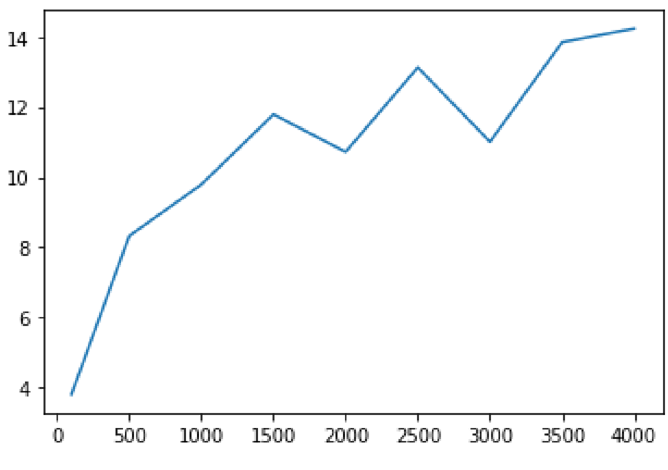

Figure 12: This figure is the visualization of optimal k vs. volume of training points. Here X-axis is the number of points in the training set. y-axis is the number of optimal k for each selection of training points.

- It could be seen that the value of optimal k is increasing with respect to the increment of numbers m of training points.
- The plotted line start from [100, 3.8] which indicates that when the training set volume is the same as H_S, v generated in question 6, the optimal k is very close to 3.
- When the number of training points became larger, the optimal training points became larger since there are more noise points to be ignored in the set, so larger k would give robustness of k-NN model against increasing noise.

2.1.4 Additional Clarifications

This is done in Question 6 by adding a frame to the plot of question 6 in order to avoid corner points. By broaden the corner.

3 Part III

3.1 Questions

Question 9

Consider the function $K_c(\mathbf{x}, \mathbf{z}) := c + \sum_{i=1}^{n} x_i z_i$ where $\mathbf{x}, \mathbf{z} \in R^n$.

a) For what values of $c \in R$ is K_c a positive definite kernel?

The suitable value of c that would make K_c a positive definite kernel is $c \geq 0$.
Proof:
To prove that K_c is a PSD kernel, need to prove the gram matrix of K_c is a PSD Matrix.
The gram matrix of K_c is:

$$\left[K_c(x_i, x_j)\right]_{m\times m}$$

where $x_i, x_j \in \mathbf{x}$.

To prove the above matrix is a PSD matrix, refer to the property of PSD matrix, could rewrite the above matrix into the sum of two PSD matrix.

Define matrix $A =$

$$\left[< x_i, x_j >\right]_{m\times m}$$

where $x_i, x_j \in \mathbf{x}$, $< x_i, x_j >$ is the inner product of them. Here matrix A is a PSD matrix obviously.
Next define a Constant matrix $C =$

$$\left[c\right]_{m\times m}$$

where each term of matrix C is constant c in the kernel function K_c. To make the gram matrix of K_c a PSD, should ensure A and C PSD st.

$$K_c = A + B$$

Hence K_c a PSD Kernel if $c \geq 0$.
Which concludes the proof.

b) Suppose we use K_c as a kernel function with linear regression(least squares). Explain how c influences the solution.

Solve:
We define

$$K\left(\boldsymbol{x}, \boldsymbol{z}\right) := \sum_{i=1}^{n} x_i z_i, where\ \boldsymbol{x}, \boldsymbol{z} \in R^n$$

$$K := \begin{bmatrix} K(\boldsymbol{x}_1, \boldsymbol{x}_1) & ... & K(\boldsymbol{x}_1, \boldsymbol{x_m}) \\ ... & \ddots & ... \\ K(\boldsymbol{x_m}, \boldsymbol{x}_1) & ... & K(\boldsymbol{x_m}, \boldsymbol{x_m}) \end{bmatrix}$$

Since,

$$K_{\boldsymbol{c}}\left(\boldsymbol{x}, \boldsymbol{z}\right) := c + \sum_{i=1}^{n} x_i z_i, where\ \boldsymbol{x}, \boldsymbol{z} \in R^n$$

Therefore,

$$K_c := \begin{bmatrix} K_c(\boldsymbol{x}_1, \boldsymbol{x}_1) & ... & K_c(\boldsymbol{x}_1, \boldsymbol{x_m}) \\ ... & \ddots & ... \\ K_c(\boldsymbol{x_m}, \boldsymbol{x}_1) & ... & K_c(\boldsymbol{x_m}, \boldsymbol{x_m}) \end{bmatrix} = C + K, where\ C = \begin{bmatrix} c & ... & c \\ ... & \ddots & ... \\ c & ... & c \end{bmatrix}$$

when we apply $K(\boldsymbol{x},\boldsymbol{z})$ into linear regression, we can calculate the weights

$$\alpha = K^{-1}y$$

Similarly, when we apply $K_c(\boldsymbol{x},\boldsymbol{z})$ into linear regression, the weights is

$$\alpha_c = {K_c}^{-1}y = (C+K)^{-1}y$$

Hence,

$$K\alpha = (C+K)\,\alpha_c$$
$$\alpha = K^{-1}(C+K)\,\alpha_c = \left(K^{-1}C + I_m\right)\alpha_c$$
$$\alpha_c = \left(K^{-1}C + I_m\right)^{-1}\alpha$$

From the formulation above, we can see how c affects regression's solution α_c. The solution α_c can be taken over by c, If c=0, C=O, $\alpha_c = \alpha$.

Question 10

Suppose we perform linear regression with a Gaussian kernel $K_\beta(\mathbf{x},\mathbf{t}) = exp(-\beta||\mathbf{x}-\mathbf{t}||^2)$ to train a classifier on a dataset $(\mathbf{x}_1, y_1)\dots(\mathbf{x}_m, y_m) \in R^n \times \{-1,1\}$. Thus obtaining a function $f : R^n \to R$ which is of the from $f(\mathbf{t}) = \sum_{i=1}^{m} \alpha_i K_\beta(\mathbf{x}_i,\mathbf{t})$. The corresponding classifier is then sign($f(\mathbf{t})$). This classifier depends on the parameter β selected for the kernel. In what scenario will chosen β enable the trained linear classifier to simulate a $1-NEAREST$ $NEIGHBOUR\ CLASSIFIER$ trained on the same dataset?

Conclusion A relativly large β should be chosen to ensure that the kernel regression classifier could simulate the effect of $1-NN$ Classifier.

Proof Sketch a proof and list the chosen of β as a function of training feature $\mathbf{x}$ and test feature $\mathbf{t}$.
Firstly, the Gaussian Kernel could be rewritten as the form of amplifying a Gaussian Distribution pdf:

$$K_\beta(\mathbf{x},\mathbf{t}) = exp(-\beta||\mathbf{x}-\mathbf{t}||^2) = (\frac{1}{\sigma(\beta)\sqrt{2\pi}}exp(\frac{-||\mathbf{x}-\mathbf{t}||^2}{2\sigma(\beta)^2})) \times C(\beta)$$

where $C(\beta) = \sigma(\beta)\sqrt{2\pi}$, and $\sigma(\beta)$ defined implicitly by the following equation:

$$\beta = \frac{1}{2\sigma(\beta)^2}$$

It is obvious that

$$\frac{1}{\sigma(\beta)\sqrt{2\pi}}exp(\frac{-||\mathbf{x}-\mathbf{t}||^2}{2\sigma(\beta)^2}))$$

forms a Gaussian distribution with mean at $\mathbf{t}$ and with variance $\sigma(\beta)^2$.
From property of Gaussian distribution, more than 99% of possibilities laid in the interval $[\mathbf{t}-3\sigma(\beta), \mathbf{t}-3\sigma(\beta)]$, using this property, we define a $\sigma(\beta)$ that is small enough that allows only one point in $\mathbf{x}$ laid in the interval $[\mathbf{t}-3\sigma(\beta), \mathbf{t}-3\sigma(\beta)]$. Which yields the following inequation:

$$[min(||\mathbf{x}_i - \mathbf{t}_i||)|first\,minimum\,element] > 3\sigma(\beta) > [min(||\mathbf{x}_i - \mathbf{t}_i||)|second\,minimum\,element]$$

where manually selected only 1 point x_i in $\mathbf{x}$ to be covered in the range:

$$[\mathbf{t}-3\sigma(\beta), \mathbf{t}-3\sigma(\beta)]$$

Now, by selecting $\sigma(\beta)$ with respect to $(\mathbf{x},\mathbf{t})$, a relationship between β and $(\mathbf{x},\mathbf{t})$ is constructed.
The affect of selecting β like above would yields only 1 points in $[\mathbf{x}]$ contribute more than 99% affect on the classifier function $f(\mathbf{t})$, which is just similar to the effect of 1-NN Classifier.
Which concludes the proof.

Question 11

note: all of the operations here observe the mod 2 operation rule. In the game of whack a mole, we have a $n \times n$ board,and moles come out from some holes in the board. We use a $n \times n$ matrix B=$[b_{ij}]$ to represent the board.

$$b_{ij} = \begin{cases} 1\ mole\ comes\ out \\ 0\ mole\ hides \end{cases} \quad ,i,j = 1...n$$

Our purpose is to make $b_{ij} = 0$ for any $i,j = 1...n$. For the question we know that every time we hit a mole with the mallet we'll cause the mole and the immediate four adjacent holes to change. So we can define the $Hit(p,q) = [H(p,q)_{ij}]$ as a $n \times n$ matrix, as below:

$$H(p,q)_{ij} = \begin{cases} 1 \quad (i,j) = (p,q)\ \ or\ \ (p-1,q)\ \ or\ \ (p+1,q)\ \ or\ \ (p,q+1)\ \ or\ \ (p,q-1) \\ 0\ \ otherwise \end{cases} \quad ,i,j = 1...n$$

We give an initial board configuration as $B^{(0)}$ and we consider $B^{(k)}$as the result of k-th hit. So every time we whack a mole in the position b_{pq}, we can get the iteration formulation below:

$$B^{(k+1)} = B^{(k)} + Hit\,(p, q)$$

If this problem exists a solution, we can find a finite sequence of $\{(p_m, q_k)\,|m = 1...n; k = 1...n\}$ to operate the $H\,(p_m, q_k)$ operation in order. So we can establish mathematical expressions, as below:

$$B_0 + \sum_{m=1}^{n} \ \sum_{k=1}^{n} \alpha_{m,k} H\,(p_m, q_k) = O \ \ , \alpha_{m,k} = 0 \ or \ 1$$

$$\sum_{m=1}^{n} \ \sum_{k=1}^{n} \alpha_{m,k} H\,(p_m, q_k) = B_0$$

Therefore, the origin problem has been reduced to a problem of solving linear equations. Now we have n^2 equations (because there is a equation relationship for every elements in the matrix). The gaussian elimination method's complexity is $O\left(m^3\right)$, where m is the rank of linear equation system. In this problem, $m = n^2$,so the complexity of this problem based on our algorithm is $O\left(n^6\right)$.

A Appendix I: Code for Part I

A.1 Plot fit function curve, give corresponding equation and calculate mse for Q1

```
import numpy as np
from numpy.linalg import inv
import matplotlib.pyplot as plt
# define mse function
def mse(label_y,predict_y):
    se_per=np.zeros_like(predict_y)
    for index in range(len(label_y)):
        se_per[index]=(label_y[index]-predict_y[index])**2
    return np.mean(se_per)
#define regression class
class regression(object):
    def __init__(self,input_x=[1,2,3,4],label_y=[3,2,0,5]):
        self.input_x=input_x
        self.label_y=label_y
    #define linear regression function,return coefficiences of basis functions,and MSE,and
     predict value

    def polynomial_basis_fun_reg(self,order):
        A=np.mat([0. for i in range(len(self.input_x)*order)]).reshape(len(self.input_x),
    order)
        for row in range(len(self.input_x)):
            for k in range(order):
                A[row,k]=(self.input_x[row])**k
        b=np.mat(self.label_y).reshape(len(self.label_y),1)
        alpha=inv((A.T)@A)@(A.T)@b#coefficiences of basis functions
        #print("regression coefficiences for order "+str(order)+' are:\n', alpha)
        alpha=np.array(alpha).reshape(len(alpha))
        #------------------caulate MSE-------------------#
        predict_y=np.array((A@alpha))
        predict_y=predict_y.reshape(len(self.label_y))
        MSE=mse(self.label_y,predict_y)
        #print("MSE for order "+str(order)+' is:', MSE)
        return alpha,MSE,predict_y
#show the equations of fit function
def create_curves_equations(order,coefficiences):
    equation=""
    for index in range(order):
        s='{}x^{}'.format(coefficiences[index,0],index)
        equation=equation+"+"+s
    return equation[1:]
order=4
create_curves_equations(order,coefficiences4)
```

Listing 1: Code for Q1

A.2 Plot mse(or Ln(mse)) for both training and test set in Q2 to explore overfitting

```
#generate sample data
def generate_traning_x(sample_num=30,round=0):
    np.random.seed(round)
    input_x = np.random.uniform(0,1,sample_num)
    return input_x
input_x=generate_traning_x()
#generate label data
def sin_square_sigma(x,sample_num=30,miu=0,sigma=0.07,round=0):
    y=np.zeros_like(x)
    np.random.seed(round)
    epsilon=np.random.normal(miu,sigma,sample_num)
    for i in range(len(x)):
        y[i]=(np.sin(2*(np.pi)*x[i]))**2 + epsilon[i]
    return y
#define a LN function to calculate ln value for every element in a list
def LN(error):
    LN_result=np.zeros_like(error)
    for i in range(len(error)):
        LN_result[i]=np.log(error[i])
    return LN_result
#(a)i  plot scatter of our generating points and the fuction curve of (sin(2*pi*x))^2
x=np.linspace(0,1,100)
plt.plot(x, (np.sin(2*np.pi*x))**2)
plt.scatter(input_x,label_y,s=25,c='r')
plt.savefig('/SL_CW/pic/Q2ai.png')
#(a)ii
#define poly_k to calculate predict result of given x via our trained model
def poly_k(alpha,x,order):
    y=0
    for k in range(order):
         y=y+alpha[k]*x**k
    return y
#Fit the data set with a polynomial basis of dimension k=2,5,10,14,18 and plot each curve
def Q2aii():
    input_x=generate_traning_x()
    label_y=sin_square_sigma(input_x)
    x=np.linspace(0,1,100)
    w=[]
    MSE=np.zeros((5,1))
    #def plot_graph(input_x=[1,2,3,4],label_y=[3,2,0,5],max_order):
    #predict_result=np.zeros((max_order,len(label_y)))#predict result for every order
    test=regression(input_x,label_y)
    alpha2,MSE[0],_=test.polynomial_basis_fun_reg(2)
    alpha5,MSE[1],_=test.polynomial_basis_fun_reg(5)
    alpha10,MSE[2],_=test.polynomial_basis_fun_reg(10)
    alpha14,MSE[3],_=test.polynomial_basis_fun_reg(14)
    alpha18,MSE[4],_=test.polynomial_basis_fun_reg(18)

    #plot=np.poly1d(predict_result[0,:])
    #
    y2=np.zeros_like(x)
    y5=np.zeros_like(x)
    y10=np.zeros_like(x)
    y14=np.zeros_like(x)
    y18=np.zeros_like(x)
    for i in range(len(x)):
        y2[i]=poly_k(alpha2,x[i],2)
        y5[i]=poly_k(alpha5,x[i],5)
        y10[i]=poly_k(alpha10,x[i],10)
        y14[i]=poly_k(alpha14,x[i],14)
        y18[i]=poly_k(alpha18,x[i],18)
    l2,=plt.plot(x, y2,label="k=2")
    l5,=plt.plot(x, y5,label="k=5")
    l10,=plt.plot(x, y10,label="k=10")
    l14,=plt.plot(x, y14,label="k=14")
    l18,=plt.plot(x, y18,label="k=18")
    plt.scatter(input_x,label_y,s=25,c='r')
    # set up axis region
    plt.legend(handles=[l2,l5,l10,l14,l18],labels=["k=2","k=5","k=10","k=14","k=18"])
    plt.xlim((0, 1))
    plt.ylim((0, 1))
```

```
    plt.savefig('/SL_CW/pic/Q2aii.png')
    plt.show()
    #print(MSE)
Q2aii()
#(b)
#define class Error_analysis to analyse mse(or ln(mse)) both in training and test sets
class Error_analysis(object):
    def __init__(self,input_x,label_y,max_order=18):
        self.input_x=input_x
        self.label_y=label_y
        self.max_order=max_order
        self.poly_dimension=np.arange(self.max_order+1)[1:]
    #return training error
    def calculate_mse(self):
        #w=[]
        training_error=np.zeros(self.max_order)
        test_Q2=regression(self.input_x,self.label_y)
        for k in range(self.max_order):
            alpha,training_error[k],_=test_Q2.polynomial_basis_fun_reg(self.poly_dimension
    [k])
            w.append(alpha)
        return training_error
    #return ln(training error)
    def calculate_Ln_mse(self):
        training_error=np.zeros(self.max_order)
        test_Q2=regression(self.input_x,self.label_y)
        for k in range(self.max_order):
            _,training_error[k],_=test_Q2.polynomial_basis_fun_reg(self.poly_dimension[k])
        LN_training_error=LN(training_error)
        return  LN_training_error
    #return Ln(test error) and test error
    def ln_mse_in_testset(self,alpha,test_x,test_y,order):
        A=np.mat([0. for i in range(len(test_x)*order)]).reshape(len(test_x),order)
        for row in range(len(test_x)):
            for k in range(order):
                A[row,k]=(test_x[row])**k
        alpha=alpha.reshape(order,1)
        predict_y=np.array(A@alpha)
        predict_y=predict_y.reshape(len(test_y))
        MSE=mse(test_y,predict_y)
        ln_mse=np.log(MSE)
        return ln_mse,MSE
#plot LN mse of training error against polynomial order k
def Q2b():
    w=[]
    input_x=generate_traning_x()
    label_y=sin_square_sigma(input_x)
    order_candidate=np.arange(19)[1:]
    reg=regression(input_x,label_y)
    for i in range(len(order_candidate)):
        alpha,_,_=reg.polynomial_basis_fun_reg(order_candidate[i])
        w.append(alpha)
    Q2_b=Error_analysis(input_x,label_y)
    LN_training_error_Q2_b=Q2_b.calculate_Ln_mse()
    print(LN_training_error_Q2_b)
    plt.xlim((0, 18))
    plt.plot(order_candidate,LN_training_error_Q2_b)
    plt.xlabel("k")
    plt.ylabel("LN mse of training error")
    plt.savefig('/SL_CW/pic/Q2b.png')
    return w
w=Q2b()
#(c)plot LN mse of test error against polynomial order k
def Q2c(w):
    input_x_Q2_c=generate_traning_x(sample_num=1000)
    label_y_Q2_c=sin_square_sigma(input_x_Q2_c,sample_num=1000,miu=0,sigma=0.07)
    Q2_c=Error_analysis(input_x_Q2_c,label_y_Q2_c)
    LN_test_MSE=np.zeros((len(w),1))
    order_candidate=np.arange(19)[1:]
    for i in range(len(w)):
        LN_test_MSE[i],_=Q2_c.ln_mse_in_testset(w[i],input_x_Q2_c,label_y_Q2_c,
    order_candidate[i])

    #Q2_c=Error_analysis(input_x_Q2_c,label_y_Q2_c)
#LN_training_error_Q2_c=Q2_c.calculate_Ln_mse()
```

```
    plt.plot(order_candidate,LN_test_MSE)
    plt.xlim((0, 18))
    #plt.plot(order_candidate,LN_training_error_Q2_b)
    plt.xlabel("k")
    plt.ylabel("LN mse of test error")
    plt.savefig('/SL_CW/pic/Q2c.png')
Q2c(w)
#(d)
#return feature maps A
def fm(input_x,label_y,order):
    m=len(input_x)
    A=np.zeros((m,order))#[U+FFFD][U+FFFD][U+FFFD][U+FFFD][U+FFFD]
    for row in range(m):
        for col in range(order):
            A[row,col]=input_x[row]**col
    return A
#return regression coefficients
def poly_reg(A,label_y):
    label_y=label_y.reshape(len(label_y),1)
    w=np.linalg.inv(A.T@A)@(A.T)@label_y
    w=np.array(w)
    w=w.reshape(len(w))
    return w
#predict  test_y for all test_x
def predict(A,w,order):
    test_y=A@w
    test_y=np.array(test_y)
    test_y=test_y.reshape(len(test_y))
    return test_y
#plot LN(avgmse) for training and test set over 100 runs
def Q2d():
    r=100
    te=np.zeros((r,len(order_set)))
    tse=np.zeros((r,len(order_set)))
    for i in range(r):
        b=i
        if i==6:
            b=107
        elif i==21:
            b=102
        elif i==66:
            b=108
        elif i==95:
            b=105
        elif i==98:
            b=104

        train_x=generate_traning_x(round=b)
        train_y=sin_square_sigma(train_x,round=b)

        test_x=generate_traning_x(sample_num=1000,round=b)
        test_y=sin_square_sigma(test_x,sample_num=1000,round=b)

        w_all_orders=[]
        #calculate training error
        for k in range(len(order_set)):
            A=fm(train_x,train_y,order_set[k])
            w=poly_reg(A,train_y)
            w_all_orders.append(w)#reserve weights for test_set
            predict_y=predict(A,w,order_set[k])
            #print(predict_y)
            te[i,k]=mse(predict_y,train_y)
        for k in range(len(order_set)):
            A_test=fm(test_x,test_y,order_set[k])
            predict_tsy=predict(A_test,w_all_orders[k],order_set[k])
            #print(predict_tsy)
            tse[i,k]=mse(predict_tsy,test_y)

    #print(te)
    #print(tse)
    LN_avg_te=LN(np.mean(te,axis=0))
    LN_avg_tse=LN(np.mean(tse,axis=0))
    return LN_avg_te,LN_avg_tse
LN_avg_te,LN_avg_tse=Q2d()
#Do plot
```

```python
def Q2d_plot():
    order_candidate=np.arange(19)[1:]
    line1,=plt.plot(order_candidate,LN_avg_te,label='train')
    line2,=plt.plot(order_candidate,LN_avg_tse,label='test')
    plt.legend(handles=[line1,line2],labels=["train",'test'])
    plt.xlim((0, 18))
    plt.xlabel('order')
    plt.ylabel('Ln average mse')
    plt.savefig('/SL_CW/pic/Q2d.png')
    plt.show()
Q2d_plot()
```

Listing 2: Q2 code

A.3 Plot mse(or ln(mse)) for both training and test set based on a different basis function for Q3

```python
#define feature map based on sin(k*pi*x) basis function
def fm_sin(input_x,label_y,order):
    m=len(input_x)
    A=np.zeros((m,order))#
    for row in range(m):
        for col in range(order):
            A[row,col]=np.sin((col+1)*np.pi*(input_x[row]))
    return A
#(b)(c)
#plot ln average mse over only one round for training and test set
def Q3(r=1):
    te=np.zeros((r,len(order_set)))
    tse=np.zeros((r,len(order_set)))
    for i in range(r):
        b=i
        if i==6:
            b=107
        elif i==21:
            b=102
        elif i==66:
            b=108
        elif i==95:
            b=105
        elif i==98:
            b=104
        train_x=generate_traning_x(round=b)
        train_y=sin_square_sigma(train_x,round=b)
        test_x=generate_traning_x(sample_num=1000,round=b)
        test_y=sin_square_sigma(test_x,sample_num=1000,round=b)
        w_all_orders=[]
        #calculate training error
        for k in range(len(order_set)):
            A=fm_sin(train_x,train_y,order_set[k])
            w=poly_reg(A,train_y)
            w_all_orders.append(w)#reserve weights for test_set
            predict_y=predict(A,w,order_set[k])
            #print(predict_y)
            te[i,k]=mse(predict_y,train_y)
        for k in range(len(order_set)):
            A_test=fm_sin(test_x,test_y,order_set[k])
            predict_tsy=predict(A_test,w_all_orders[k],order_set[k])
            #print(predict_tsy)
            tse[i,k]=mse(predict_tsy,test_y)
    LN_avg_te=LN(np.mean(te,axis=0))
    LN_avg_tse=LN(np.mean(tse,axis=0))
    return LN_avg_te,LN_avg_tse
LN_avg_te,LN_avg_tse=Q3()
 #Do plot
def Q3_plot():
    order_candidate=np.arange(19)[1:]
    line1,=plt.plot(order_candidate,LN_avg_te,label='train')
    line2,=plt.plot(order_candidate,LN_avg_tse,label='test')
    plt.legend(handles=[line1,line2],labels=["train",'test'])
    plt.xlim((0, 18))
    plt.xlabel('order')
```

```
    plt.ylabel('Ln average mse')
    plt.savefig('/SL_CW/pic/Q3bc.png')
    plt.show()
Q3_plot()
#(d)plot ln average mse over 100 round for training and test set
def Q3(r=100):
    te=np.zeros((r,len(order_set)))
    tse=np.zeros((r,len(order_set)))
    for i in range(r):
        b=i
        if i==6:
            b=107
        elif i==21:
            b=102
        elif i==66:
            b=108
        elif i==95:
            b=105
        elif i==98:
            b=104

        train_x=generate_traning_x(round=b)
        train_y=sin_square_sigma(train_x,round=b)

        test_x=generate_traning_x(sample_num=1000,round=b)
        test_y=sin_square_sigma(test_x,sample_num=1000,round=b)

        w_all_orders=[]
        #calculate training error
        for k in range(len(order_set)):
            A=fm_sin(train_x,train_y,order_set[k])
            w=poly_reg(A,train_y)
            w_all_orders.append(w)#reserve weights for test_set
            predict_y=predict(A,w,order_set[k])
            #print(predict_y)
            te[i,k]=mse(predict_y,train_y)
        for k in range(len(order_set)):
            A_test=fm_sin(test_x,test_y,order_set[k])
            predict_tsy=predict(A_test,w_all_orders[k],order_set[k])
            #print(predict_tsy)
            tse[i,k]=mse(predict_tsy,test_y)
    LN_avg_te=LN(np.mean(te,axis=0))
    LN_avg_tse=LN(np.mean(tse,axis=0))
    return LN_avg_te,LN_avg_tse
LN_avg_te,LN_avg_tse=Q3()
#Do plot
def Q3_plot():
    order_candidate=np.arange(19)[1:]
    line1,=plt.plot(order_candidate,LN_avg_te,label='train')
    line2,=plt.plot(order_candidate,LN_avg_tse,label='test')
    plt.legend(handles=[line1,line2],labels=["train",'test'])
    plt.xlim((0, 18))
    plt.xlabel('order')
    plt.ylabel('Ln average mse')
    plt.savefig('/SL_CW/pic/Q3d.png')
    plt.show()
Q3_plot()
```

Listing 3: Q3 code

A.4 Baseline versus full linear regression

```
#naive regression,linear_reg_single_attributes,linear_reg_all_attributes
class naive_regression(object):
  def __init__(self,input_x,label_y):
    self.input_x=input_x
    self.label_y=label_y
  def fm(self):
    m=len(self.input_x)
    A=np.ones((m,1))
    return A
  def naive_reg(self,A):
    w=np.linalg.inv(A.T@A)@(A.T)@(self.label_y)
    w=np.array(w)
```

```
    w=w.reshape(len(w))
    return w
  def predict(self,A,w):
    test_y=A@w
    test_y=np.array(test_y)
    test_y=test_y.reshape(len(test_y))
    return test_y
class linear_reg_single_attributes(object):
  def __init__(self,input_x,label_y,rank):
    self.rank=rank
    self.x=input_x[:,self.rank]
    self.label_y=label_y
  def fm(self):
    m=len(self.x)
    A=np.zeros((m,2))
    for row in range(m):
      for k in range(2):
        A[row,k]=(self.x[row])**k
    return A
  def single_reg(self,A):
    w=np.linalg.inv(A.T@A)@(A.T)@(self.label_y)
    w=np.array(w)
    w=w.reshape(len(w))
    return w
  def predict(self,A,w):
    test_y=A@w
    test_y=np.array(test_y)
    test_y=test_y.reshape(len(test_y))
    return test_y
class linear_reg_all_attributes(object):
  def __init__(self,input_x,label_y):
    self.input_x=input_x
    self.label_y=label_y
  def fm(self):
    m=len(self.input_x)
    bias=np.ones((len(self.input_x),1))
    A=np.hstack((bias,self.input_x))
    return A
  def multi_reg(self,A):
    w=np.linalg.inv(A.T@A)@(A.T)@(self.label_y)
    w=np.array(w)
    w=w.reshape(len(w))
    return w
  def predict(self,A,w):
    test_y=A@w
    test_y=np.array(test_y)
    test_y=test_y.reshape(len(test_y))
    return test_y
#perform comparision
def Q4():
  totalRuns=20
  gamma_candidates=np.array([2**(i) for i in range(-40,-25)])
  sigma_candidates=np.array([2**7,2**(7.5),2**(8),2**(8.5),2**(9),2**(9.5),2**(10)
    ,2**(10.5),2**(11),2**(11.5),2**(12),2**(12.5),2**(13)])
  dataset=pd.read_csv(r'/content/Boston-filtered.csv').to_numpy()
  m,n=dataset.shape
  num_trainset=int(m*(2/3))
  #for single reg
  train_error_SR=np.zeros((totalRuns,12))
  test_error_SR=np.zeros((totalRuns,12))
  #for naive reg
  train_error_NR=np.zeros((totalRuns,1))
  test_error_NR=np.zeros((totalRuns,1))
  train_error_ML=np.zeros((totalRuns,1))
  test_error_ML=np.zeros((totalRuns,1))
  #-------------------------------------------------#
  for i in range(totalRuns):
      dataset2=np.random.permutation(dataset)
      X=dataset2[:,:-1]
      Y=dataset2[:,-1:]
      train_X=X[0:num_trainset]
      train_Y=Y[0:num_trainset]
      test_X=X[num_trainset:]
      test_Y=Y[num_trainset:]
      #linear_reg with single attribute
```

```python
      for rank in range(12):
        Q4_SR=linear_reg_single_attributes(train_X,train_Y,rank)
        A_train=Q4_SR.fm()
        w=Q4_SR.single_reg(A_train)
        predict_y=Q4_SR.predict(A_train,w)
        train_error_SR[i,rank]=mse(predict_y,train_Y)
        #test
        Q4_SR_test=linear_reg_single_attributes(test_X,test_Y,rank)
        A_test=Q4_SR_test.fm()
        predict_tsy=Q4_SR.predict(A_test,w)
        test_error_SR[i,rank]=mse(predict_tsy,test_Y)
      #naive reg
      Q4_NR=naive_regression(train_X,train_Y)
      A_train2=Q4_NR.fm()
      w2=Q4_NR.naive_reg(A_train2)
      predict_y2=Q4_NR.predict(A_train2,w2)
      train_error_NR[i]=mse(predict_y2,train_Y)
      #test
      Q4_NR_test=naive_regression(test_X,test_Y)
      A_test2=Q4_NR_test.fm()
      predict_tsy2=Q4_NR.predict(A_test2,w2)
      test_error_NR[i]=mse(predict_tsy2,test_Y)
      #multi-linear-reg
      Q4_ML=linear_reg_all_attributes(train_X,train_Y)
      A_train3=Q4_ML.fm()
      w3=Q4_ML.multi_reg(A_train3)
      predict_y3=Q4_ML.predict(A_train3,w3)
      train_error_ML[i]=mse(predict_y3,train_Y)
      #test
      Q4_ML_test=linear_reg_all_attributes(test_X,test_Y)
      A_test3=Q4_ML_test.fm()
      predict_tsy3=Q4_ML.predict(A_test3,w3)
      test_error_ML[i]=mse(predict_tsy3,test_Y)

  #linear_reg with single attribute:mean and std of results of 20 rounds
  avg_trainError_SR=np.mean(train_error_SR,axis=0)
  std_trainError_SR=np.std(train_error_SR,axis=0)
  avg_testError_SR=np.mean(test_error_SR,axis=0)
  std_testError_SR=np.std(test_error_SR,axis=0)
  #Naive Regression:mean and std of results of 20 rounds
  avg_trainError_NR=np.mean(train_error_NR)
  std_trainError_NR=np.std(train_error_NR)
  avg_testError_NR=np.mean(test_error_NR)
  std_testError_NR=np.std(test_error_NR)
  #multi-linear-reg:mean and std of results of 20 rounds
  avg_trainError_ML=np.mean(train_error_ML)
  std_trainError_ML=np.std(train_error_ML)
  avg_testError_ML=np.mean(test_error_ML)
  std_testError_ML=np.std(test_error_ML)
  print('avg_trainError_NR:',avg_trainError_NR)
  print('std_trainError_NR:',std_trainError_NR)
  print('avg_testError_NR:',avg_testError_NR)
  print('std_testError_NR:',std_testError_NR)
  print('+-----------------------------------+')
  print('avg_trainError_SR:',avg_trainError_SR)
  print('std_trainError_SR:',std_trainError_SR)
  print('avg_testError_SR:',avg_testError_SR)
  print('std_testError_SR:',std_testError_SR)
  print('+-----------------------------------+')
  print('avg_trainError_ML:',avg_trainError_ML)
  print('std_trainError_ML:',std_trainError_ML)
  print('avg_testError_ML:',avg_testError_ML)
  print('std_testError_ML:',std_testError_ML)
  print('+-----------------------------------+')
Q4()
```

Listing 4: Q4 code

A.5 Perform KRR ,find the best pair of gamma and sigma for KRR, compare the performance with other methods.

```python
#derived data set
```

```
dataset=pd.read_csv(r'/SL_CW/Boston-filtered.csv').to_numpy()
#define guassian kernel
def Gaussian_kernel(xi,xj,sigma):
  n=np.linalg.norm((xi-xj),ord=2)
  return np.exp(n**2/(-2*sigma**2))
#define guassian kernel feature map
def fm_GK(input_x,sigma):
    m=len(input_x)
    A=np.zeros((m,m))
    for row in range(m):
        for col in range(m):
            A[row,col]=Gaussian_kernel(input_x[row],input_x[col],sigma)
    return A
#define predict value
def predict_GK(train_x,test_x,w,sigma):
    m=len(train_x)
    n=len(test_x)
    A_test=np.zeros((m,n))
    for row in range(m):
        for col in range(n):
            A_test[row,col]=Gaussian_kernel(train_x[row],test_x[col],sigma)
    test_y=(A_test.T)@w
    test_y=np.array(test_y)
    test_y=test_y.reshape(len(test_y))
    return test_y
#define five fold cross validation for training hyper parameters
def five_fold_cv(train_X,train_Y,gamma,sigma):
    #sigma=2**7
    val_set_size=int(0.2*len(train_X))
    test_mse_each_fold=np.zeros((5,1))
    for i in range(5):
        if i==4:
            V_X=train_X[i*val_set_size:]
            V_Y=train_Y[i*val_set_size:]
            T_X=train_X[:i*val_set_size]
            T_Y=train_Y[:i*val_set_size]
        else:
            V_X=train_X[i*val_set_size:(i+1)*val_set_size]
            V_Y=train_Y[i*val_set_size:(i+1)*val_set_size]
            T_X=np.vstack((train_X[:i*val_set_size],train_X[(i+1)*val_set_size:]))
            T_Y=np.vstack((train_Y[:i*val_set_size],train_Y[(i+1)*val_set_size:]))
        #Q5a=kernel_methods(T_X,T_Y)
        A=fm_GK(T_X,sigma)
        w=kernel_reg(A,T_Y,gamma)
        predict_y=predict_GK(T_X,V_X,w,sigma)
        test_mse_each_fold[i]=mse(predict_y,V_Y)
    avgtest_mse=np.mean(test_mse_each_fold)
    return avgtest_mse
#(a)
#define a function to calculate average mse for every pair of hyperparameters and find the
     best pair
def avgmse_with_parameters(train_X,train_Y,gamma_candidates,sigma_candidates):
    num_g=len(gamma_candidates)
    num_s=len(sigma_candidates)
    mse_for_each_par=np.zeros((num_g,num_s))
    for g in range(num_g):
        for s in range(num_s):
            mse_for_each_par[g,s]=five_fold_cv(train_X,train_Y,gamma_candidates[g],
    sigma_candidates[s])
    best_par=np.where(mse_for_each_par==np.min(mse_for_each_par))
    best_gamma_index=best_par[0][0]
    best_sigma_index=best_par[1][0]
    return  mse_for_each_par,best_gamma_index,best_sigma_index
gamma_candidates=np.array([2**(i) for i in range(-40,-25)])
sigma_candidates=np.array([2**7,2**(7.5),2**(8),2**(8.5),2**(9),2**(9.5),2**(10),2**(10.5)
    ,2**(11),2**(11.5),2**(12),2**(12.5),2**(13)])
mse_for_each_par,best_gamma_index,best_sigma_index=avgmse_with_parameters(train_X,train_Y,
    gamma_candidates,sigma_candidates)
print(mse_for_each_par)
print(best_gamma_index)
print(best_sigma_index)
#(b)plot the "cross-validation error" as a function of gamma and sigma
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
```

```
fig=plt.figure(figsize=(10.0,9.0))
#fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
xs=sigma_candidates
ys=gamma_candidates
X,Y=np.meshgrid(xs,ys)
zs=mse_for_each_par
ax.plot_surface(X,Y,zs)
ax.set_xlabel('sigma')
ax.set_ylabel('gamma')
ax.set_zlabel('mse')
plt.savefig('D:\SL_CW\pic\Q5b.png')
plt.show()
#(c)
#calculte the training and test mse based on the best pair of gamma and sigma
def Q4c():
    gamma=gamma_candidates[best_gamma_index]
    sigma=sigma_candidates[best_sigma_index]
    np.random.seed(0)
    dataset3 = np.random.permutation(dataset)
    X=dataset3[:,:-1]
    Y=dataset3[:,-1:]
    train_X=X[0:num_trainset]
    train_Y=Y[0:num_trainset]
    test_X=X[num_trainset:]
    test_Y=Y[num_trainset:]
    A=fm_GK(train_X,sigma)
    w=kernel_reg(A,train_Y,gamma)
    predict_y=predict_GK(train_X,train_X,w,sigma)
    train_mse=mse(predict_y,train_Y)
    predict_tsy=predict_GK(train_X,test_X,w,sigma)
    test_mse=mse(predict_tsy,test_Y)
    print(train_mse)
    print(test_mse)
Q4c()
#(d) performance comparing among baseline predicting, sinlge/multi linear regression and
    kernel method over 20 rounds
def Q5d():
  totalRuns=20
  gamma_candidates=np.array([2**(i) for i in range(-40,-25)])
  sigma_candidates=np.array([2**7,2**(7.5),2**(8),2**(8.5),2**(9),2**(9.5),2**(10)
    ,2**(10.5),2**(11),2**(11.5),2**(12),2**(12.5),2**(13)])
  dataset=pd.read_csv(r'/content/Boston-filtered.csv').to_numpy()
  m,n=dataset.shape
  num_trainset=int(m*(2/3))
  #for single reg
  train_error_SR=np.zeros((totalRuns,12))
  test_error_SR=np.zeros((totalRuns,12))
  #for naive reg
  train_error_NR=np.zeros((totalRuns,1))
  test_error_NR=np.zeros((totalRuns,1))
  train_error_ML=np.zeros((totalRuns,1))
  test_error_ML=np.zeros((totalRuns,1))
  #-----------------------------------------------#
  #for kernel_method
  Store_best_par=np.zeros((totalRuns,2))
  train_error_KM=np.zeros((totalRuns,1))
  test_error_KM=np.zeros((totalRuns,1))
  for i in range(totalRuns):
      dataset2=np.random.permutation(dataset)
      X=dataset2[:,:-1]
      Y=dataset2[:,-1:]
      train_X=X[0:num_trainset]
      train_Y=Y[0:num_trainset]
      test_X=X[num_trainset:]
      test_Y=Y[num_trainset:]
      #linear_reg with single attribute
      for rank in range(12):
        Q4_SR=linear_reg_single_attributes(train_X,train_Y,rank)
        A_train=Q4_SR.fm()
        w=Q4_SR.single_reg(A_train)
        predict_y=Q4_SR.predict(A_train,w)
        train_error_SR[i,rank]=mse(predict_y,train_Y)
        #test
        Q4_SR_test=linear_reg_single_attributes(test_X,test_Y,rank)
```

```
        A_test=Q4_SR_test.fm()
        predict_tsy=Q4_SR.predict(A_test,w)
        test_error_SR[i,rank]=mse(predict_tsy,test_Y)
      #naive reg
      Q4_NR=naive_regression(train_X,train_Y)
      A_train2=Q4_NR.fm()
      w2=Q4_NR.naive_reg(A_train2)
      predict_y2=Q4_NR.predict(A_train2,w2)
      train_error_NR[i]=mse(predict_y2,train_Y)
      #test
      Q4_NR_test=naive_regression(test_X,test_Y)
      A_test2=Q4_NR_test.fm()
      predict_tsy2=Q4_NR.predict(A_test2,w2)
      test_error_NR[i]=mse(predict_tsy2,test_Y)
      #multi-linear-reg
      Q4_ML=linear_reg_all_attributes(train_X,train_Y)
      A_train3=Q4_ML.fm()
      w3=Q4_ML.multi_reg(A_train3)
      predict_y3=Q4_ML.predict(A_train3,w3)
      train_error_ML[i]=mse(predict_y3,train_Y)
      #test
      Q4_ML_test=linear_reg_all_attributes(test_X,test_Y)
      A_test3=Q4_ML_test.fm()
      predict_tsy3=Q4_ML.predict(A_test3,w3)
      test_error_ML[i]=mse(predict_tsy3,test_Y)
      #kernel method
      Store_best_par[i,0],Store_best_par[i,1]=find_best_par(train_X,train_Y,
    gamma_candidates,sigma_candidates)
      ga,si=Store_best_par[i,0],Store_best_par[i,1]
      A_train4=fm_GK(train_X,si)
      w4=kernel_reg(A_train4,train_Y,ga)
      predict_y4=predict_GK(train_X,train_X,w4,si)
      train_error_KM[i]=mse(predict_y4,train_Y)
      #test
      predict_tsy4=predict_GK(train_X,test_X,w4,si)
      test_error_KM[i]=mse(predict_tsy4,test_Y)
  #linear_reg with single attribute:mean and std of results of 20 rounds
  avg_trainError_SR=np.mean(train_error_SR,axis=0)
  std_trainError_SR=np.std(train_error_SR,axis=0)
  avg_testError_SR=np.mean(test_error_SR,axis=0)
  std_testError_SR=np.std(test_error_SR,axis=0)
  #Naive Regression:mean and std of results of 20 rounds
  avg_trainError_NR=np.mean(train_error_NR)
  std_trainError_NR=np.std(train_error_NR)
  avg_testError_NR=np.mean(test_error_NR)
  std_testError_NR=np.std(test_error_NR)
  #multi-linear-reg:mean and std of results of 20 rounds
  avg_trainError_ML=np.mean(train_error_ML)
  std_trainError_ML=np.std(train_error_ML)
  avg_testError_ML=np.mean(test_error_ML)
  std_testError_ML=np.std(test_error_ML)
  #KM:mean and std of results of 20 rounds
  avg_trainError_KM=np.mean(train_error_KM)
  std_trainError_KM=np.std(train_error_KM)
  avg_testError_KM=np.mean(test_error_KM)
  std_testError_KM=np.std(test_error_KM)
  print('avg_trainError_NR:',avg_trainError_NR)
  print('std_trainError_NR:',std_trainError_NR)
  print('avg_testError_NR:',avg_testError_NR)
  print('std_testError_NR:',std_testError_NR)
  print('+----------------------------------+')
  print('avg_trainError_SR:',avg_trainError_SR)
  print('std_trainError_SR:',std_trainError_SR)
  print('avg_testError_SR:',avg_testError_SR)
  print('std_testError_SR:',std_testError_SR)
  print('+----------------------------------+')
  print('avg_trainError_ML:',avg_trainError_ML)
  print('std_trainError_ML:',std_trainError_ML)
  print('avg_testError_ML:',avg_testError_ML)
  print('std_testError_ML:',std_testError_ML)
  print('+----------------------------------+')
  print('avg_trainError_KM:',avg_trainError_KM)
  print('std_trainError_KM:',std_trainError_KM)
  print('avg_testError_KM:',avg_testError_KM)
  print('std_testError_KM:',std_testError_KM)
```

```
  print('best pairs of parameters:',Store_best_par)
Q5d()
```

Listing 5: Q5 code

B Appendix II: Code for Part II

B.1 K-NN Classifier and generating points

```
import numpy as np
from numba import njit,jit
import numba
from matplotlib import pyplot as plt

def Draw_Hs():
  '''Draw a distribution for the following process firstly'''
  n_points = 100

  X_Hs = np.random.rand(n_points,2)
  y_Hs = np.random.choice([0,1],n_points)

  return X_Hs, y_Hs

@njit
def euc_dis(sample1, sample2):
    '''
    Euclidean distance between two sample points
    sample1: A test sample. 2-tuple
    sample2: A test sample. 2-tuple
    '''
    distance = np.sqrt(np.sum((sample1 - sample2)**2))
    return distance
@njit
def get_distance(X, testInstance):
    """
    Use numba to accelerate the process of getting distances.
    """
    distances = [euc_dis(x, testInstance) for x in X]
    return distances

def knn_classify(X, y, testInstance, k):
    '''
    Given a test data point testInstance, predict its label from knn classifier.
    X: Data feature
    y: Data label
    testInstance: test sample
    k: Number of neighbors chosen for one vote center.
    '''
    distances = get_distance(X, testInstance)
    kneighbors = np.argsort(distances)[:k]
    count = np.bincount(y[kneighbors])
    predict_label = np.argmax(count)
    return predict_label

def generate_points(X_Hs, y_Hs, sample_size, noise_size, k):
  '''

  '''
  X_Hs_sample = np.random.rand(sample_size, 2)
  y_Hs_sample = [knn_classify(X_Hs, y_Hs, data, k) for data in X_Hs_sample]

  X_noise = np.random.rand(noise_size,2)
  y_noise = np.random.choice([0,1],noise_size)

  X = np.r_[X_Hs_sample, X_noise]
  y = np.r_[y_Hs_sample, y_noise]

  return X, y
```

Listing 6: K-nn Classifier

B.2 Do k-nn labelling and plot question 6 figure

```
X_Hs, y_Hs = Draw_Hs()
#Draw the distribution of H_S,v which would be used in q6, 7 and 8.
def question6(X_Hs, y_Hs):
    # Uniformly draw some points as the training set.
    n_points = 100

    X = X_Hs
    y = y_Hs

    # Choosing K for the classifier
    k=3

    # Visualization
    x_min, x_max = X[:, 0].min() - 0.01, X[:, 0].max() + 0.01 #Avoid corner case
    y_min, y_max = X[:, 1].min() - 0.01, X[:, 1].max() + 0.01 #Avoid corner case
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.001),
                         np.arange(y_min, y_max, 0.001))

    Z = [knn_classify(X, y, data, k) for data in np.c_[xx.ravel(), yy.ravel()]]

    plt.scatter(xx,yy,Z)

    plt.scatter(X[:, 0], X[:, 1], c=-y,
                                  s=20, edgecolor='k')
    plt.title('KNN (k=%d)'%k)

    plt.show()

question6(X_Hs, y_Hs)
```

Listing 7: K-nn test

B.3 Do k-nn test and plot question 7 figure

```
def do_knn_test(X_hs, y_Hs, train_points, test_points,k):
    # Setting volume of train and test sets
    # Uniformly drawing the train set and test set
    train_sample_size = int(0.8 * train_points)
    train_noise_size = int(0.2 * train_points)

    X_train, y_train = generate_points(X_Hs, y_Hs, train_sample_size, train_noise_size, 3)


    test_sample_size = int(0.8 * test_points)
    test_noise_size = int(0.2 * test_points)

    X_test, y_test = generate_points(X_Hs, y_Hs, test_sample_size, test_noise_size, 3)

    # Predictions
    predictions = [knn_classify(X_train, y_train, data, k) for data in X_test]
    # Check the accuracy of predictions.
    errornum = np.count_nonzero((predictions==y_test)==False)
    return errornum/y_test.shape[0]

k_upbd = 50 #Set the upper bound for k, actual value of k would reach k-1
neighbours=np.arange(k_upbd)
#Initiate generalized error
generalized_error=np.zeros(len(neighbours))
for k in neighbours[1:k_upbd]:
    #Initiate error
    error = np.zeros(100)
    for i in np.arange(100):
        #Do 100 runs for each k in neighbours list.
        error[i] = do_knn_test(X_Hs, y_Hs, 4000, 1000, k)
    generalized_error[k] = np.mean(error)
    #plot(generalized_error vs. number of k)
plt.xlabel('Number of Neighbours, k')
plt.ylabel('Generalization Error')
plt.title('Generalization Error Rate for 1000 test points')
```

```
plt.plot(neighbours[1:k_upbd],generalized_error[1:k_upbd])
```
Listing 8: Calculating generalization error

B.4 Do k-nn test and plot question 8 figure

```
import numpy as np
from matplotlib import pyplot as plt
def question8(k_upbd):
    #Set the upper bound for k, actual value of k would reach k_upbd-1
    arrayaa = [100]
    arraybb = np.arange(500, 4500, 500)
    train_points = np.r_[arrayaa, arraybb] #array([ 100,  500, 1000, 1500, 2000, 2500,
    3000, 3500, 4000])
    neighbours=np.arange(k_upbd)
    moptimal_k = np.zeros(np.size(train_points))
    j = 0
    for m in train_points:
        ioptimal_k = np.zeros(100)
        for i in np.arange(100):
            # Initiate generalized error
            generalized_error=np.zeros(len(neighbours)) # array[0..49]
            generalized_error[0]=10000 # Avoid k=zero's error being the min value
            for k in neighbours[1:k_upbd]:
                generalized_error[k] = do_knn_test(X_Hs, y_Hs, m, 1000, k)
            ioptimal_k[i] = np.where(generalized_error == np.min(generalized_error))[0][0]
     # First optimal k for this one-time run.
        moptimal_k[j] = np.mean(ioptimal_k)
        j += 1
    print(moptimal_k)

    plt.plot(train_points, moptimal_k)
    return moptimal_k

k_optimals49=question8(50)
```
Listing 9: Finding optimal k