Supervised Learning (COMP0078) Coursework 1

Yuan Li (18057497)
Yuan Gao (18064382)

10/11/2018

1 PART I

1.1 Linear Regression

1. For each of the polynomial bases of dimension $k = 1, 2, 3, 4$ fit the data set of Figure 1 {(1,3),(2,2),(3,0),(4,5)}.

   (a) Produce a plot similar to Figure 1, superimposing the four different curves corresponding to each fit over the four data points.
   Solution: Figure 1 is the four different curves corresponding to fit from dimension 1 to 4 over the four data points.
   Code used to implement this question can be seen in appendix. Function name: part1_1_1_a().

   (b) Give the equations corresponding to the curves fitted for $k = 1, 2, 3$. The equation corresponding to $k = 4$ is $-5 + 15.17x - 8.5x^2 + 1.33x^3$.
   Solution:

   - when $k = 1$, $y = 2.5$,
   - when $k = 2$, $y = 1.5 + 0.4x$
   - when $k = 3$, $y = 9 - 7.1x + 1.5x^2$

   (c) For each fitted curve $k = 1, 2, 3, 4$ give the mean square error where $MSE = \frac{SSE}{m}$
   Solution:

   - when $k = 1$, MSE = 3.250000
   - when $k = 2$, MSE = 3.050000
   - when $k = 3$, MSE = 0.800000
   - when $k = 4$, MSE = 0.000000

   Code used to implement this question can be seen in appendix. Function name: part1_1_1_c().

2. In this part we will illustrate the phenomena of *overfitting*.

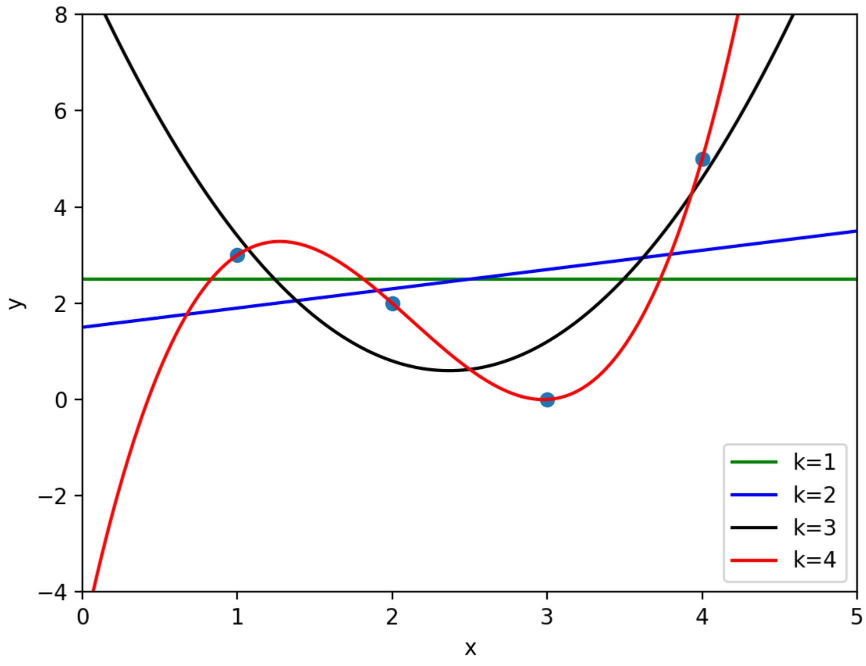

Figure 1: Curves corresponding to fit from dimension 1 to 4

(a) i. Plot the function $sin^2(2\pi x)$ in the range $0 \leq x \leq 1$ with the points of the above data set superimposed. The plot should resemble.
Solution: Figure 2 is the function $sin^2(2\pi x)$ in the range $0 \leq x \leq 1$ with the points of the above data set superimposed.
Code used to implement this question can be seen in appendix. Function name: part1_1_2_a_i().

ii. Fit the data set with a polynomial bases of dimension $k = 2, 5, 10, 14, 18$ plot each of these 5 curves superimposed over a plot of a data points.
Solution: Figure 3 is curves fitted with dataset using polynomial bases of dimension $k = 2, 5, 10, 14, 18$ superimposed over a plot of data points.
Code used to implement this question can be seen in appendix. Function name: part1_1_2_a_ii().

(b) Let the training error $te_k(S)$ denote the MSE of the fitting of the data set S with polynomial basis of dimension k. Plot the log of the training error versus the polynomial dimension $k = 1, ..., 18$(this should be a decreasing function).
Solution: Figure 4 is the plot of the log of the training error from polynomial dimension $k = 1$ to $k = 18$. It is a decreasing function
Code used to implement this question can be seen in appendix. Function name: part1_1_2_b().

(c) Generate a test set T of a thousand points,

$$T_{0.07,100} = \{(x_1, g_{0.07}(x1)), ..., (x_{1000}, g_{0.07}(x_{1000}))\}$$

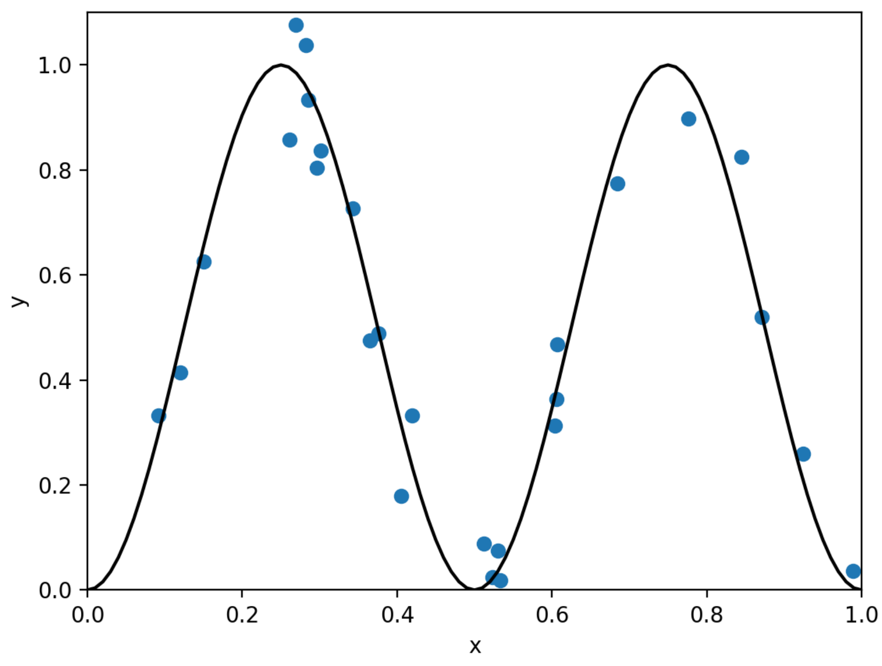

Figure 2: $sin^2(2\pi x)$ in the range $0 \leq x \leq 1$ with points within dataset

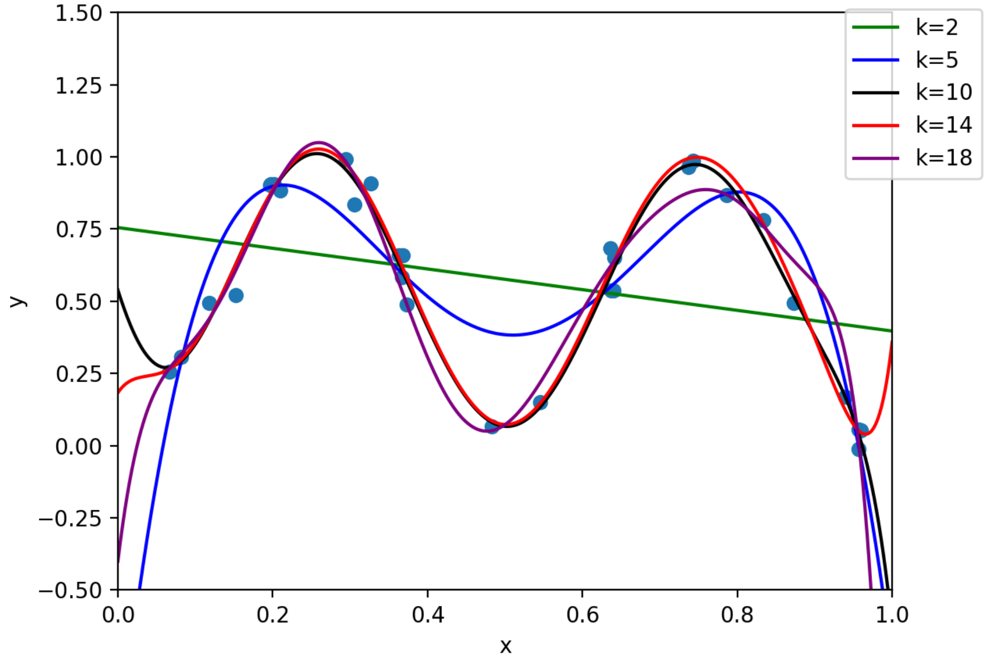

Figure 3: Curves with dimension $k = 2, 5, 10, 14, 18$ superimposed data points

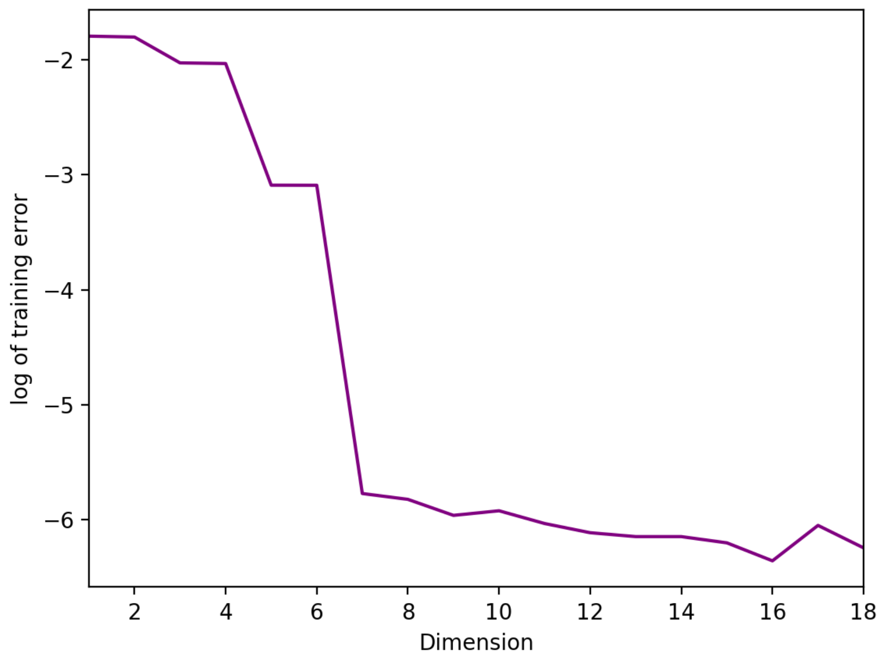

Figure 4: Log of training error from polynomial dimension $k = 1$ to $k = 18$

Define the test error $tse_k(S, T)$ to be the MSE of the test T on the polynomial of dimension k fitted from training set S. Plot the log of the test error versus the polynomial dimension $k = 1, ..., 18$.
Solution: Figure 5 is the plot of the log of the test error versus the polynomial dimension from $k = 1$ to $k = 18$.
Code used to implement this question can be seen in appendix. Function name: part1_1_2_c().

(d) For any given set of random numbers we will get slightly different training curves and test curves. It is instructive to see these curves smoothed out. For this part repeat items (b) and (c) but instead of plotting the results of a single run plot the average results of a 100 runs (note: plot the log(avg) rather than the avg(log)).
Solution: After repeat items (b) and (c) 100 times, figure 6 is the log of 100 times of average of training error and figure 7 is the log of 100 times average of testing error.
Code used to implement this question can be seen in appendix. Function name: part1_1_2_d().

3. Now use basis (for $k = 1, ..., 18$)

$$sin(1\pi x), sin(2\pi x), sin(3\pi x), ..., sin(k\pi x)$$

Repeat the experiments in 2 (b-d) with the above basis.
Solution: Figure 8 is the log of training error with new basis $sin(k\pi x)$. Figure 9 is the log of testing error with new basis $sin(k\pi x)$. Then after repeat previous steps 100 times,

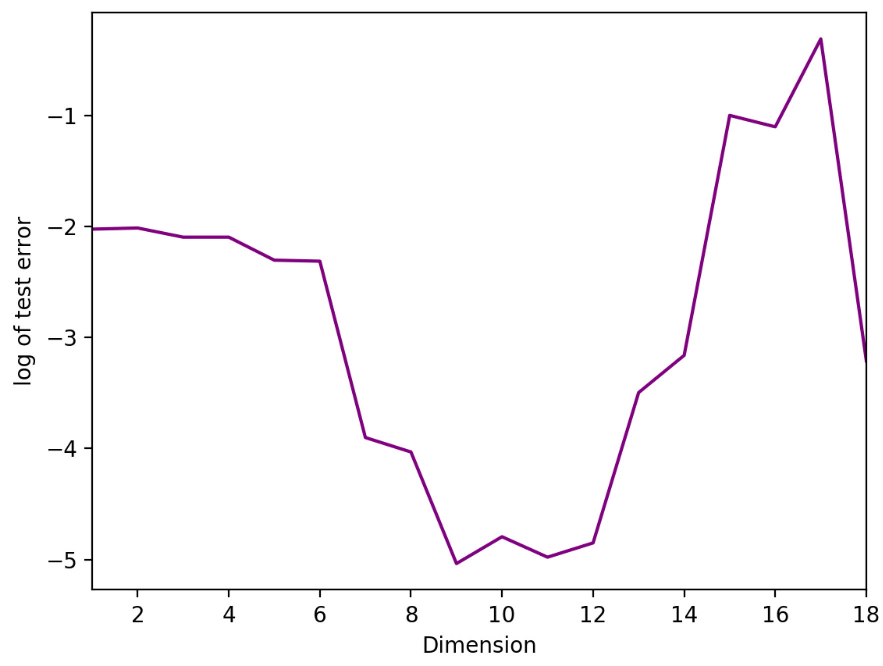

Figure 5: Log of test error from polynomial dimension $k = 1$ to $k = 18$

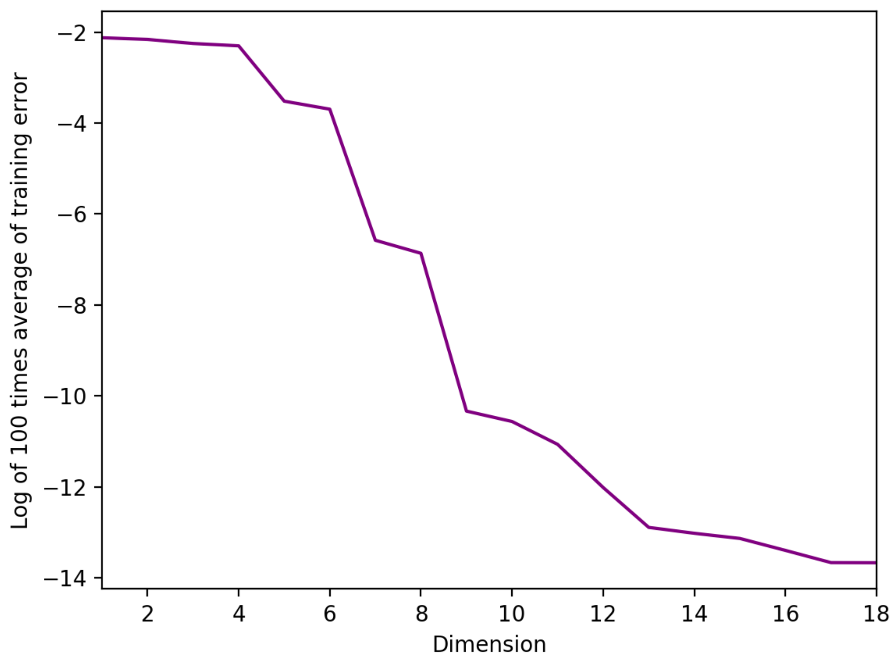

Figure 6: Log of 100 times average of training error from polynomial dimension $k = 1$ to $k = 18$

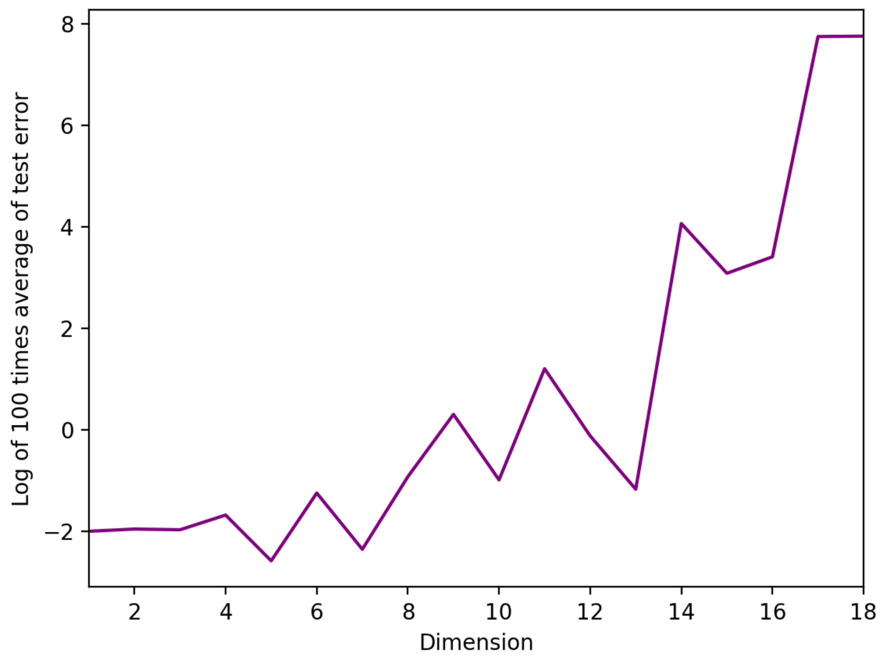

Figure 7: Log of 100 times average of testing error from polynomial dimension $k = 1$ to $k = 18$

figure 10 is the log of 100 times average training error and figure 11 is the log of 100 times testing error.
Code used to implement this question can be seen in appendix. Function name: part1_1_3().

1.2 Boston housing and kernels

4. a. Naive Regression Create a vector of ones that is the same length as the training set using the function ones. Do the same for the test set. By using these vectors we will be fitting the data with a constant function. Perform linear regression on the training set. Calculate the MSE on the training and test sets and note down the results.
   Solution:
   - Average MSE on the training set is: 83.18474650412023
   - Average MSE on the test set is: 84.90134359424864

   Code used to implement this question can be seen in appendix. Function name: part1_2_a().

   b. Give a simple interpretation of the constant function in a. above.
   Solution: Using constant vector w to fit the data is same as calculating the average of label values. Because it just add each data all together then divided it by the number of data.

   c. Linear Regression with single attributes. For each of the thirteen attributes, perform a linear regression using only the single attribute but incorporating a bias term so that

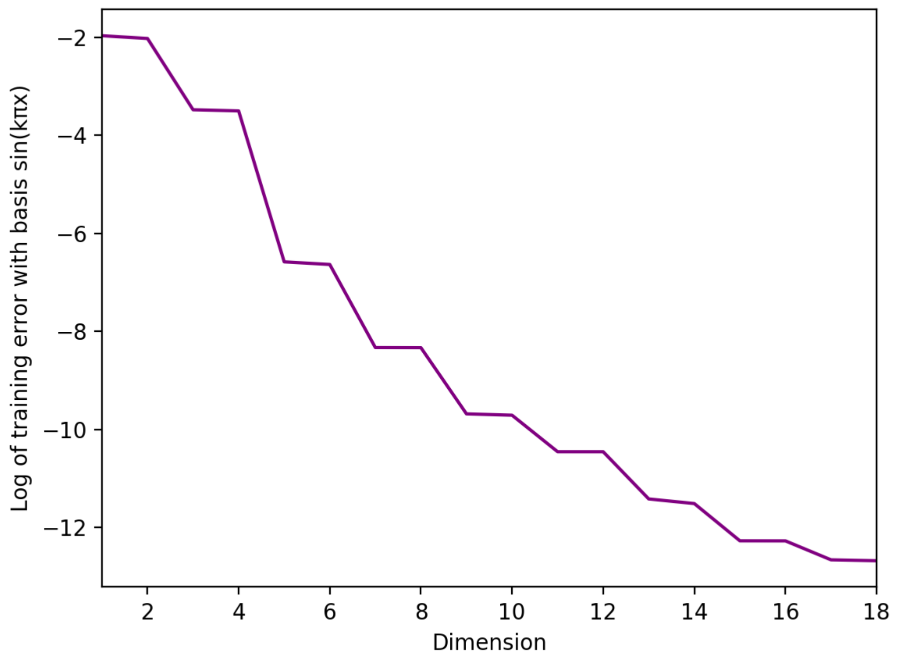

Figure 8: Log of training error with basis $sin(k\pi x)$ from dimension $k = 1$ to $k = 18$

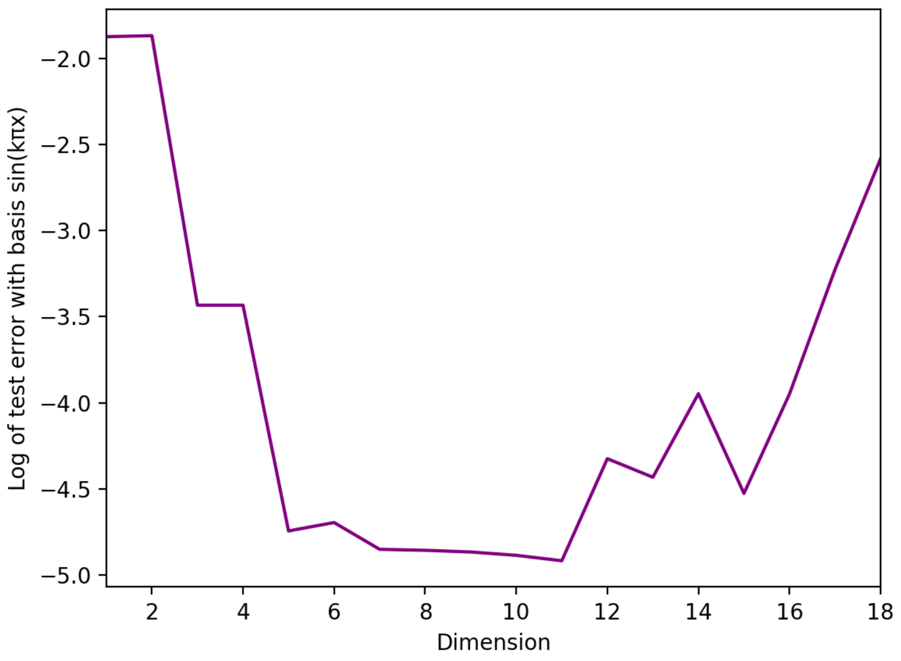

Figure 9: Log of testing error with basis $sin(k\pi x)$ from dimension $k = 1$ to $k = 18$

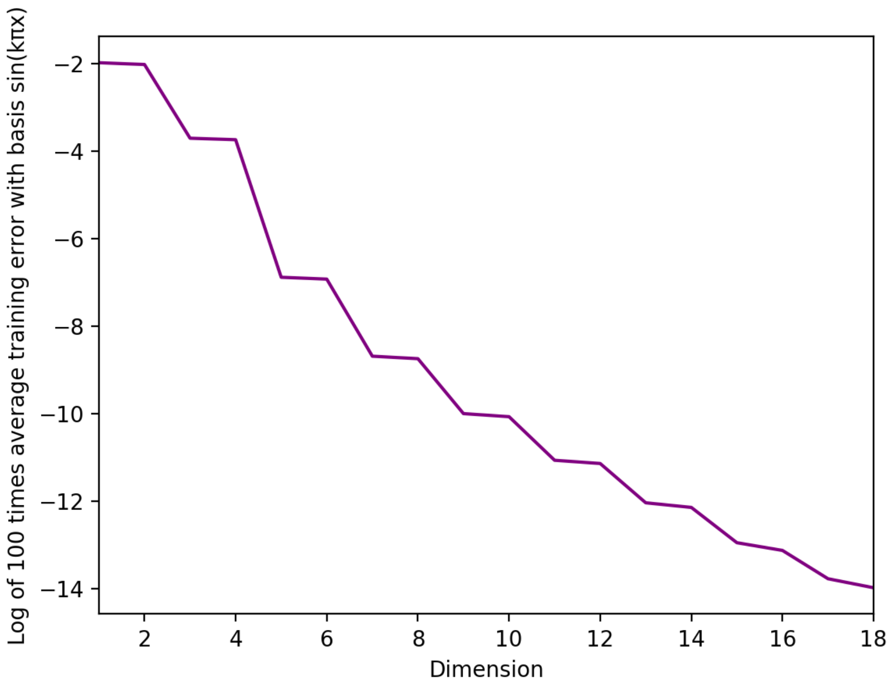

Figure 10: Log of 100 times average training error with basis $sin(k\pi x)$ from dimension $k = 1$ to $k = 18$

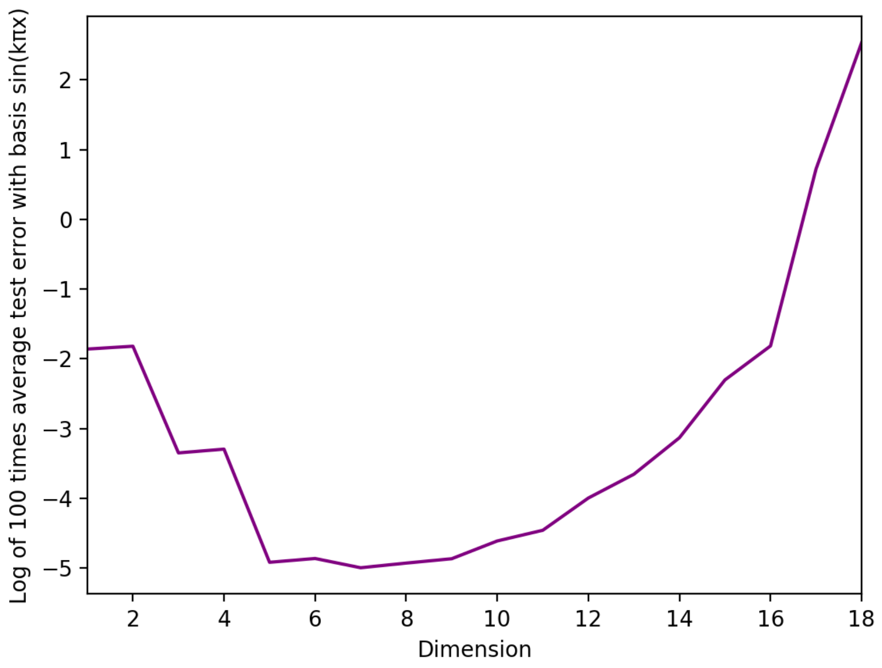

Figure 11: Log of 100 times average testing error with basis $sin(k\pi x)$ from dimension $k = 1$ to $k = 18$

the inputs are augmented with an additional 1 entry, $(x_i, 1)$, so that we learn a weight vector $\omega \in \Re^2$.
Solution:

- Linear regression attribute 1, MSE on training set: 72.30141326902927 , MSE on testing set: 69.10735255636949
- Linear regression attribute 2, MSE on training set: 74.0315638327025 , MSE on testing set: 71.29842498259369
- Linear regression attribute 3, MSE on training set: 65.45393344352883 , MSE on testing set: 62.31547619072579
- Linear regression attribute 4, MSE on training set: 82.30218185624605 , MSE on testing set: 79.4267323157446
- Linear regression attribute 5, MSE on training set: 69.79071425404584 , MSE on testing set: 66.65809840227988
- Linear regression attribute 6, MSE on training set: 43.72842548380269 , MSE on testing set: 42.65290144503359
- Linear regression attribute 7, MSE on training set: 73.35922379572553 , MSE on testing set: 69.65415973232386
- Linear regression attribute 8, MSE on training set: 79.92713993907894 , MSE on testing set: 76.62181525331376
- Linear regression attribute 9, MSE on training set: 73.0208459509806 , MSE on testing set: 69.59720176949277
- Linear regression attribute 10, MSE on training set: 66.79346452482874 , MSE on testing set: 63.3898277304275
- Linear regression attribute 11, MSE on training set: 63.279843291292046 , MSE on testing set: 60.492102193894276
- Linear regression attribute 12, MSE on training set: 75.66367779825535 , MSE on testing set: 72.92602617873219
- Linear regression attribute 13, MSE on training set: 38.91799370292425 , MSE on testing set: 37.20829362053159

Code used to implement this question can be seen in appendix. Function name: part1_2_c().

d. Linear Regression using all attributes. Now we would like to perform linear regression using all of the data attributes at once.
Perform linear regression on the training set using this regressor, and incorporate a bias term as above. Calculate the MSE on the training and test sets and note down the results. You should find that this method outperforms any of the individual regressors.
Solution:

- MSE on the training set is(with all attributes): 20.77517682633871
- MSE on the testing set is(with all attributes): 21.109852013414205

Code used to implement this question can be seen in appendix. Function name: part1_2_d().

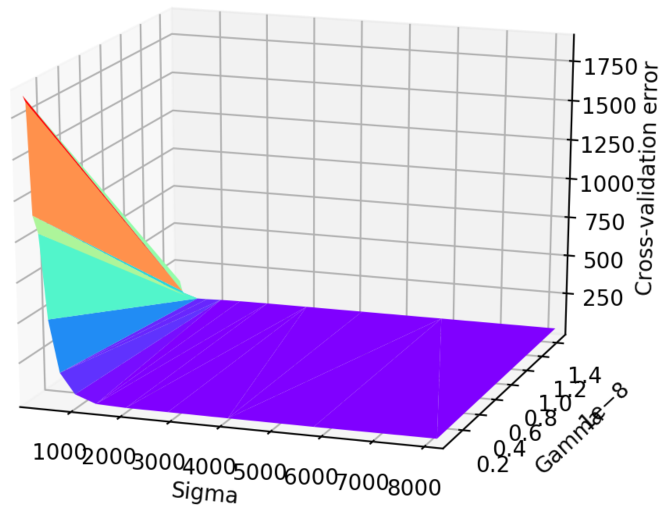

Figure 12: Plot of cross-validation error regarding to γ and σ in 3 dimension view.

1.3 Kernelised ridge regression

5. a. Create a vector of γ values $[2^{-40}, ..., 2^{-26}]$ and a vector of σ values $[2^{7}, 2^{7.5}, ..., 2^{12.5}, 2^{13}]$. Perform kernel ridge regression on the training set using five-fold cross-validation to choose among all pairing of the values of γ and σ. Choose the indices of the γ and σ values that perform the best to compute the predictor that you will use to report the test and training error.

   Solution: The best γ and σ values we choose to report the test and training error is:

   - $\gamma = 2^{-32}$
   - $\sigma = 2^{10}$

   Code used to implement this question and calculate best γ and σ values can be seen in appendix. Function name: part1_3a()

   b. Plot the cross-validation error (mean over folds of validation error) as a function of γ and σ.

   Solution: Figure 12 is the plot of cross-validation error in 3 dimensional view and figure 13 is the plot of cross-validation in matrix view.

   Code used to implement this question can be seen in appendix. Function name: part1_3b()

   c. Calculate the MSE on the training and test sets for the best γ and σ.

   Solution:

   - MSE on the training set for the best γ and σ is: 9.413394

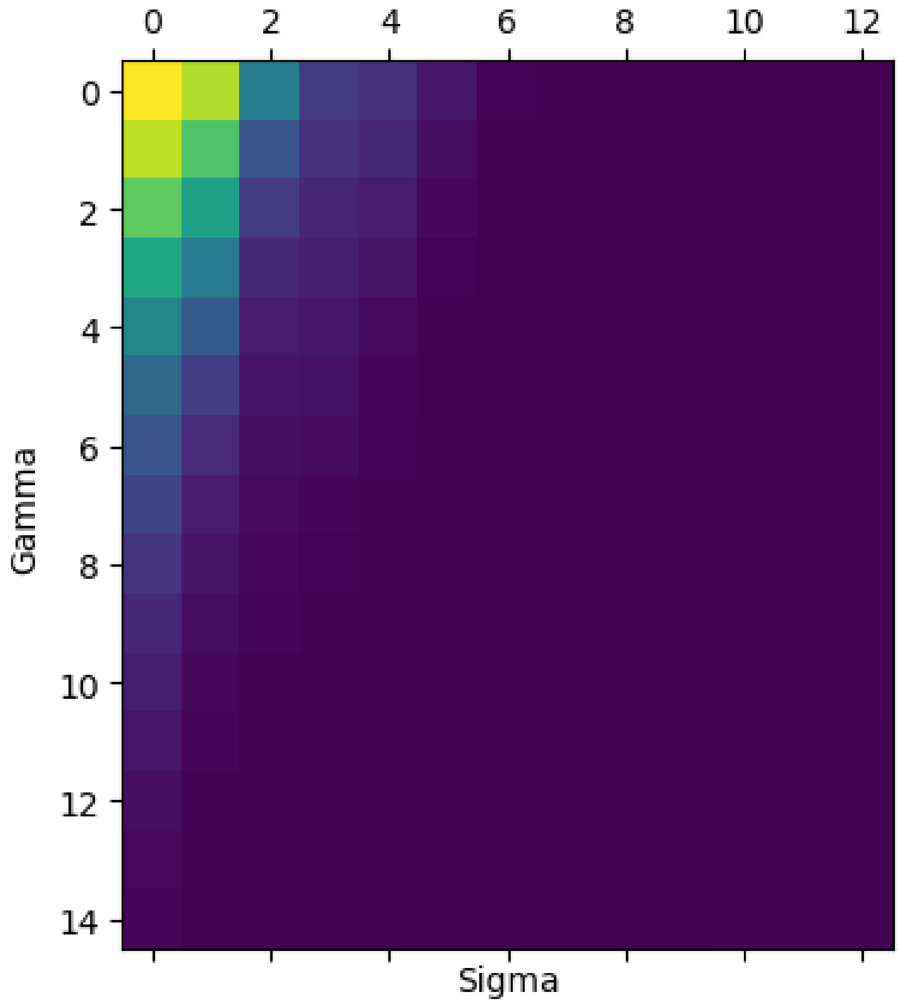

Figure 13: Plot of cross-validation error regarding to γ and σ in matrix view.

Method	**MSE train**	**MSE test**
Naive Regression	85.51 ± 3.50	88.82 ± 7.22
Linear Regression (attribute 1)	71.29 ± 4.61	70.77 ± 9.98
Linear Regression (attribute 2)	72.97 ± 4.16	73.32 ± 8.83
Linear Regression (attribute 3)	64.26 ± 4.18	64.77 ± 8.97
Linear Regression (attribute 4)	81.24 ± 4.62	81.73 ± 10.48
Linear Regression (attribute 5)	68.92 ± 4.15	68.54 ± 8.86
Linear Regression (attribute 6)	42.87 ± 4.36	44.24 ± 9.26
Linear Regression (attribute 7)	72.52 ± 4.86	71.49 ± 10.47
Linear Regression (attribute 8)	78.93 ± 4.85	78.63 ± 10.70
Linear Regression (attribute 9)	71.82 ± 4.41	71.87 ± 9.62
Linear Regression (attribute 10)	65.62 ± 4.17	65.58 ± 9.05
Linear Regression (attribute 11)	61.92 ± 3.52	63.14 ± 7.92
Linear Regression (attribute 12)	74.73 ± 3.92	74.81 ± 8.89
Linear Regression (attribute 13)	38.58 ± 2.31	37.79 ± 4.97
Linear Regression (all attributes)	21.49 ± 1.58	20.16 ± 3.56
Kernel Ridge Regression	7.78 ± 0.92	13.21 ± 2.81

Table 1: Results Summary

- MSE on the test set for the best γ and σ is: 16.937240

Code used to implement this question can be seen in appendix. Function name: part1_3c().

d. Repeat "exercise 4a,c,d" and "exercise 5c" over 20 random (2/3, 1/3) splits of your data record the train/test error and the standard deviations of the train/test errors and summarise these results in the following type of table.
Solution: After repeat "exercise 4a,c,d" and "exercise 5c" over 20 random (2/3, 1/3) splits of data. Table 1 is the table which contains the result.
Code used to implement this question can be seen in appendix. Function name: part1_3d()

2 PART II

2.1 Questions

6. *Bayes estimator.* In the both of the following subquestions you will need to find the Bayes estimator with respect to the the probability mass function $p(x, y)$ over (X, Y) where X and Y are finite thus $\sum_{x \in X} \sum_{y \in Y} p(x, y) = 1$.

   (a) For this subquestion $Y = [k]$ and let $c \in [0, \infty]^k$ be a vector of k costs. Define $L_c : [k] \times [k] \to [0, \infty]$

   Solution: $Y = [k]$, $L_c(y, \hat{y}) = [y \neq \hat{y}]c_y$, X and Y are *finite*. We need to minimise posterior expected loss $E\left(L_c(y,\hat{y})|X\right)$

$$\begin{aligned}
E\left(L_c(y,\hat{y})|X\right) &= \sum_{y=1}^{k} [y \neq \hat{y}]\, c_y\, P(y|X=x) \\
&= \sum_{y=1}^{k} [1-[y=\hat{y}]]\, c_y\, P(y|X=x) \\
&= \sum_{y=1}^{k} c_y\, P(y|X=x) - \underset{y\in[1,k]}{[y=\hat{y}]} c_y\, P(y|X=x)
\end{aligned} \tag{1}$$

$\sum_{y=1}^{k} c_y\, P(y|X=x)$ is a constant, so if we want to minimise posterior expected loss, we need to maximise $[y=\hat{y}]c_y\, P(y|X=x)$

So bayes estimator is $\hat{y} = \underset{y}{\operatorname{argmax}}\, c_y P(y|X=x)$.

(b) For this sub question $Y \subset \Re$. Let $L(y,\hat{y}) := |y-\hat{y}|$.Derive the Bayes estimator.

Solution: $Y \subset \Re$, $\hat{y} \in \Re$, X and Y are *finite*, $L(y,\hat{y}) := |y-\hat{y}|$

$$g(\hat{y}) = E(L_c(y,\hat{y})\,|x) = \sum_{y\in Y} |y-\hat{y}|\, P(y|x) \quad (\hat{y}\in\Re) \tag{2}$$

$g(\hat{y})$ is a continuous function. So we can take derivatives. We will use the following property.

$$\begin{aligned}
\frac{d|x|}{dx} &= \frac{d\sqrt{x^2}}{dx} = \frac{d((x^2)^{\frac{1}{2}})}{dx} \\
&= \frac{1}{2}(x^2)^{-\frac{1}{2}}x = \frac{x}{|x|}
\end{aligned} \tag{3}$$

$$\frac{d(g(\hat{y}))}{d\hat{y}} = \sum_{y\in Y} \frac{\hat{y}-y}{|y-\hat{y}|} P(y|x) \tag{4}$$

Since Y is *finite* set. So we rank every element in Y from smallest to largest $Y = [y_1, y_2 \ldots y_N]$. $\exists\, i$ such that $Y_1 = [y_1, y_2, \ldots y_i]$ and $Y_2 = [y_{i+1}, y_{i+2}, \ldots y_N]$ for $\sum_{y\in Y_1} \frac{\hat{y}-y}{|y-\hat{y}|} P(y|x) < 0$ and for $\sum_{y\in(Y_1\cup\{y_{i+1}\})} \frac{\hat{y}-y}{|y-\hat{y}|} P(y|x) \geqslant 0$.
Bayes estimator is $\hat{y} = y_{i+1}$, $F(\hat{y}|X) = \frac{1}{2}$, $\hat{y}$ is the median of the data set Y.

7. *Kernel modification* Consider the function $K_c(x,z) := c + \sum_{i=1}^{n} x_i z_i$ where $x, z \in \Re^n$.

(a) For what values of $c \in \Re$ is K_c a positive semidefinite kernel? Give an argument supporting your claim.
Solution: For $c \geq 0$ is K_c a positive semidefinite kernel. Function $K_c(x,z) := c + \sum_{i=1}^{n} x_i z_i$ can be written as $k_c(x,z) := c + x^T z$. For a polynomial kernel $K(x,t) = (a + x^T t)^r$ if we let $r = 1$ then we get the same form with K_c. $K(x,t)$

requires $a \geq 0$, then it is sufficient to say that when $c \geq 0$, K_c is a positive semidifinite kernel.
In order to prove this, assume $c \geq 0$, we have to show that the kernel matrix K_c should be a positive simi-defined matirx which means that for any give vector $t = [t_1, ..., t_n]$, $t^T K_c t \geq 0$.
Since:

$$\begin{aligned} t^T K_c t &= \sum_i \sum_j [\mathbf{K_c}]_{ij} t_i t_j \\ &= \sum_i \sum_j K_c(x_i, x_j) t_i t_j \\ &= \sum_i \sum_j (\sum_{k=1}^{n} x_{ik} x_{jk} + c) t_i t_j \\ &= \sum_i \sum_j (\sum_{k=1}^{n} x_{ik} x_{jk}) \sum_i \sum_j t_i t_j + c \sum_i \sum_j t_i t_j \\ &= \mathbf{K_c}^2 ||\mathbf{t}||^2 + c||\mathbf{t}||^2 \\ &\geq 0 \end{aligned} \tag{5}$$

Since K_c is positive semi-definite when $c \geq 0$, then function $K_c(x, z) := c + \sum_{i=1}^{n} x_i z_i$ is a positive semidifinite kernel when $c \geq 0$.

(b) Suppose we use K_c as a kernel function with linear regression (least squares). Explain how c influences the solution.
Solution: We use the linear regression with K_c when $c \geqslant 0$. $K_c(\boldsymbol{x}, \boldsymbol{z})$ is $\phi(\boldsymbol{x})$=$(x_1, x_2, \ldots x_l, \sqrt{c})^T$ so $K_c(\boldsymbol{x}, \boldsymbol{z})$=$\phi(\boldsymbol{x})^T \phi(\boldsymbol{z}) + c$ the traditional linear regression model is $y = \boldsymbol{w}^T \phi(\boldsymbol{x})$, thus, c will amplification the fluctuation range of model prediction value.
Regrading to this question we did some experiment in terms of different c. We use B̈oston Housingäs our data set. Figure 14 is plot of predicted y and actual y within the data set when $c = 9$. Figure 15 is the plot of predicted y and actual y within the data set when $c = 9^7$. Figure 16 is the plot of predicted y and actual y within the data set when $c = 9^{15}$ [H] By comparing figure 14, 15,16 we can conclude that c can affect the fluctuation range of predicted y which will course the increasing of the MSE.

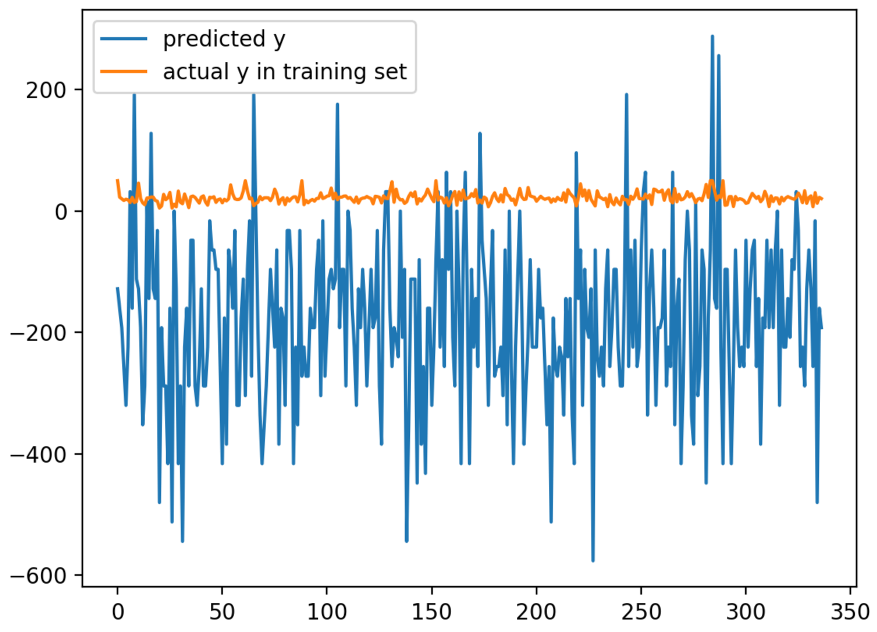

Figure 14: Plot of plot of predicted y and actual y when $c = 9$

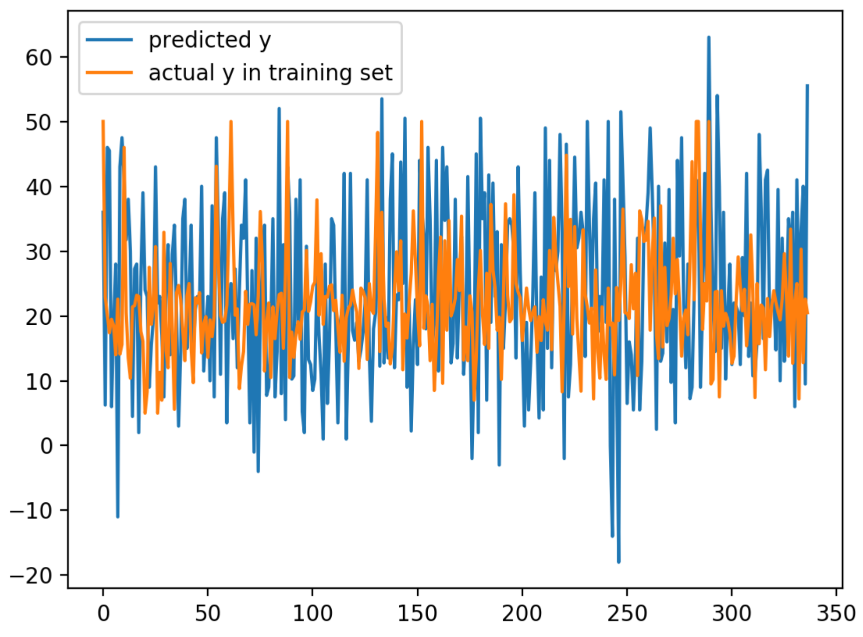

Figure 15: Plot of plot of predicted y and actual y when $c = 9^7$

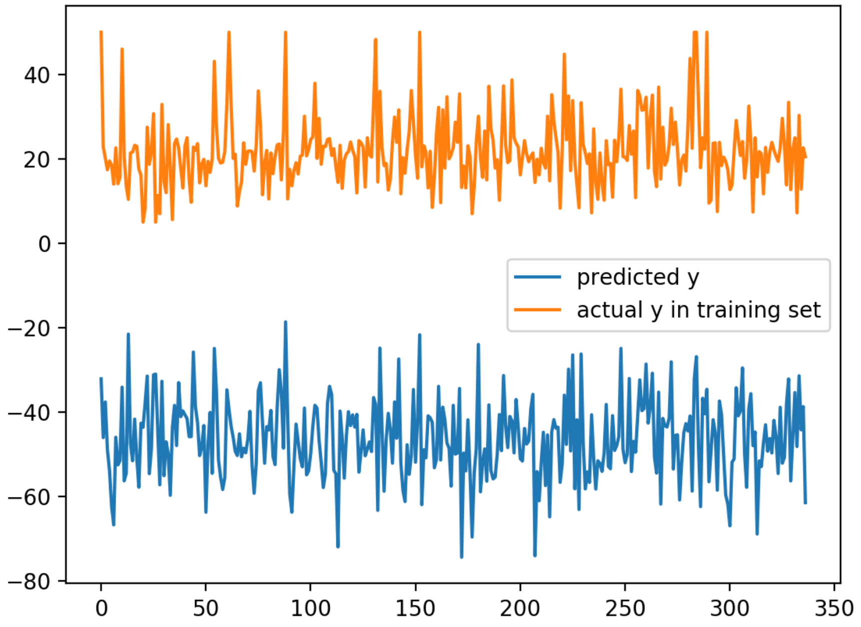

Figure 16: Plot of plot of predicted y and actual y when $c = 9^{15}$

8. Suppose we perform linear regression with a Gaussian kernel $K_\beta(x,t) = exp(-\beta||x - t||^2)$ to train a classifier for two-class data (i.e., $y \in \{-1, 1\}$). This classifier depends on the parameter β selected for the kernel. How should one choose β so that the learned linear classifier simulates a 1-NEAREST NEIGHBOR CLASSIFIER? Give an argument supporting your reasoning.

Solution: We need to simulate the 1-NEAREST NEIGHBOR CLASSIFIER using linear regression with a Gaussian kernel. Using the kernel trick, we have α^*: dual optimization formulation after kernelization which represents a linear weight vector. and

$$\begin{aligned} K_\beta(x,t) &= exp\left(-\beta||x-t||^2\right) \\ \alpha^* &= K_\beta^{-1}y \\ y_{test} &= \sum_{i=1}^{l} \alpha_i^* K_\beta(x_i, x_{test}) \end{aligned} \tag{6}$$

Since this linear classifier simulates a 1-NEAREST NEIGHBOR CLASSIFIER, we put x_{1NN} in to $1NN$ algorithm as we put corresponding x_{test} into linear regression with gaussian kernel. The label that assign to x_{test} should be same as y_{1NN}.So we need to make sure:
•$\alpha_{1NN}K(x_{1NN}, x_{test})$ should have largest weight in $\sum_{i=1}^{l} \alpha_i K(x_i, x_{test})$.

In $\alpha_{1NN}K(x_{1NN}, x_{test})$ which is $\alpha_{1NN}\, exp(-\beta||x_{1NN} - x_{test}||)$, the only variable we can adjust is β.
Comparing to other term $\alpha_{others}\, exp(-\beta||x_{others} - x_{test}||)$. We focus on the ratio of exp term

$$\frac{exp(-\beta||x_{1NN} - x_{test}||)}{exp(-\beta||x_{others} - x_{test}||)} = exp\left(-\beta(||x_{1NN} - x_{test}|| - ||x_{others} - x_{test}||)\right) \tag{7}$$

We need to make the ratio as large as possible. So β should be greater than zero and as large as possible theoretically.
$\alpha^* = K_\beta^{-1}y$ as $\beta \to \infty$ The kernel matrix will more likely to an identity matrix. Because the diagonal entries of kernel matrix will be 1 and non-diagonal entries of kernel martix will approach to 0. As $K^{-1} \to \infty$, $\alpha^* \to y$, $\alpha_{1NN} \to y_{1NN}$.
But in practical case, when $\beta \to \infty$ it will cause underflow when calculating the $exp(-\beta\, ||x_{1NN} - x_{test}||)$ So, when taking the value of β, we should take the maximum of the value that doesn't cause underflow.

9. Generalized Whack-A-Mole problem

Task: Design an algorithm for an $n \times n$ board which, given an initial board configuration, finds a sequence of holes that you can hit in order to empty the board if such a sequence exists. The algorithm must be polynomial in n and you must provide an argument that it is correct.

Solution: We can use matrix to express each situation on the board. For example, we are playing this game on a board with 2×2 holes, $(1,2),(2,1),(2,2)$ have moles while $(1,1)$ doesn't have mole. So we can use matrix

$$\begin{bmatrix} L_3 & L_4 \\ L_1 & L_2 \end{bmatrix}$$

And we use L_1, L_2, L_3, L_4 to record current state. L_i can be 1 or 0.

$$\begin{cases} L_1 = 1 \\ L_2 = 1 \\ L_3 = 0 \\ L_4 = 1 \end{cases} \tag{8}$$

If this hole has mole, we assign 1 to this element, if this hole doesn't have mole, we assign 0 to this element.

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

And we use x_1, x_2, x_3, x_4 to express the whack for each hole. If we whack hole 1 one time, $x_1 = 1$, similarly $x_1 = 0$ means we do nothing with hole.
operation x_1, x_2, x_3 can affect hole 1;
operation x_1, x_2, x_4 can affect hole 2;
operation x_1, x_3, x_4 can affect hole 3;
operation x_2, x_3, x_4 can affect hole 4;
In our algorithm, we have following theorem.

Corollary 2.1 (Operation corollary) *For the same hole, we whack even times is equal to whack zero time; we whack odd times is equal to whack one time.*

We assume at the first time, all the holes don't have any moles.

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

And we do the operation x_1, x_2, x_3, x_4 the situation should be

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

If we do the same operation again, according to operation corollary the situation will back to the original one.

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Because,we do the same operation twice, it won't change the state. In this case,

x_1, x_2, x_3, x_4 will be our solution to this state.

$$\begin{cases} x_1 + x_2 + x_3 = L_1 = 1 \\ x_1 + x_2 + x_4 = L_2 = 1 \\ x_1 + x_3 + x_4 = L_3 = 0 \\ x_2 + x_3 + x_4 = L_4 = 1 \end{cases} \tag{9}$$

So we transfer this game to a solving equations problem. All the solution $x_i, i \in n$ for $n \times n$ board is in $\{0, 1\}$.
By solving the equation above example situation, the only solution we have is:

$$\begin{cases} x_1 = 0 \\ x_2 = 1 \\ x_3 = 0 \\ x_4 = 0 \end{cases} \tag{10}$$

Which means we just need to whack hole 2, the hitting sequence is (2).
For $n \times n$ board, we can consider any initial board configuration as a $n \times n$ matrix K_n

$$\begin{bmatrix} L_{n^2-n+1} & L_{n^2-n+2} & L_{n^2-n+3} & \dots & L_{n^2} \\ L_{n^2-2n+1} & L_{n^2-2n+2} & L_{n^2-2n+3} & \dots & L_{n^2-n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_1 & L_2 & L_3 & \dots & L_n \end{bmatrix}$$

We use this matrix to record current state. And we have following linear equations

$$\begin{cases} x_1 + x_2 + x_{n+1} & = L_1 \\ x_1 + x_3 + x_{n+2} & = L_2 \\ x_2 + x_4 + x_{n+3} & = L_3 \\ \quad \vdots & \\ x_{n^2-1} + x_{n^2-n} + x_{n^2} & = L_{n^2} \end{cases} \tag{11}$$

So we have n^2 equations, we can use augmented matrix M_a to express above linear equations. And the matrix only conclude coefficients is coefficient matrix M_c.

$$\left[\begin{array}{cccc|c} x_1 & x_2 & \dots & x_{n+1} & L_1 \\ x_1 & x_3 & \dots & x_{n+2} & L_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{n^2-1} & x_{n^2-n} & \dots & x_{n^2} & L_{n^2} \end{array}\right]$$

Firstly, we should calculate the rank of coefficient matrix r_c and augmented matrix r_a. If $r_c < r_a$, this augmented matrix has no solution which mean corresponding situation has no solution. We are solving equations in $\{0, 1\}$, so the calculate rule will be slightly different especially for plus and minus:

$0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, $1 + 1 = 0$
$0 - 0 = 0$, $0 - 1 = 1$, $1 - 0 = 1$, $1 - 1 = 0$

Corollary 2.2 (rank in binary domain) .
(1) *When $r_c < r_a$, linear equations have no solution.*
(2) *When $r_c = r_a = n$, linear equations have unique solution.*
(3) *When $r_c = r_a = k < n$, the number of solutions for linear equations is 2^{n-k}. Because we have degree of freedom $n - k$ in binary domain.*

There is no infinity solution because we are solving the problem in binary domain. We use gaussian elimination to solve these n^2 equations. We perform three types of elementary row operations:
• Swapping two rows,
• Multiplying a row by a nonzero number,
• Adding a multiple of one row to another row.

By using above operations, we can transfer the augmented matrix into an upper triangular matrix, and in fact one that is in row echelon form. Then back substitute to get the results.
Here is the algorithm complexity:
According to gaussian elimination in Wikipedia it is $\mathcal{O}(n^6)$ to solve a n^2 equations because it requires:
• $\frac{n^2(n^2+1)}{2}$ divisions
• $\frac{2n^6+3n^4-5n^2}{6}$ multiplications
• $\frac{2n^6+3n^4-5n^2}{6}$ substractions

An average of $\frac{2n^6}{3}$ operations.

Appendix

Code for linear regression

```
"""
Created on Fri 20 Oct 2018

@author: Yuan Gao
"""
#------------------------------------------------
import numpy as np

def pl_featured(x, k):
  x_mat = np.zeros(shape=(len(x),k))#matrix initialize
  #construct X matrix after applying feature map
  for element in range(len(x)):
    pl_list = []
    for i in range(k):
      pl_list.append(x[element]**i)
    x_mat[element] = pl_list
  return x_mat

def pl_featured_sin(x, k):
  x_mat = np.zeros(shape=(len(x),k))#matrix initialize
  #construct X matrix after applying feature map
  for element in range(len(x)):
    pl_list = []
    for i in range(1,k+1):
      pl_list.append(np.sin(i * np.pi * x[element]))
    x_mat[element] = pl_list
  return x_mat

def fit(x,y,k):
  X = np.around(pl_featured(x, k).astype(np.float64), decimals = 6)
  return np.linalg.inv(np.dot(X.T,X)).dot(X.T).dot(y)

def fit_sin(x, y, k):
  x_mat = np.mat(pl_featured_sin(x, k))#convert nparray to matrix
  y_mat = np.mat(y).T#y transpose
  xTx = x_mat.T @ x_mat
  w = xTx.I @ x_mat.T @ y_mat#calculate w according to least square method
  return np.array(w.T).flatten()#convert matrix to list

def estimate(x, w):
  return x @ w.T
  #return w @ x.T

def mean_squared_error(y, y_e):
  diff = list(map(lambda x: (x[0]-x[1])**2, zip(y, y_e)))
  mse = np.sum(diff)/len(diff)
  return mse

#test code
```

```
'''
if __name__ == '__main__':
  x1          = np.random.uniform ( 0 , 1 , 30)
  epsilon = np.random.normal (0 , 0.07**2 , 30)
  y1          = np.square((np.sin ( 2 * np.pi * x1 )))+epsilon
  p = pl_featured(x1,1)
  alpha = fit(x1,y1,18)
  print(p)
  print(alpha)
'''
```

Code for kernelized ridge regression

```
"""
Created on Mon 29 Oct 2018

@author: Yuan Gao (18064382)
"""
#----------------------------------------------
import numpy as np


#method used to calculate kernel matrix K
def kernelMat(x_train, sigma):# arguments: array and scalar
  l = len(x_train)
  K = np.zeros((l,l))
  for i in range (0,l):
    K[i,i] = 1
    for j in range (i+1,l):
      K[i,j] = np.exp(-np.linalg.norm(x_train[i]-x_train[j])**2/(2*sigma**
                                    2))
      K[j,i] = K[i,j]
  K = np.mat(K)
  return K

#method used to train model and return an array of alpha
def fit(xi, sigma, gamma, y):
  alpha = np.linalg.inv((kernelMat(xi,sigma) + gamma* len(xi) * np.
                                     identity(len(xi)))) @ y
  #print(alpha)
  return np.array(alpha).flatten()

#method used to calculate value Gaussian kernel
def Gaussian_kernel(Xi,Xj,sigma): # arguments: arrays and scalar
  K = np.exp(-np.linalg.norm(Xi - Xj)**2/(2*sigma**2))
  return K

#method used to predict value
def estimate(x,xtest,sigma,alpha):
  y_e = np.zeros((len(xtest),1))
  for i in range(len(xtest)):
    for j in range(len(x)):
      y_e[i] += alpha[j]*Gaussian_kernel(x[j],xtest[i],sigma)
  return y_e

'''
#test code
if __name__ == '__main__':

  xi = np.array([[1,2,3],[3,2,1]])
  xj = np.array([[6,7,8],[3,2,2]])
  y = np.array([1,2])
  print(kernelMat(xi,xj,0.5))
  print(fit(xi,0.5,0.3,y))
  print(estimate(xi,xj,0.5,a))
```

```
'''
```

Code for main program of Part I

```
"""
Created on Fri 20 Oct 2018

@author: Yuan Gao (18064382), Yuan Li (18057497)
"""
#--------------------------------------------------
import numpy as np
import matplotlib.pyplot as plt
import linaerRegression as lr
import scipy.io as scio
import KernelisedRR as krr
from sklearn.model_selection import train_test_split, KFold
from numpy.linalg import inv
from mpl_toolkits.mplot3d import Axes3D
from prettytable import PrettyTable

x = [1, 2, 3, 4]
y = [3, 2, 0, 5]

def part1_1_1_a():
    #define all variables as global variables
    global model_d1, model_d2, model_d3, model_d4

    #calculate w for dimension 1 to 4
    model_d1 = lr.fit(x, y, 1)
    model_d2 = lr.fit(x, y, 2)
    model_d3 = lr.fit(x, y, 3)
    model_d4 = lr.fit(x, y, 4)

    #configuration of plot
    plt.xlim(0, 5)
    plt.ylim(-4, 8)
    plt.xlabel('x')
    plt.ylabel('y')

    #draw plot
    x_axis1 = np.linspace(0, 5, num = 1000)
    y_axis1 = np.linspace(2.5, 2.5, num = 1000)
    '''
    y_axis2 = 1.5 + 0.4 * x_axis1
    y_axis3 = 9 - 7.1 * x_axis1 + 1.5 * x_axis1 ** 2
    y_axis4 = -5 + 15.17 * x_axis1 - 8.5 * x_axis1 ** 2 + 1.33 * x_axis1
                                                  ** 3
    '''
    y_axis2 = lr.estimate(lr.pl_featured(x_axis1,2), model_d2)
    y_axis3 = lr.estimate(lr.pl_featured(x_axis1,3), model_d3)
    y_axis4 = lr.estimate(lr.pl_featured(x_axis1,4), model_d4)

    plt.scatter([1,2,3,4],[3,2,0,5])
    l1, = plt.plot(x_axis1, y_axis1, color='green', label='dimension 1')
    l2, = plt.plot(x_axis1, y_axis2, color='blue',  label='dimension 2')
    l3, = plt.plot(x_axis1, y_axis3, color='black', label='dimension 3')
    l4, = plt.plot(x_axis1, y_axis4, color='red',   label='dimension 4')
    # Place a legend to the right of this smaller subplot.
    plt.legend(handles = [l1,l2,l3,l4],labels = ['k=1','k=2','k=3','k=4'],
                                                 loc='best')
    plt.show()

def part1_1_1_b():

    print('y1 = 2.5')
    print('y2 = 1.5 + 0.4 * x')
    print('y3 = 9 - 7.1 * x + 1.5 * x ** 2')
    print('y4 = -5 + 15.17 * x - 8.5 * x ** 2 + 1.33 * x ** 3')

#This function is depend on part1_1_1_a
#part1_1_1_a should be run before this function
def part1_1_1_c():
    #calculate estimate value for different dimension
    estimate_y1 = lr.estimate(lr.pl_featured(x,1), model_d1)
    estimate_y2 = lr.estimate(lr.pl_featured(x,2), model_d2)
    estimate_y3 = lr.estimate(lr.pl_featured(x,3), model_d3)
    estimate_y4 = lr.estimate(lr.pl_featured(x,4), model_d4)

    #calculate MSEs
    mse1 = lr.mean_squared_error(y, estimate_y1)
    mse2 = lr.mean_squared_error(y, estimate_y2)
    mse3 = lr.mean_squared_error(y, estimate_y3)
    mse4 = lr.mean_squared_error(y, estimate_y4)

    print("MSE 1D: %f, MSE 2D: %f, MSE 3D: %f, MSE 4D: %f" % (mse1, mse2,
                                                mse3, mse4))

def part1_1_2_a_i():
    x1      = np.random.uniform ( 0 , 1    , 30 )
    epsilon = np.random.normal  ( 0 , 0.07 , 30 )
    g       = ( np.sin( 2 * np.pi * x1 )** 2 ) + epsilon
    xrange  = np.linspace( 0 , 1 , num = 101 )

    #configuration of plot
    plt.xlim(0 , 1  )
    plt.ylim(0 , 1.1)
    plt.xlabel('x')
    plt.ylabel('y')
    #plot the data set and function
    plt.plot(xrange, ( np.sin ( 2 * np.pi * xrange ) ** 2 ), color='black'
                                                 )
    plt.scatter( x1 , g )
    plt.show()

def part1_1_2_a_ii():
    x1         = np.random.uniform ( 0 , 1    , 30 )
    epsilon    = np.random.normal  ( 0 , 0.07 , 30 )
    y1         = ( np.sin( 2 * np.pi * x1 )** 2 ) + epsilon
    model_d2   = lr.fit(x1, y1, 2)
    model_d5   = lr.fit(x1, y1, 5)
```

```
model_d10 = lr.fit(x1, y1, 10)
model_d14 = lr.fit(x1, y1, 14)
model_d18 = lr.fit(x1, y1, 18)
x_axis1   = np.linspace(0, 1, num=1000)
# configuration of plot
plt.xlim(0, 1)
plt.ylim(-0.5, 1.5)
plt.xlabel('x')
plt.ylabel('y')



estimate_y2 = lr.estimate(lr.pl_featured(x_axis1, 2), model_d2)
l1, = plt.plot(x_axis1, estimate_y2, color='green', label='dimension 2
                                     ')
plt.scatter(x1, y1)
# Place a legend to the right of this smaller subplot.
#plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
#plt.show()


estimate_y5 = lr.estimate(lr.pl_featured(x_axis1, 5), model_d5)
l2, = plt.plot(x_axis1, estimate_y5, color='blue', label='dimension 5'
                                     )
#plt.scatter(x1, y1)
# Place a legend to the right of this smaller subplot.
#plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
#plt.show()


estimate_y10 = lr.estimate(lr.pl_featured(x_axis1, 10), model_d10)
l3, = plt.plot(x_axis1, estimate_y10, color='black', label='dimension
                                     10')
#plt.scatter(x1, y1)
# Place a legend to the right of this smaller subplot.
#plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
#plt.show()


estimate_y14 = lr.estimate(lr.pl_featured(x_axis1, 14), model_d14)
l4, = plt.plot(x_axis1, estimate_y14, color='red',  label='dimension
                                     14')
#plt.scatter(x1, y1)
# Place a legend to the right of this smaller subplot.
#plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
#plt.show()


estimate_y18 = lr.estimate(lr.pl_featured(x_axis1, 18), model_d18)
l5, = plt.plot(x_axis1, estimate_y18, color='purple', label='dimension
                                      18')
#plt.scatter(x1, y1)
# Place a legend to the right of this smaller subplot.
```

```
    plt.legend(handles = [l1,l2,l3,l4,l5],labels = ['k=2','k=5','k=10','k=
                                            14','k=18'], bbox_to_anchor=(1.13
                                            ,1.025),loc='upper right')
    plt.show()

#calculate MSE from 1d to 18d with original basis
def mse_1d_to_18d(x1, y1, xt, yt):
    mse = np.zeros(shape=(18,1))
    for k in range(1,19):
        #calculate w for dimension 1 to 18
        model_dk     = lr.fit(x1, y1, k)
        #calculate estimate value for different dimension
        estimate_yt  = lr.estimate(lr.pl_featured(xt,k), model_dk)
        #calculate MSEs
        mse[k-1]     = lr.mean_squared_error(yt, estimate_yt)
    return mse

#calculate MSE from 1d to 18d with new basis
def mse_1d_to_18d_sin(x1, y1, xt, yt):
    mse = np.zeros(shape=(18,1))
    for k in range(1,19):
        #calculate w for dimension 1 to 18
        model_dk     = lr.fit_sin(x1, y1, k)
        #calculate estimate value for different dimension
        estimate_yt  = lr.estimate(lr.pl_featured_sin(xt,k), model_dk)
        #calculate MSEs
        mse[k-1]         = lr.mean_squared_error(yt, estimate_yt)
    return mse

def part1_1_2_b():
    x1          = np.random.uniform ( 0  , 1    , 30 )
    epsilon = np.random.normal (0 , 0.07 , 30 )
    y1          = (np.sin ( 2 * np.pi * x1 ) ** 2)+epsilon
    #calculate mse between training set and fitting result from 1d to 18d
    mse = np.log(mse_1d_to_18d(x1,y1,x1,y1))
    x_axis1 = np.linspace(1, 18, num=18)

    # configuration of plot
    plt.xlim(1, 18)
    plt.xlabel('Dimension')
    plt.ylabel('log of training error')
    plt.plot(x_axis1, mse, color='purple', label='log of MSE vs k')

    # Place a legend to the right of this smaller subplot.
    #plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
    plt.show()

def part1_1_2_c():
    #training set
    x1          = np.random.uniform ( 0  , 1    , 30 )
    train_epsilon = np.random.normal (0 , 0.07 , 30 )
    y1          = (np.sin ( 2 * np.pi * x1 ) ** 2)+ train_epsilon
    #test set
    xc = np.random.uniform(0 , 1    , 1000 )
    test_epsilon = np.random.normal (0 , 0.07 , 1000 )
    gc          = ( np.sin( 2 * np.pi * xc )** 2 ) + test_epsilon
    #calculate mse between test set and fitting result from 1d to 18d
    mse = np.log(mse_1d_to_18d(x1,y1,xc,gc))
    x_axis1 = np.linspace(1, 18, num=18)
    # configuration of plot
    plt.xlim(1, 18)
    plt.xlabel('Dimension')
    plt.ylabel('log of test error')
    plt.plot(x_axis1, mse, color='purple', label='log of MSE vs k with
                                             epsilon')
    # Place a legend to the right of this smaller subplot.
    #plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
    plt.show()

def part1_1_2_d():
    #loop b for 100 times
    mse = np.zeros(shape=(100,18))
    for i in range(100):
        x1 = np.random.uniform ( 0 , 1    , 30 )
        y1 = np.sin ( 2 * np.pi * x1 ) ** 2
        mse_inloop = mse_1d_to_18d(x1,y1,x1,y1)
        mse[i] = mse_inloop.T
    #calculate average for each column within mse matrix
    mse_meaned = np.log(np.mean(mse, axis=0))
    #plot
    x_axis1 = np.linspace(1, 18, num=18)
    # configuration of plot
    plt.xlim(1, 18)
    plt.xlabel('Dimension')
    plt.ylabel('Log of 100 times average of training error')
    plt.plot(x_axis1, mse_meaned, color='purple', label='100 times average
                                              MSE from 1d to 18d')
    # Place a legend to the right of this smaller subplot.
    #plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
    plt.show()

    #loop c for 100 times
    mse_c = np.zeros(shape=(100,18))
    for i in range(100):
        #training set
        x1           = np.random.uniform ( 0 , 1    , 30 )
        train_epsilon = np.random.normal (0 , 0.07 , 30 )
        y1           = (np.sin ( 2 * np.pi * x1 ) ** 2)+ train_epsilon
        #test set
        xc = np.random.uniform(0 , 1    , 1000 )
        test_epsilon = np.random.normal (0 , 0.07 , 1000 )
        gc           = ( np.sin( 2 * np.pi * xc )** 2 ) + test_epsilon

        mse_inloop2 = mse_1d_to_18d(x1,y1,xc,gc)
        mse_c[i] = mse_inloop2.T
    #calculate average for each column within mse_c matrix
    mse_c_meaned = np.log(np.mean(mse_c, axis=0))
    #plot
    x_axis1 = np.linspace(1, 18, num=18)
    # configuration of plot
    plt.xlim(1, 18)
    plt.xlabel('Dimension')
    plt.ylabel('Log of 100 times average of test error')
    plt.plot(x_axis1, mse_c_meaned, color='purple', label='100 times
                                              average MSE from 1d to 18d with
                                              epsilon')
    # Place a legend to the right of this smaller subplot.
    #plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
    plt.show()

def part1_1_3():
    #experiment 2b with new basis
    x1          = np.random.uniform ( 0 , 1    , 30 )
    y1          = np.sin ( 2 * np.pi * x1 ) ** 2
    mse_b = np.log(mse_1d_to_18d_sin(x1,y1,x1,y1))
    x_axis1 = np.linspace(1, 18, num=18)
    # configuration of plot
    plt.xlim(1, 18)
    plt.xlabel('Dimension')
    plt.ylabel('Log of training error with basis sin( k  x )')
    plt.plot(x_axis1, mse_b, color='purple', label='log of MSE vs k with
                                              basis sin( k  x )')
    # Place a legend to the right of this smaller subplot.
    #plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
    plt.show()

    #experiment 2c with new basis
    #training set
    x1          = np.random.uniform ( 0 , 1    , 30 )
    train_epsilon = np.random.normal (0 , 0.07 , 30 )
    y1          = (np.sin ( 2 * np.pi * x1 ) ** 2)+ train_epsilon
    #test set
    xc = np.random.uniform(0 , 1    , 1000 )
    test_epsilon = np.random.normal (0 , 0.07 , 1000 )
    gc          = ( np.sin( 2 * np.pi * xc )** 2 ) + test_epsilon

    mse_c = np.log(mse_1d_to_18d_sin(x1,y1,xc,gc))
    x_axis1 = np.linspace(1, 18, num=18)
    # configuration of plot
    plt.xlim(1, 18)
    plt.xlabel('Dimension')
    plt.ylabel('Log of test error with basis sin( k  x )')
    plt.plot(x_axis1, mse_c, color='purple', label='log of MSE vs k with
                                              epsilon with basis sin( k  x ))')
    # Place a legend to the right of this smaller subplot.
    #plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
    plt.show()

    #experiment 2d with new basis
    #loop b for 100 times
    mse_d_b = np.zeros(shape=(100,18))
    for i in range(100):
        x1 = np.random.uniform ( 0 , 1    , 30 )
        y1 = np.sin ( 2 * np.pi * x1 ) ** 2
        mse_inloop = mse_1d_to_18d_sin(x1,y1,x1,y1)
        mse_d_b[i] = mse_inloop.T
    #calculate average for each column within mse matrix
    mse_d_b_meaned = np.log(np.mean(mse_d_b, axis=0))
    #plot
    x_axis1 = np.linspace(1, 18, num=18)
    # configuration of plot
    plt.xlim(1, 18)
    plt.xlabel('Dimension')
    plt.ylabel('Log of 100 times average training error with basis sin(
                                            k x)')
    plt.plot(x_axis1, mse_d_b_meaned, color='purple', label='100 times
                                            average MSE from 1d to 18d with
                                            basis sin(k x)')
    # Place a legend to the right of this smaller subplot.
    #plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
    plt.show()

    #loop c for 100 times
    mse_d_c = np.zeros(shape=(100,18))
    for i in range(100):
        #training set
        x1          = np.random.uniform ( 0 , 1    , 30 )
        train_epsilon = np.random.normal (0 , 0.07 , 30 )
        y1          = (np.sin ( 2 * np.pi * x1 ) ** 2)+ train_epsilon
        #test set
        xc = np.random.uniform(0 , 1    , 1000 )
        epsilon = np.random.normal (0 , 0.07 , 1000 )
        gc          = ( np.sin( 2 * np.pi * xc )** 2 ) + epsilon

        mse_inloop2 = mse_1d_to_18d_sin(x1,y1,xc,gc)
        mse_d_c[i] = mse_inloop2.T
    #calculate average for each column within mse_c matrix
    mse_d_c_meaned = np.log(np.mean(mse_d_c, axis=0))
    #plot
    x_axis1 = np.linspace(1, 18, num=18)
    # configuration of plot
    plt.xlim(1, 18)
    plt.xlabel('Dimension')
    plt.ylabel('Log of 100 times average test error with basis sin(k x)')
    plt.plot(x_axis1, mse_d_c_meaned, color='purple', label='100 times
                                            average MSE from 1d to 18d with
                                            epsilon with basis sin(k x)')
    # Place a legend to the right of this smaller subplot.
    #plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
    plt.show()


def part1_2a():
    #load the data
    data                    = (scio.loadmat('boston.mat'))['boston']
    #spile the train data and test data
    data_train ,data_test = train_test_split(data,test_size=0.33)
    #initialize the train and test sets
    x_train                = np.asmatrix(np.ones(len(data_train))).T
    x_test                 = np.asmatrix(np.ones(len(data_test))).T
    mse_price_train        = np.zeros(20)
    mse_price_test         = np.zeros(20)
    for j in range(20):
        #fit the data with constant function
        w_price_train      = inv((x_train.T)@(x_train))@(x_train.T)
                                               @data_train[:,13]
        w_price_test       = inv((x_test.T)@(x_test))@(x_test.T)@data_test[
                                               :,13]
        #calculate MSEs
        mse_price_train[j]= lr.mean_squared_error(x_train*w_price_train,
                                               data_train[:,13])
        mse_price_test [j]= lr.mean_squared_error(x_test*w_price_test  ,
                                               data_train[:,13])

        data_train ,data_test = train_test_split(data,test_size=0.33)
    train_std = np.std(mse_price_train, ddof = 1)
    test_std = np.std(mse_price_test, ddof = 1)
    mse_price_train        = np.sum(mse_price_train)/20
    mse_price_test         = np.sum(mse_price_test )/20
    print('average MSE for train set\t',mse_price_train,'\naverage MSE for
                                           test set\t',mse_price_test)
    return mse_price_train, mse_price_test, train_std, test_std

def part1_2c():
    #load the data
    data                   = (scio.loadmat('boston.mat'))['boston']
    #spile the train data and test data
    data_train ,data_test = train_test_split(data,test_size=0.33)
    #initialize the MSE
    mse_price_train        = np.zeros((20,13))
    mse_price_test         = np.zeros((20,13))
    mse_mean_price_train   = np.zeros((1,13))
    mse_mean_price_test   = np.zeros((1,13))
    for j in range(20):
        for i in range(13):
            #initialize the train and test sets
            x_train                = data_train[:,i]
            x_test                 = data_test [:,i]
            #add a bias term
            x_train                = np.asmatrix(np.vstack((x_train,np.ones(
                                           len(data_train))))).T#339
                                           *2
            x_test                 = np.asmatrix(np.vstack((x_test,np.ones(
                                           len(data_test))))).T
            mse_price_train[j,i] = lr.mean_squared_error((x_train@(inv((
                                           x_train.T)@(x_train))@(
                                           x_train.T)@data_train[:,
                                           13]).T), data_train[:,13]
                                           )
            mse_price_test [j,i] = lr.mean_squared_error((x_test@(inv((
                                                    x_test.T)@(x_test))@(
                                                    x_test.T)@data_test[:,13]
                                                    ).T), data_test[:,13])

        #random spilt the train and test set after each linear regression
        data_train ,data_test = train_test_split(data,test_size=0.33)
    #take the average for results over 20 runs
    train_std = np.std(mse_price_train,axis = 0, ddof = 1)
    test_std = np.std(mse_price_test,axis = 0, ddof = 1)
    mse_mean_price_train = np.sum(mse_price_train,axis = 0)/20
    mse_mean_price_test  = np.sum(mse_price_test,axis = 0 )/20
    print("For liner regression with single attribute\t ")
    for i in range(len(mse_mean_price_train)):
        print("Linear regression attribute",i+1,"\tMSE train\t",
                                                mse_mean_price_train[i],"\
                                                tMSE test\t",
                                                mse_mean_price_test[i],"\n")
    return mse_mean_price_train, mse_mean_price_test, train_std, test_std

def part1_2d():
    #load the data
    data                  = (scio.loadmat('boston.mat'))['boston']
    #spile the train data and test data
    data_train ,data_test = train_test_split(data,test_size=0.33)
    #initialize the vector w and b
    mse_price_train = np.zeros(20)
    mse_price_test  = np.zeros(20)
    for j in range(20):
        #initialize the train and test sets
        x_train             = data_train[:,range(13)]
        x_test              = data_test [:,range(13)]
        #add a bias term
        x_train             = np.asmatrix(np.c_[(x_train,np.ones(len(
                                                data_train)))])
        x_test              = np.asmatrix(np.c_[(x_test ,np.ones(len(
                                                data_test )))])
        mse_price_train[j] = lr.mean_squared_error((x_train@((inv((x_train
                                                .T)@(x_train))@(x_train.T)
                                                @data_train[:,13]).T)),
                                                data_train[:,13])
        mse_price_test [j] = lr.mean_squared_error((x_test@((inv((x_test.T
                                                )@(x_test))@(x_test.T)
                                                @data_test[:,13]).T)),
                                                data_test[:,13])
        #random spilt the train and test set after each linear regression
        data_train ,data_test = train_test_split(data,test_size=0.33)
    #calculate standard deviations on training set and testing set
    train_std = np.std(mse_price_train, ddof = 1)
    test_std = np.std(mse_price_test, ddof = 1)
    #take the average MSE for results over 20 runs on the train and test
                                        set
    mse_price_train = np.sum(mse_price_train)/20
    mse_price_test  = np.sum(mse_price_test )/20
    print("Linear regression with all attributes\t",'\t MSE train\t',
                                             mse_price_train,'\tMSE test\t',
                                             mse_price_test)
    return mse_price_train, mse_price_test, train_std, test_std

def part1_3a():
    #load data from .mat file
    data = (scio.loadmat('boston.mat'))['boston']
    '''full data
    x = data[:,range(13)]
    y = data[:,13]
    '''
    data_train ,data_test = train_test_split(data,test_size=0.33)
    x_train = data_train[:,range(13)]
    y_train = data_train[:,13]
    x_test = data_test[:,range(13)]
    y_test = data_test[:,13]
    #x = np.array([[1,2,3,4],[5,6,7,8],[2,3,4,5],[7,8,9,10],[5,7,8,9]])
    #y = [1,2,3,4,5]
    global record_list, cv_mse_mat
    #construct gamma vector and sigma vector
    gamma = []
    sigma = []
    for i in range(-40, -25):
        gamma.append(2**i)
    for i in np.arange(7,13.5,0.5):
        sigma.append(2**i)

    #for all permutations do cross validation
    cv_mse_mat = np.zeros((len(gamma),len(sigma)))
    record_list = []
    counter = 0
    min_mse = float('inf')
    min_gamma_idx = 0
    min_sigma_idx = 0
    for i in range(len(gamma)):
        for j in range(len(sigma)):
            mean_mse = k_fold_crossvalidation(5, x_train, y_train, sigma[j
                                                     ], gamma[i])
            cv_mse_mat[i,j] = mean_mse
            record_list.append((gamma[i],sigma[j],mean_mse))
            if(mean_mse<min_mse):
                min_mse = mean_mse
                min_gamma_idx = i
                min_sigma_idx = j
            #code used to show progress
            counter = counter +1
            print('loop ' + str(counter) +', '+str((len(gamma)*len(sigma)-
                                                     counter))+' left. '+'MSE:
                                                      '+str(mean_mse))
    #print(mean_mse_list)
    #print(record_list)
    print('min_gamma_idx: %d, min_sigma_idx: %d, min_mse: %f'%(
                                             min_gamma_idx, min_sigma_idx,
                                                        min_mse))
    file=open('data.txt','w')
    file.write(str(record_list));
    file.close()

def part1_3b():
    #plot 3D surface view
    #configuration of plot
    fig = plt.figure()
    #plot configuration
    ax = fig.add_subplot(111, projection='3d')
    ax.set_xlabel('Sigma')
    ax.set_ylabel('Gamma')
    ax.set_zlabel('Cross-validation error')
    plt.xlim(2**7,2**13)
    plt.ylim(2**(-40),2**(-26))

    sigma = []
    gamma = []
    mse = []
    for t in record_list:
        sigma.append(t[1])
        gamma.append(t[0])
        mse.append(t[2])
    ax.plot_trisurf(sigma, gamma, mse, cmap='rainbow')

    plt.show()

    #plot mat view
    plt.matshow(cv_mse_mat)
    plt.xlabel("Sigma")
    plt.ylabel("Gamma")
    plt.show()

def k_fold_crossvalidation(k, data_x, data_y, sigma, gamma):
    #split data into k segments averagely
    kf=KFold(n_splits=k)
    a = kf.split(data_x)
    mse_list = []
    for train_data_index, test_data_index in a:
        x_train = data_x[train_data_index]
        x_test = data_x[test_data_index]
        y_train = data_y[train_data_index]
        y_test = data_y[test_data_index]
        alpha = krr.fit(x_train, sigma, gamma, y_train)
        y_e = krr.estimate(x_train, x_test, sigma, alpha)
        mse = lr.mean_squared_error(y_test,y_e)
        mse_list.append(mse)
    sum = 0
    for i in range(len(mse_list)):
        sum+= mse_list[i]
    mean_mse = sum/len(mse_list)
    return mean_mse
    #alpha = krr.fit(data, data, sigma, gamma, y)
```

```
def part1_3c():
    #best performance parameters
    sigma = 2**10
    gamma = 2**(-31)
    #load data from .mat file
    data = (scio.loadmat('boston.mat'))['boston']
    #spile the train data and test data
    data_train ,data_test = train_test_split(data,test_size=0.33)
    x_train = data_train[:,range(13)]
    y_train = data_train[:,13]
    x_test = data_test[:,range(13)]
    y_test = data_test[:,13]
    #train model
    alpha = krr.fit(x_train, sigma, gamma, y_train)
    #calculate mse
    y_e_train = krr.estimate(x_train, x_train, sigma, alpha)
    mse_train = lr.mean_squared_error(y_train, y_e_train)
    y_e_test = krr.estimate(x_train, x_test, sigma, alpha)
    mse_test = lr.mean_squared_error(y_test, y_e_test)
    #print('MSE on training set is: %f MSE on test set is: %f'%(mse_train,
                                        mse_test))
    return mse_train, mse_test

def part1_3d():
    table = PrettyTable(["Method","MSE train","MSE test"])
    table.padding_width = 1

    #repeat 1.2a 20 times with random split data set
    mse_price_train, mse_price_test, std_train, std_test = part1_2a()
    table.add_row(["Naive Regression",str(mse_price_train)+'    '+str(
                                     std_train),str(mse_price_test)+'
                                        '+str(std_test)])

    #repeat 1.2c 20 times with random split data set
    mse_price_train, mse_price_test, std_train, std_test = part1_2c()
    for i in range(len(mse_price_train)):
        table.add_row(["Linear Regression (attribute"+str(i+1)+")", str(
                                            mse_price_train[i])+'    '+
                                            str(std_train[i]), str(
                                            mse_price_test[i])+'    '+str
                                            (std_test[i])])

    #repeat 1.2d 20 times with random split data set
    mse_price_train, mse_price_test, std_train, std_test = part1_2d()
    table.add_row(["Linear Regression (all attributes)", str(
                                     mse_price_train) +'    '+str(
                                     std_train),str(mse_price_test)+'
                                        '+str(std_test)])
    #repeat 5c 20 times with random split data set
    result_list_train_13c = []
    result_list_test_13c = []
    for i in range(20):
        mse_price_train, mse_price_test = part1_3c()
        result_list_train_13c.append(mse_price_train)
        result_list_test_13c.append(mse_price_test)
    train_std = np.std(result_list_train_13c, ddof = 1)
    test_std = np.std(result_list_test_13c, ddof = 1)
    result_list_train_13c = sum(result_list_train_13c)/20
    result_list_test_13c = sum(result_list_test_13c)/20
    table.add_row(["Kernel Ridge Regression", str(result_list_train_13c) +
                                         '    '+str(train_std), str(
                                         result_list_test_13c) +'    '+str
                                         (test_std)])
    print(table)

if __name__ == '__main__':
    #part1_1_1_a()
    #part1_1_1_b()
    #part1_1_1_c()
    #part1_1_2_a_i()
    #part1_1_2_a_ii()
    #part1_1_2_b()
    #part1_1_2_c()
    #part1_1_2_d()
    #part1_1_3()
    #part1_2a()
    #part1_2c()
    #part1_2d()
    #part1_3a()
    #part1_3b()
    #part1_3c()
    part1_3d()
```