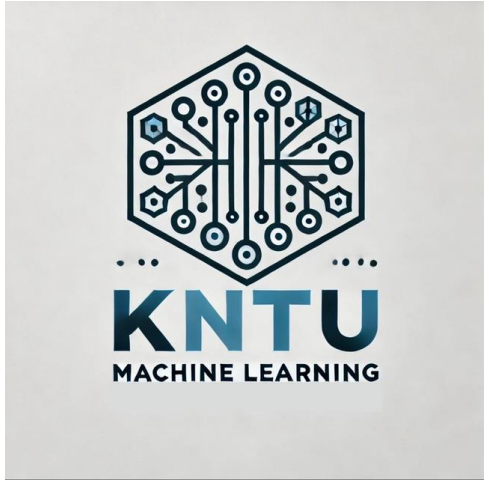
K. N. Toosi
University of Technology

HomeWork2

Prerequisites for Machine Learning

Professor: **Dr.B.Nasersharif**

Provider: **Mehran Tamjidi**



Telegram Group: t.me/ML_403_1

First approach

Question 1

Question 1 (20 marks)

The weight and systolic blood pressure of 26 randomly selected males in the age group 25 to 30 are shown in the following table. Assume that weight and blood pressure are jointly normally distributed. Find a regression line relating systolic blood pressure to weight.

Subject	Weight	Systolic BP	Subject	Weight	Systolic BP
1	165	130	14	172	153
2	167	133	15	159	128
3	180	150	16	168	132
4	155	128	17	174	149
5	212	151	18	183	158
6	175	146	19	215	150
7	190	150	20	195	163
8	210	140	21	180	156
9	200	148	22	143	124
10	149	125	23	240	170
11	158	133	24	235	165
12	169	135	25	192	160
13	170	150	26	187	159

What is the linear regression?

Y_i : first feature
X_i : Second feature

We want to find W, the parameter that relates Y and X → $\boldsymbol{Y_i = W\, X_i}$

Which means that we can find Y given W and X

We are trying to find the most probable estimation of y

$$p(y_1, y_2 \ldots y_n \mid x_1, x_2, x_3, \ldots x_n, W)$$

To find that we will use maximum likelihood estimation …

Maximum likelihood estimation of w

We can convert the **joint probability** into the **product of individual iid probabilities.**

$$P(x_1, x_2, \ldots, x_n|\theta) = \prod_{i=1}^{n} P(x_i|\theta)$$

But under certain assumptions :

1. **Independence Assumption**:
   - The data points $x_1, x_2, \ldots, x_n$ are **independent**.
2. **Identical Distribution (i.i.d.)**:
   - All the data points $x_1, x_2, \ldots, x_n$ are drawn from the **same distribution**.

So instead of maximizing $p(y_1, y_2 \ldots y_n | X_1, X_2, X_3, \ldots X_n, W)$ We maximize $\prod_{i=1}^{n} p(y_i|w, x_i)$

So we will find that:

For what w is

$$\prod_{i=1}^{n} p(y_i|w, x_i) \text{ maximized?}$$

For what w is

$$\prod_{i=1}^{n} \exp\left(-\frac{1}{2}\left(\frac{y_i - wx_i}{\sigma}\right)^2\right) \text{ maximized?}$$

For what w is

$$\sum_{i=1}^{n} -\frac{1}{2}\left(\frac{y_i - wx_i}{\sigma}\right)^2 \text{ maximized?}$$

For what w is

$$\sum_{i=1}^{n} \left(y_i - wx_i\right)^2 \text{ minimized?}$$

$$\sum_i y_i^2 - \left(2\sum x_i y_i\right)w + \left(\sum x_i^2\right)w^2 \xrightarrow{W = -\frac{b}{2a}} w = \frac{\sum x_i y_i}{\sum x_i^2}$$

Now lets back to the Question 1:

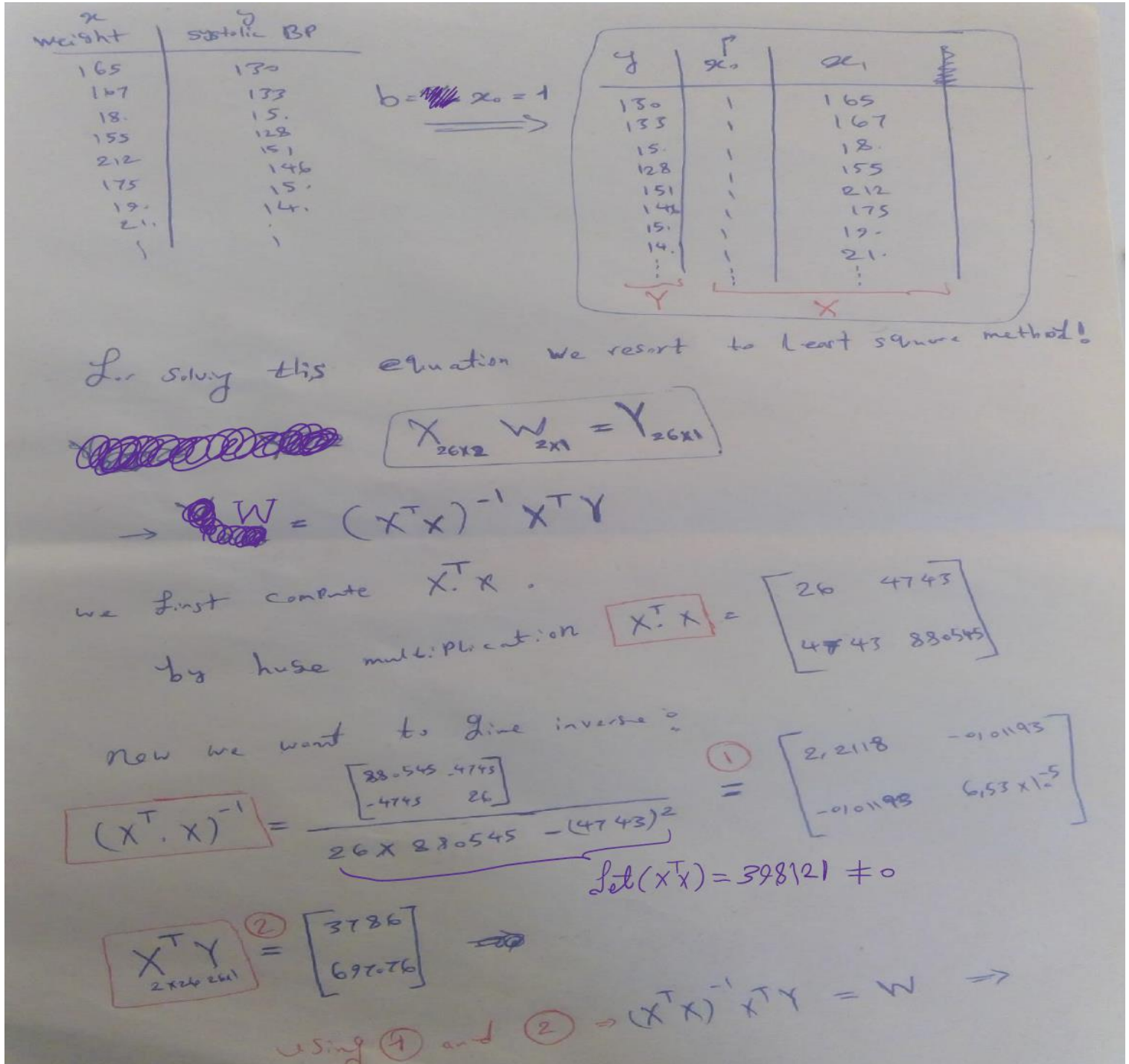
$$w = \frac{\sum x_i y_i}{\sum x_i^{\,2}}$$

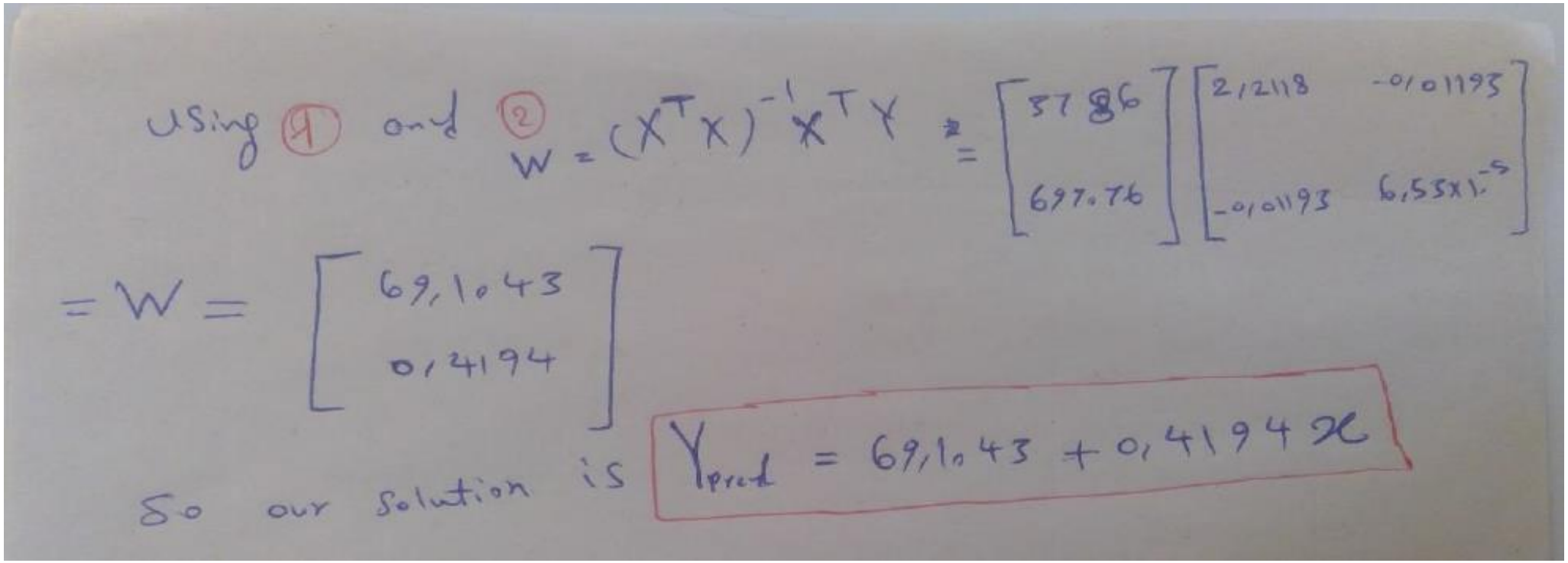
$$w = \frac{(165 \cdot 130) + (167 \cdot 133) + \cdots + (187 \cdot 159)}{130^2 + 133^2 + \cdots + 159^2} \approx 0.23$$

Second approach

To answer this question because the features are matrix, we resort to **least square method**!

$$W = (X^T X)^{-1} X^T Y$$

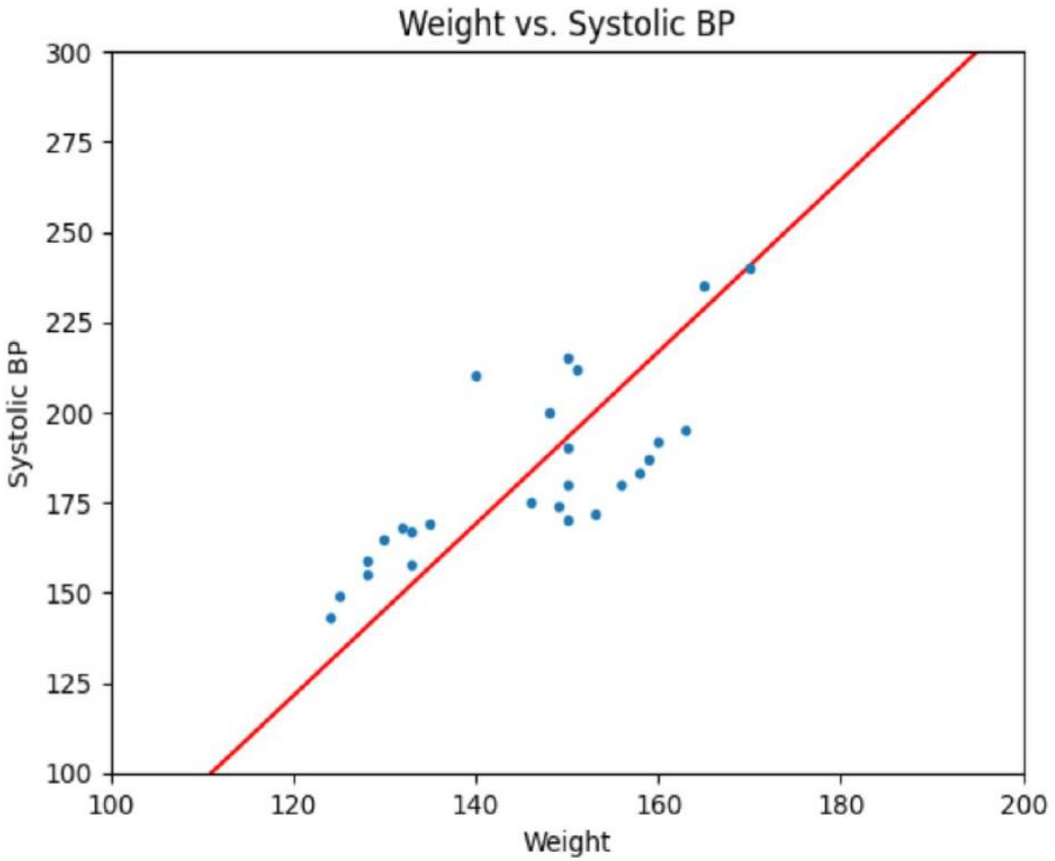




This result shows that Y_pred is a line (linear regression with one variable) that can be used for prediction.

Now we validate the result with Numpy

```
X=np.vstack((np.ones(len(weight)), np.array(weight))).T
Y=np.array(systolic_bp).reshape(26, 1)
W,_,_,_ = np.linalg.lstsq(X, Y, rcond=None)
print('shape = ',X.shape )
print('rank = ',np.linalg.matrix_rank(X.T@X))
print(W)
y=W[0]+W[1]*np.linspace(0,300,100)
plt.plot(y, np.linspace(0,300,100),'-',c='r')
plt.plot( systolic_bp,weight ,'.')
plt.xlabel("Weight")
plt.ylabel("Systolic BP")
plt.xlim(100,200)
plt.ylim(100,300)
plt.title("Weight vs. Systolic BP")
plt.show()
```




```
shape =  (26, 2)
rank =  2
[[69.10437279]
 [ 0.4194152 ]]
```

→ This validate our answer

Question 2

Question 2 (30 marks)

A study was performed on the wear of a bearing y and its relationship with x1 oil viscosity and x2 load. The following data were obtained.

(a) Fit a multivariate linear regression model to these data. (10 Marks)

(b) Use the model to predict wear when x_1 = 25 and x_2 =1000. (5 Marks)

(c) Fit a multivariate linear regression model with an interaction term (x_1x_2) to these data. (15 Marks)

y	x_1	x_2
293	1.6	851
230	15.5	816
172	22.0	1058
91	43.0	1201
113	33.0	1357
125	40.0	1115

Part A :

To find the solution we will repeat all we did for the first example! Except the fact that, now we have three columns for our X feature matrix.

Our data is row major so we place our data in the row of each matrix!!

	y	x1	x2
1	293	1.6	851
2	230	15.5	816
3	172	22.0	1058
4	91	43.0	1201
5	113	33.0	1357
6	125	40.0	1115

$$\begin{bmatrix} 293 & 1.6 & 851 \\ 230 & 15.5 & 816 \\ 172 & 22.0 & 1058 \\ 91 & 43.0 & 1201 \\ 113 & 33.0 & 1357 \\ 125 & 40.0 & 1115 \end{bmatrix}$$

$$\begin{bmatrix} 293 & 1.6 & 851 \\ 230 & 15.5 & 816 \\ 172 & 22.0 & 1058 \\ 91 & 43.0 & 1201 \\ 113 & 33.0 & 1357 \\ 125 & 40.0 & 1115 \end{bmatrix}$$

↓

y	x_0	x_1	x_2
295	1	116	850
230	1	15,5	816
172	1	22,5	1058
91	1	43,0	1201
113	1	33	1357
125	1	4.	1115

Multi-Variate Least Square

$$W = (X^T X)^{-1} X^T Y$$

$X_{6\times3}\ W_{3\times1} = Y_{6\times1}$

$W = (X^T X)^{-1} X Y$

$\rightarrow X^T X = \begin{bmatrix} 6 & 155 & 639,8 \\ 155,1 & 5264 & 17830,6 \\ 639,8 & 17830,9 & 70364,6 \end{bmatrix}_{3\times3}$

$(X^T X)^{-1} \Longrightarrow$ checking non-singularity

$\det(X^T X) = 6(52647,8 \times 70364,6 - 17830,6 \times 17830,1)$

$-155,1(155,1 \times 70364,6 - 17830,6 \times 6398)$

$+ 6398(155,1 \times 17830,6 - 6 \times 6398)$

$= \det(X^T X) = 61098069 \neq 0$

Since the matrix is full rank:

$$(X^TX)^{-1} = \begin{bmatrix} 8,595 & 0,08095 & -0,0098667 \\ 0,08095 & 0,0021024 & 0,0001268 \\ -0,009887 & 0,0001268 & 0,0000125 \end{bmatrix}$$

Then we compute:

$$X^TY = \begin{bmatrix} 1024 \\ 20459,8 \\ 1021006 \end{bmatrix}$$

$$W = (X^TX)^{-1}(X^TY) = \begin{bmatrix} 8,595 & 0,08095 & 0,0098667 \\ 0,08095 & 0,0021024 & 0,0001268 \\ -0,009887 & 0,0001268 & 0,0000125 \end{bmatrix} \begin{bmatrix} 1024 \\ 20459,8 \\ 1021006 \end{bmatrix}$$

$$W = \begin{bmatrix} 383,801 \\ -3,680 \\ -0,11168 \end{bmatrix}$$

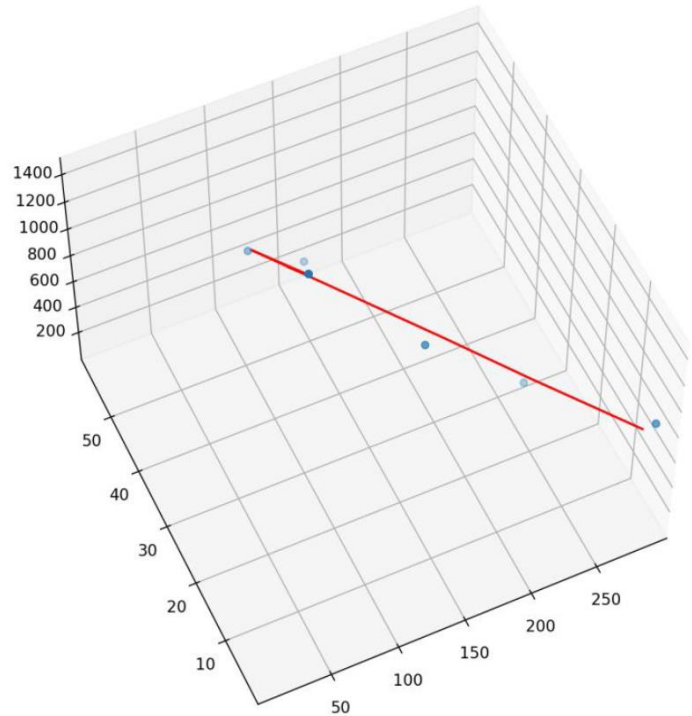
$$Y_{pred} = 383,801 - 3,680\,X_1 - 0,11168\,X_2$$

To verify our answer we use the Matplotlib and Numpy again.

```
Y = table[:, 0].reshape(6, 1)
X = np.vstack((np.ones((1, 6)), table[:, 1], table[:, 2])).T

print('rank = ', np.linalg.matrix_rank(X))

W, _, _, _ = np.linalg.lstsq(X, Y, rcond=None)
print('W=', W)

x1, x2 = np.meshgrid(range(60), range(1500))
Z = W[0] + W[1] * x1 + W[2] * x2

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(Y, table[:, 1], table[:, 2], label='Data Points')  # x, y, z
ax.plot_surface(Z, x1, x2, alpha=0.5, color='r', label='Regression Plane')

ax.set_xlim(0, 300)
ax.set_ylim(0, 60)
ax.set_zlim(0, 1500)
ax.set_xlabel('Y')
ax.set_ylabel('X1')
ax.set_zlabel('X2')
ax.view_init(elev=20, azim=30)  # for better visualization
plt.show()
```

```
rank =  3
W= [[ 3.83801029e+02]
 [-3.63808834e+00]
 [-1.11681568e-01]]
```

From the top, it's looks like a line!



Part B :

Now we can simply predict y for an arbitrary X1 and X2 pair.

for $x_1 = 25$ & $x_2 = 1000$ $\quad y = ?$

$$y = 385,801 - 3,168.2 x_1 - 0,11168 x_2$$

by substitution → $y = 181,17$

Part B :

y	x_0	x_1	x_2	x_1x_2
239	1	1,6	850	136116
230	1	1515	876	12648
172	1	2210	1058	23276
91	1	4310	1201	51643
113	1	3310	1357	44781
125	1	40	1115	44600

Y $\quad$ X

$$X_{6\times4} W_{4\times1} = Y_{6\times1}$$

$$W = (X^T X)^{-1} X Y$$

$$X^T X = \begin{bmatrix} 6 & 155,1 & 6378 & 1783096 \\ 1591 & 52648 & 1785096 & 6192716156 \\ 6378 & 1783096 & 7036496 & 2086.558 \\ 1783096 & 6192716156 & 2086.5558 & 7365.95440 \end{bmatrix}_{4\times4} \longrightarrow \det(X^T X) \neq 0$$

$$(X^T X)^{-1} = \begin{bmatrix} 69,816 & -2,37 & -0,0773 & 0,00249 \\ -2,87 & 0,1006 & 0,00258 & -0,0000102 \\ -0,0773b & 0,00258 & 0,0000867 & -0,0000275 \\ 0,00249 & -0,0000102 & -0,0000275 & 0,0000001,7 \end{bmatrix}$$

$$X^T Y = \begin{bmatrix} 1024,0 \\ 2045,8 \\ 1021,06 \\ 212646 \times 1,7 \end{bmatrix}$$

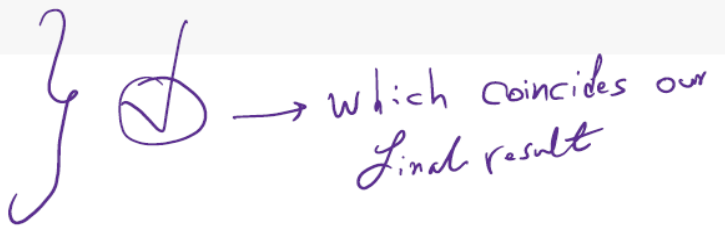
$$W = (X^T X^{-1})\, X^T Y = \begin{bmatrix} 483,96 \\ -7,656 \\ -0,22211 \\ 0,004,89 \end{bmatrix}$$

$$Y = 483,96 - 7,656 X_1 - 0,22211 X_2 + 0,004,89 X_1 X_2$$

We can also verify our result by python

```
Y = table[:, 0].reshape(6, 1)
X = np.vstack((np.ones((1, 6)), table[:, 1], table[:, 2],
np.multiply(table[:, 1], table[:, 2]))).T

W, _, _, _ = np.linalg.lstsq(X, Y, rcond=None)
X1=X[:,1]
X2=X[:,2]
X3=X[:,3]
print('W=',W)

Z = W[0] + W[1] * X1 + W[2] * X2 + W[3] * X3
```

```
W= [[ 4.83966183e+02]
 [-7.65603415e+00]
 [-2.22115334e-01]
 [ 4.08697965e-03]]
```



Question 3

Question 3 (30 Marks)

In a regression problem, we predict O based on one n-dimensional vector (sample) X using the following relation:

$$o = w_0 + w_1 x_1 + w_1 x_1^3 + w_2 x_2 + w_2 x_2^3 + \ldots + w_n x_n + w_n x_n^3$$

(a) Suppose that we have m samples in our dataset. Write the Matrix form of O=ZW with complete displays of O, Z, and W matrices (Z is constructed based on X) (20 Marks)

(b) If we want to obtain W using the gradient-descent method, write the gradient descent relation for this problem such as below (10 marks)

$$w_i \leftarrow w_i + \ldots \text{ for } 1 \leq i \leq n.$$

The update relation is not required for the bias term (w_0)

Part A

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & x_{1n} \\ 1 & x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m1} & x_{m2} & \dots & x_{mn} \end{bmatrix}$$

Here is matrix X

$$W = [w_0 \quad w_1 \quad w_2 \quad \dots \quad w_n]$$

And our weight vector

$$O = w_0 + \underbrace{w_1(x_1 + x_1^3)}_{w_1 x_1 + w_1 x_1^3} + \underbrace{w_2(x_2 + x_2^3)}_{w_2 x_2 + w_2 x_2^3} + \dots + \underbrace{w_n x_n + w_n x_n^3}_{w_n(x_n + x_n^3)}$$

$\Rightarrow$ This can be written as: $O = ZW$

$$O = \left[\underbrace{1}_{\varphi_0^n}, \underbrace{x_1 + x_1^3}_{\varphi_1^n}, \underbrace{x_2 + x_2^3}_{\varphi_2^n}, \dots, \underbrace{x_n + x_n^3}_{\varphi_n^n} \right] \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

where φ_i^n is n dimentional vector

M.Tamjidi

on the other hand it can be written as:

$$\begin{cases} \varphi_{in}(x) = x_i + x_i^3 & i \neq 0 \\ \varphi_{on}(x) = 1 \end{cases}$$

$$O = \left[\varphi_{0n}(x), \varphi_{1n}(x), \varphi_{2n}(x), \cdots + \varphi_{nn}(x) \right] \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

where $\varphi_{in} = \begin{bmatrix} \varphi_{in}(x)^{[1]} \\ \varphi_{in}(x)^{[2]} \\ \vdots \\ \varphi_{in}(x)^{[n]} \end{bmatrix}$ for $i \neq 0 = \begin{bmatrix} x_{i_1} + x_{i_1}^5 \\ x_{i_2} + x_{i_2}^3 \\ \vdots \\ x_{in} + x_{in}^3 \end{bmatrix}$ & for $i = 0 = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$

Part A

so we can see the matrix Z :

$$Z = \begin{bmatrix} 1 & x_{11} + x_{11}^3 & x_{12} + x_{12}^3 & \dots & x_{1n} + x_{1n}^3 \\ 1 & x_{21} + x_{21}^3 & x_{22} + x_{22}^3 & \dots & x_{2n} + x_{2n}^3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m1} + x_{m1}^3 & x_{m2} + x_{m2}^3 & \dots & x_{mn} + x_{mn}^3 \end{bmatrix}$$

$$O^T = \begin{bmatrix} w_0 z_{10} + w_1 z_{11} + \cdots + w_n z_{1n} \\ w_0 z_{20} + w_1 z_{21} + \cdots + w_n z_{2n} \\ \vdots \\ w_0 z_{m0} + w_1 z_{m1} + \cdots + w_n z_{mn} \end{bmatrix}$$

Part B

First Step: we have to compute cost function :

$$\mathcal{J}_{x,y}(W) = \sum_{i=0}^{m} (y_i - \hat{y}_i)^2$$

$$\mathcal{J}_{x,y}(W) = \sum_{i=0}^{m} (y_i - \hat{y}_i)^2 = \sum_{i=0}^{m} \left(y_i - \sum_{j=0}^{n} \phi_j(x) W_j \right)^2$$

Part B

Second Step: getting the derivative of the cost function with respect to the weights:

$$\mathcal{J}_{x,y}(W) = \sum_{i=0}^{m} (y_i - \hat{y}_i)^2$$

$$J(W) = \sum_{i=1}^{m} (y_i - \hat{y}_i)^2 \quad \longleftarrow \quad \hat{y}_i = \sum_{j=1}^{n} w_j \phi_j(x^i)$$

$$\frac{\partial J(W)}{\partial w_j} = \frac{\partial}{\partial w_j} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2 = 2 \sum_{i=1}^{m} (y_i - \hat{y}_i) \left(-\frac{\partial}{\partial w_j} \sum_{j=1}^{n} w_j \phi_j(x^i) \right)$$

$$= 2 \sum_{i=1}^{m} (y_i - \hat{y}_i) \phi_j(x^i)$$

Part B

third Step: getting the derivative of the cost function with respect to the weights:

$$\frac{\partial J(W)}{\partial w_j} = 2\sum_{i=1}^{m}(y_i - \hat{y}_i)\phi_j(x^i)$$

$$\frac{\partial J(W)}{\partial w_j} = 2\sum_{i=1}^{m}\left(y_i - \sum_{j=1}^{n} w_j\phi_j(x_i)\right)\phi_j(x_i)$$

$$W_j^{(i+1)} = W_j^{(i)} - \eta\frac{\partial \mathcal{J}(W)}{\partial W_j}$$

$$w_j = w_j - \lambda\left[2\sum_{i=1}^{m}\left(y_i - \sum_{j=1}^{n} w_j\phi_j(x^i)\right)\phi_j(x^i)\right]$$

And then we repeat that for all of the examples for i = 1, 2, ..., m and j= 1, 2, ..., n.

Part B

Which m is number of samples and n in number of terms in vector O.

So, we write down all of this steps together.

Here we provide a sudo code for all of this (stochastic gradient descent):

```
for i in range(0 ,m):
    Yforward=phi[i]*W
    term=Y[i]-Yforward
    for j in range(1, n):   # we don't update the bias term as question wanted!
        dJ[j]= 2*( X[j]+X[j]^3)*term  #here we calculated gradients of cost function J with respect of W[j]
    For j in range(1,n):
        W[j]=W[j]-lambd*dJ[j] #updating parameters
```

But we can also customize this sudo code by vectorization for enhancing the speed using the NumPy.
Note that all of that is only one epoch and we can repeat it for several epoch that we want to minimize our loss.

Question 4

Question 4 (70 marks) (implementation)

In this question you will implement linear basis function regression with polynomial and Gaussian bases.

Start by downloading the code and dataset from the website. The dataset is the AutoMPG dataset from the UCI repository (https://archive.ics.uci.edu/dataset/9/auto+mpg). The task is to predict fuel efficiency (miles per gallon) from 7 features describing a car.

Functions are provided for loading the data[1], and normalizing the features and targets to have 0 mean and unit variance.

```
[t,X] = loadData();
X_n = normalizeData(X);
t = normalizeData(t);
```

For the following, use these normalized features X_n and targets.

You may also find the provided function designMatrix.m useful.

Polynomial basis functions

Implement linear basis function regression with polynomial basis functions. Perform the following experiments:

1. Using the first 100 points as training data, and the remainder as testing data, fit a polynomial basis function regression for degree 1 to degree 10 polynomials. Do not use any regularization. Plot training error and test error (in RMS error) versus polynomial degree. **Put this plot, along with a brief comment on what you see, in your report.** For the basis functions:
   (a) Include the bias function as always.
   (b) For each input variable x_i, $i = 1, \ldots, 7$, and for each power $k = 1, \ldots, degree$, there is a basis function $\Phi_{i,k}$ that returns x_i^k. So you should have $7 \cdot degree$ basis functions, plus the bias function, and hence that many weights. In words, treat each input variable as separate single variable and then follow the treatment in the book and use the single-variable powers up to the degree. The degree (maximum power) should run from 1 to 10.
2. It is difficult to visualize the results of high-dimensional regression. Instead, only use one of the features (use X_n(:,3)) and again perform polynomial regression. Produce plots of the training data points, learned polynomial, and test data points. The code visualizes 1d.m may be useful. **Put 2 or 3 of these plots, for interesting (low-order, high-order) results, in your report. Include brief comments.**

[1] Note that loadData reorders the datapoints using a fixed permutation. Use this fixed permutation for the questions in this assignment. If you are interested in what happens in "reality", try using a random permutation afterwards. Results will not always be as clean as you will get with the fixed permutation provided.

Step1 : Loading data …

Part A

```
def loadData():

    column_names = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
'acceleration', 'model_year', 'origin', 'car\'name']
    data = pd.read_csv(r'                              \auto-mpg.data',
delim_whitespace=True, header=None, names=column_names, na_values="?")

    data = data.dropna()

    t = data['mpg'].values
    X = data[['cylinders', 'displacement', 'horsepower', 'weight',
'acceleration', 'model_year', 'origin']].values.astype(float)

    rp = loadmat(r'                              \rp.mat')['rp'].flatten() - 1

    X = X[rp, :]
    t = t[rp]

    return t, X
```

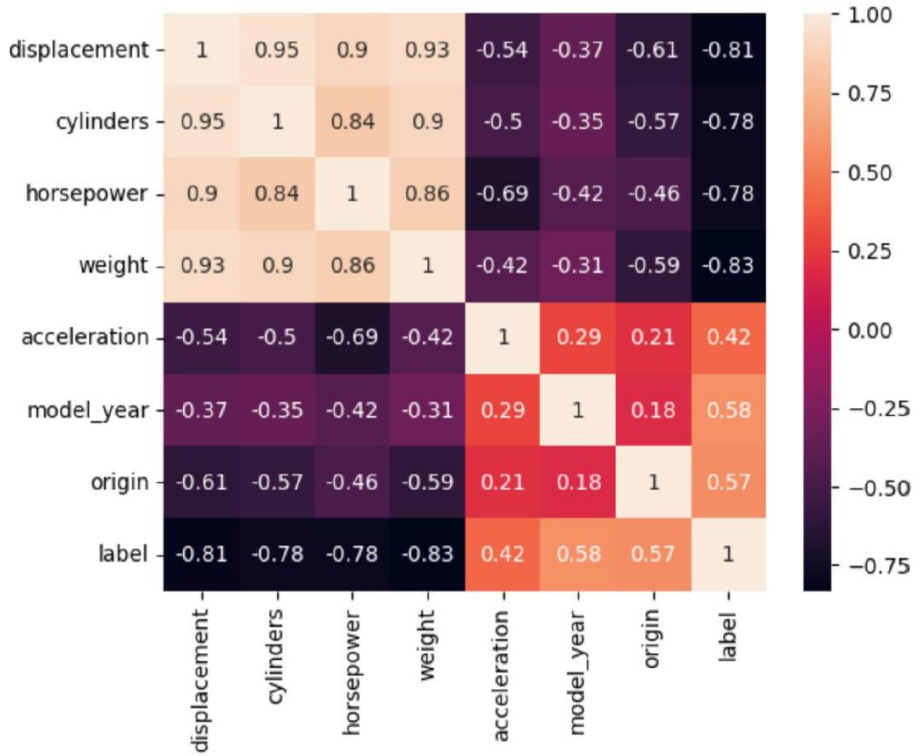
Step2 : Normalizing dataset

```python
def normalizeData(X):

    mu = np.mean(X, axis=0)
    sig = np.std(X, axis=0, ddof=0)

    X_n = (X - mu) / sig

    return X_n
```

checking feature relations

Part A

```python
import seaborn as sns

correlation_matrix = X_normalized.corr().round(2)
# annot = True to print the values inside the square
sns.heatmap(data=correlation_matrix, annot=True)
```



Step2 : implementing multivariate regression

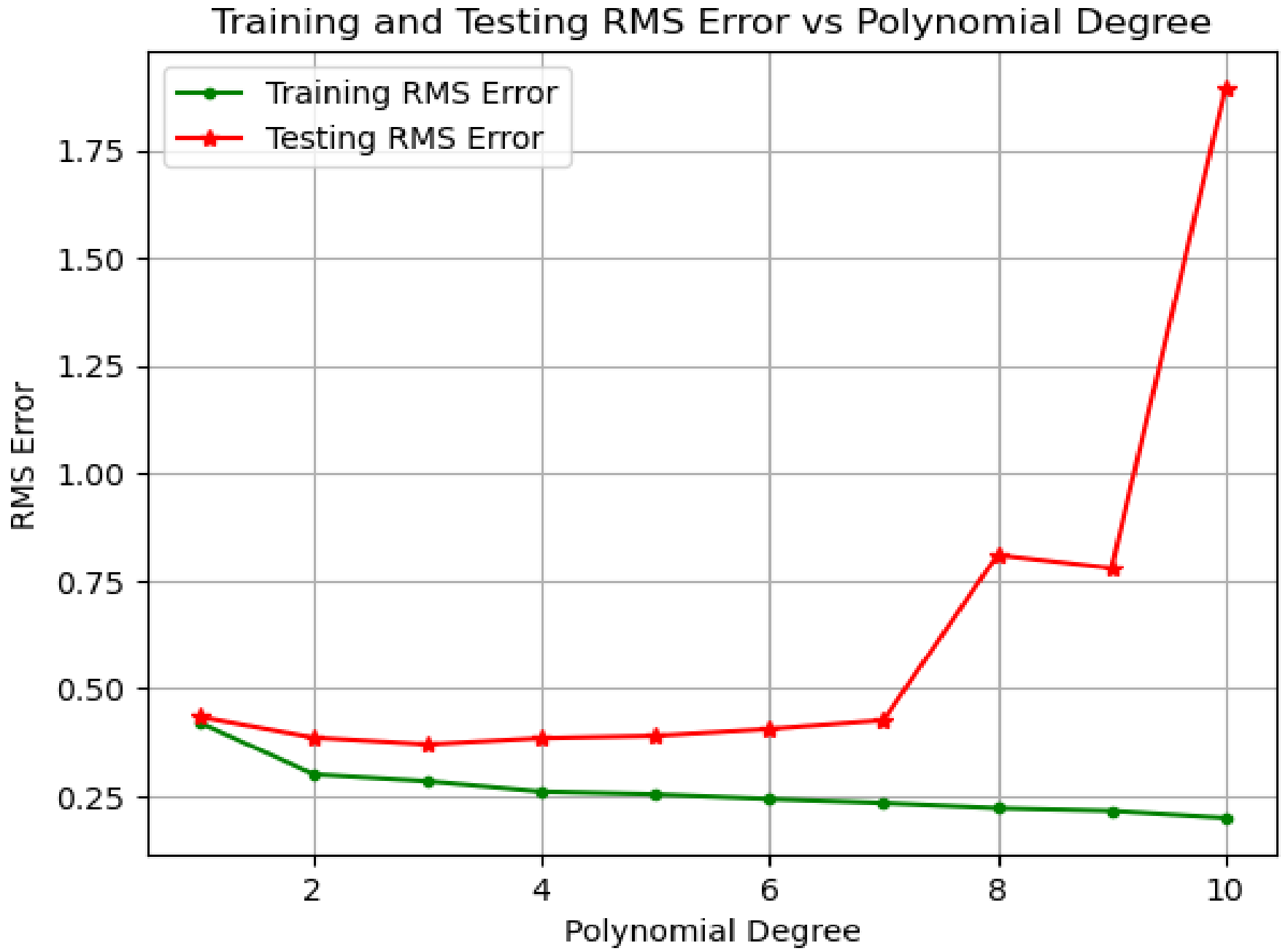
Part A

```python
plt.figure(figsize=(15, 10))

degrees = [3, 7, 10]

for d in degrees:

    basis_train = np.ones((X_train_single.shape[0], 1))
    for i in range(1, d + 1):
        basis_train = np.column_stack((basis_train, X_train_single ** i))

    w = np.linalg.pinv(basis_train.T @ basis_train) @ basis_train.T @ t_train

    plt.subplot(2, 2, degrees.index(d) + 1)
    plt.scatter(X_train_single, t_train, marker='o', color='blue', label='Training Data')
    plt.scatter(X_test_single, t_test, color='red',marker='.', label='Test Data')

    X_range = np.linspace(X_train_single.min(), X_train_single.max(), 100).reshape(-1, 1)

    B = np.ones((X_range.shape[0], 1))
    for i in range(1, d + 1):
        B = np.column_stack((B, X_range ** i))

    y_range_pred = B @ w
    plt.plot(X_range, y_range_pred, color='green', label=f'Degree {d}')

    plt.xlabel('Single Feature')
    plt.ylabel('Label')
    plt.title(f'Polynomial Degree {d}')
    plt.legend()
    plt.grid(True)

plt.tight_layout()
plt.show()
```

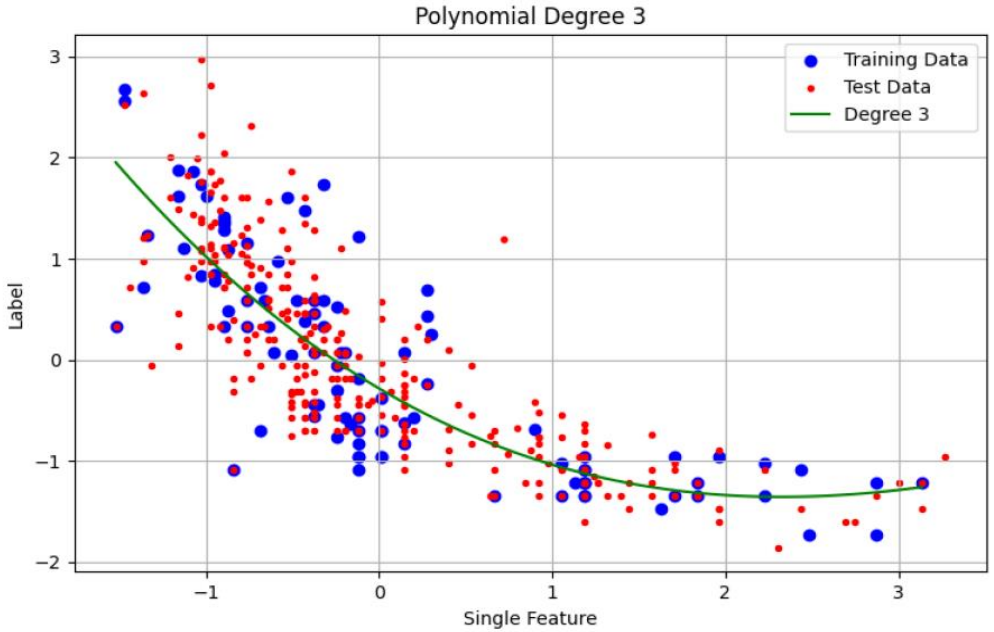
Ridge regression

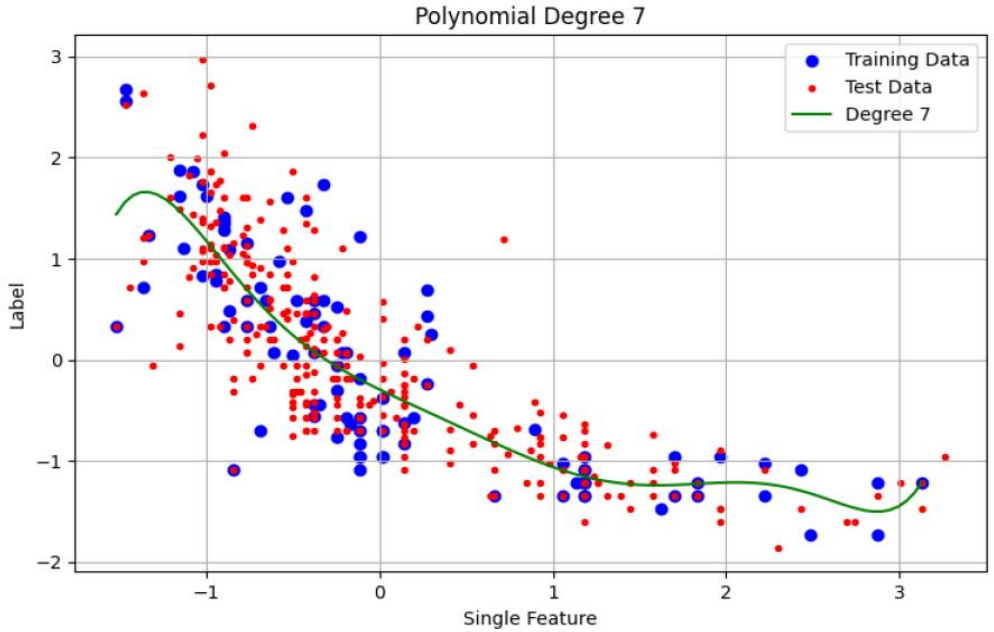
$$\hat{\beta}_R = \left(\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}\right)^{-1} \mathbf{X}^\top \mathbf{y}$$

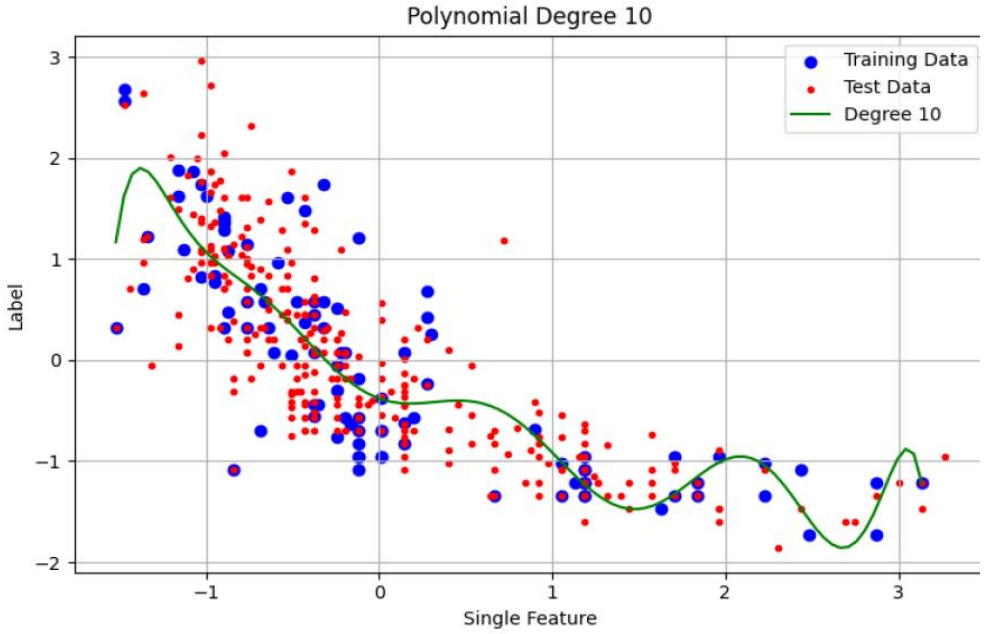
Part A

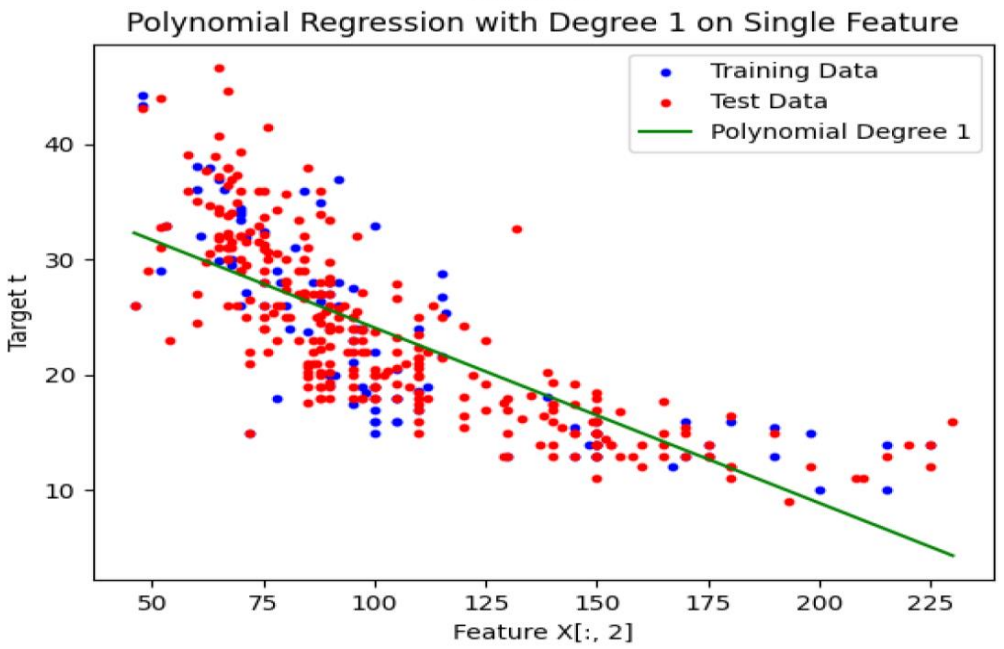

Training and Testing RMS Error vs Polynomial Degree
Training RMS Error
Testing RMS Error
RMS Error
1.75
1.50
1.25
1.00
0.75
0.50
0.25
2
4
6
8
10
Polynomial Degree

Part A









M.Tamjidi

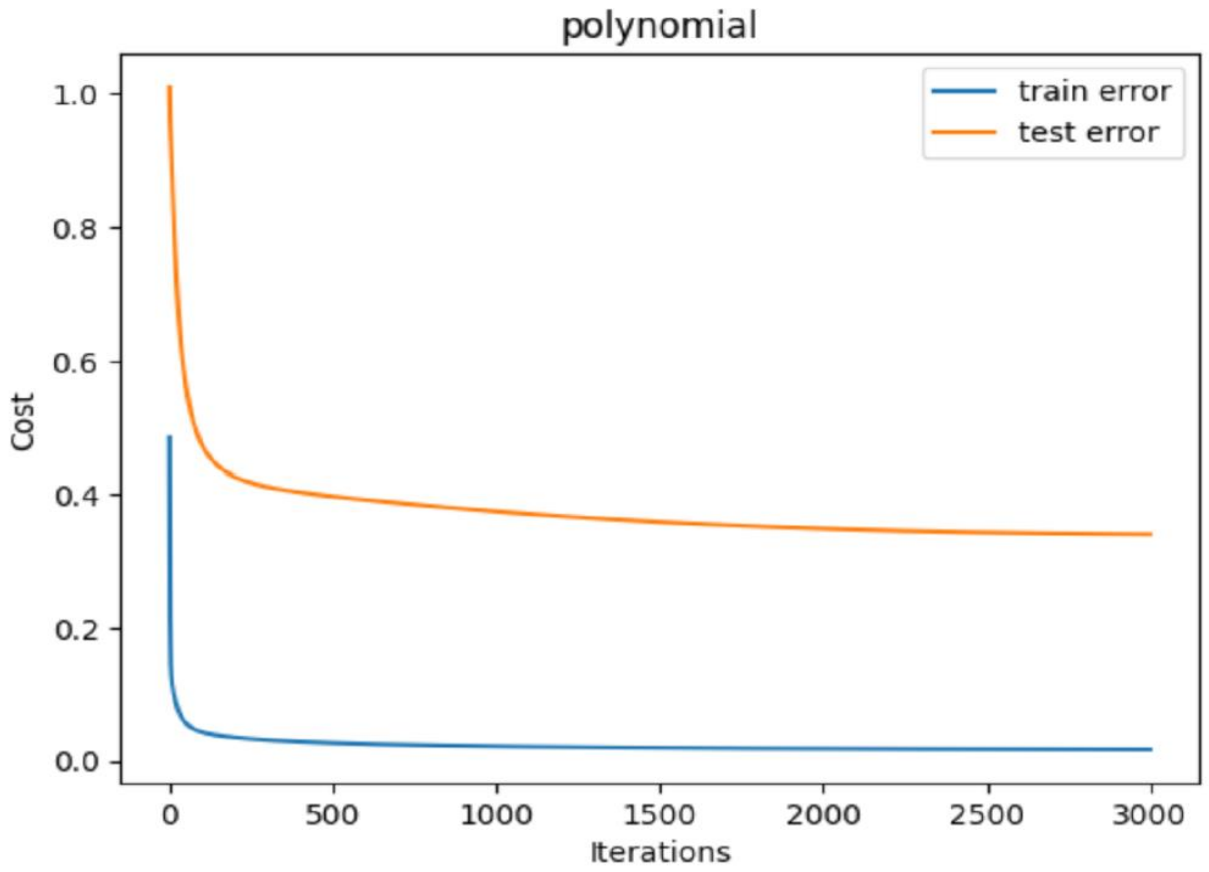
We also can use gradient based methods :

Part A

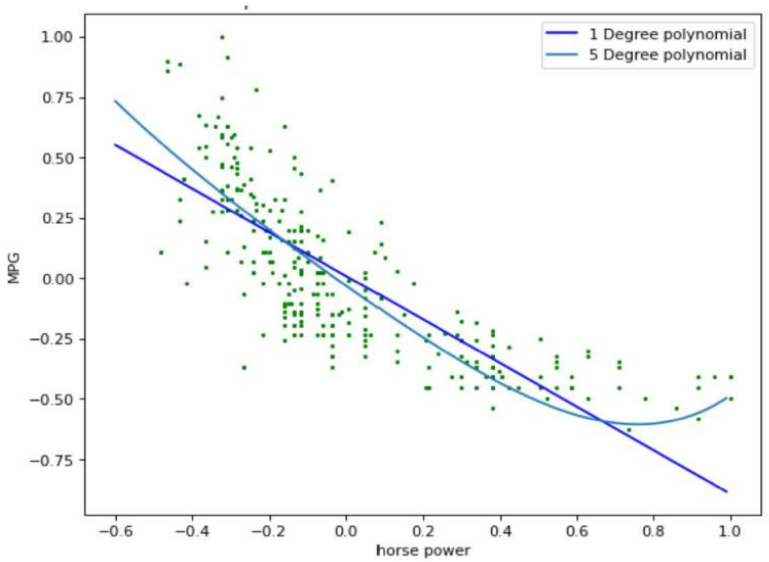
```python
class Regression:
  J=0 #cost function
  def __init__(self, X, Y, W):
    self.X = X
    self.Y = Y
    self.W = W
  def cost(self):
    m=self.X.shape[0]
    J=(1 / (2 * m)) * np.sum ((np.dot(self.X , self.W) - self.Y) ** 2)
    return(J)

  def batch_GD(self, alpha, num_iters, X_test, Y_test):
    cost_list=[]
    eval_cost_list=[]
    m = self.X.shape[0]
    n = self.X.shape[1]
    grad = np.zeros (n)
    for i in range (0,num_iters+1):
      prediction = np.dot (self.X, self.W)
      J=(1 / (2 * m)) * np.sum ((prediction - self.Y) ** 2)
      MSE,_=self.evaluation(X_test, Y_test)
      if i%200==0:
        print('epoch=%i and the train_cost is= %f and evaluation error is %f'%(i,J, MSE))
      grad = (1 / m) * np.dot (self.X.T, (prediction - self.Y))
      self.W = self.W - alpha * grad
      cost_list.append([i,J])
      eval_cost_list.append([i,MSE])
    return cost_list, eval_cost_list, self.W

  def evaluation(self, X_test, Y_test):
    m = self.X.shape[0]
    Y_prediction = np.dot(X_test, self.W)
    mse= np.mean((Y_test - Y_prediction)**2)
    mae = np.mean(Y_test- Y_prediction)
    return mse, mae
```

Part A

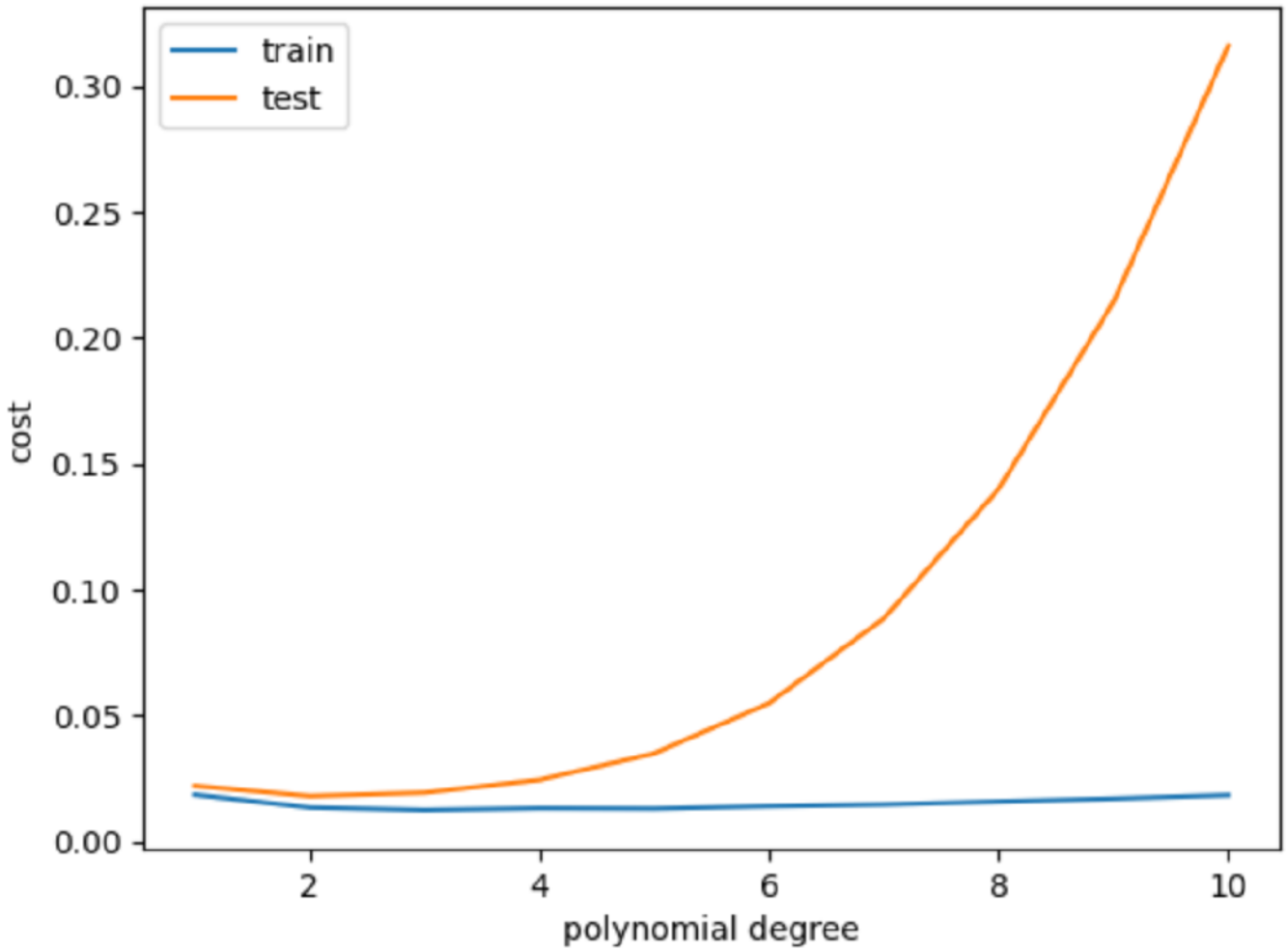
This is polynomial for degree=2



We also can see how well it predict and virtualize it in another way



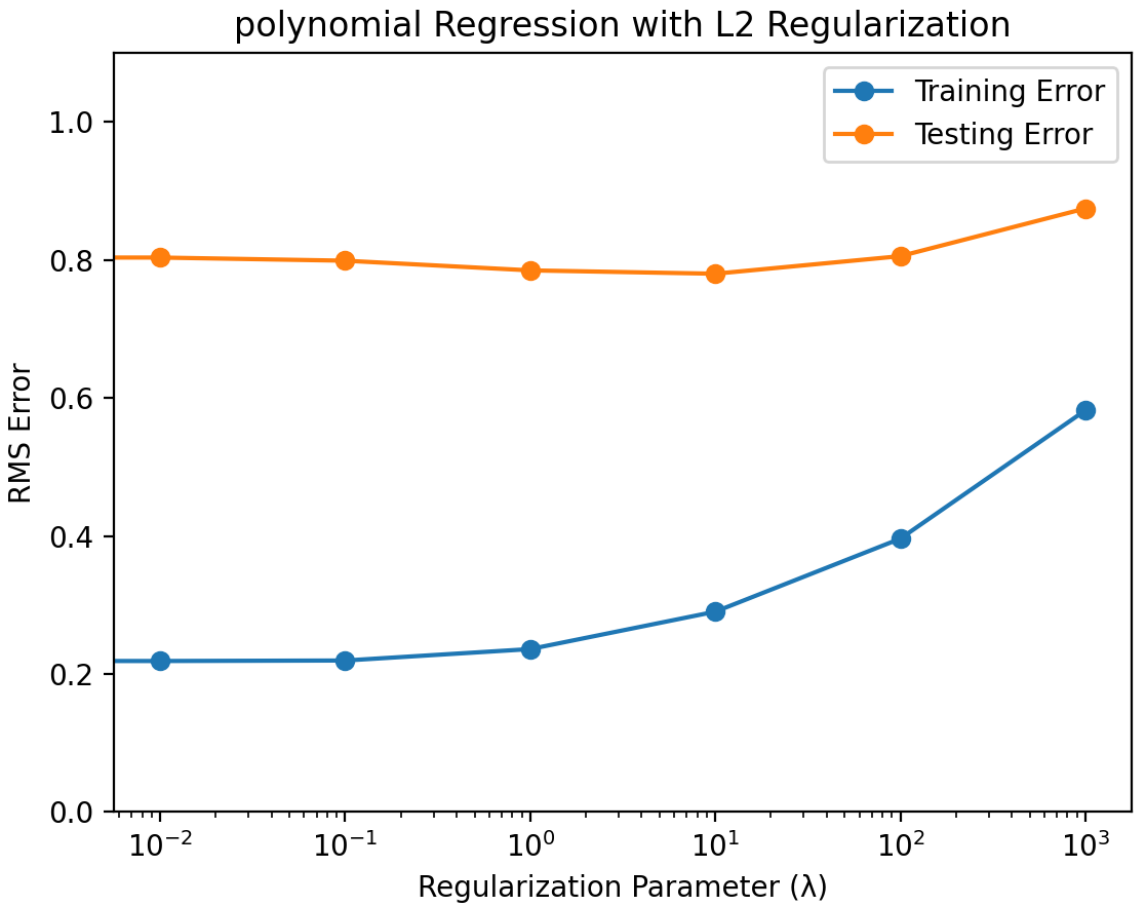
Part A



Which shows that with increasing the polynomails degree, we will getting high variance problem and the model **overfitts**. And cannot predict the dev set well.

Step2 :using regularization

```python
lambdas = [0, 0.01, 0.1, 1, 10, 100, 1000]

train_rms_errors = []
test_rms_errors = []

basis_train = np.ones((X_train_single.shape[0], 1))
basis_test = np.ones((X_test_single.shape[0], 1))

for d in range(1, 9):
    basis_train = np.column_stack((basis_train, X_train_single ** d))
    basis_test = np.column_stack((basis_test, X_test_single ** d))

for lamb in lambdas:

    I = np.eye(basis_train.shape[1])
    w = np.linalg.pinv(basis_train.T @ basis_train + lamb * I) @ basis_train.T @ t_train

    train_prediction = basis_train @ w
    test_prediction = basis_test @ w

    train_rmse = np.sqrt(np.mean((train_prediction - t_train) ** 2))
    test_rmse = np.sqrt(np.mean((test_prediction - t_test) ** 2))

    train_rms_errors.append(train_rmse)
    test_rms_errors.append(test_rmse)

plt.figure(figsize=(15, 10))
plt.plot(lambdas, train_rms_errors, label='Training Error', marker='o', color='blue')
plt.plot(lambdas, test_rms_errors, label='Testing Error', marker='o', color='red')

plt.xlabel('Regularization Parameter (λ)')
plt.ylabel('RMS Error')
plt.title('Train and Test RMS Error vs Regularization (λ) for Degree 8 Polynomial')
plt.legend()
plt.grid(True)
plt.show()
```

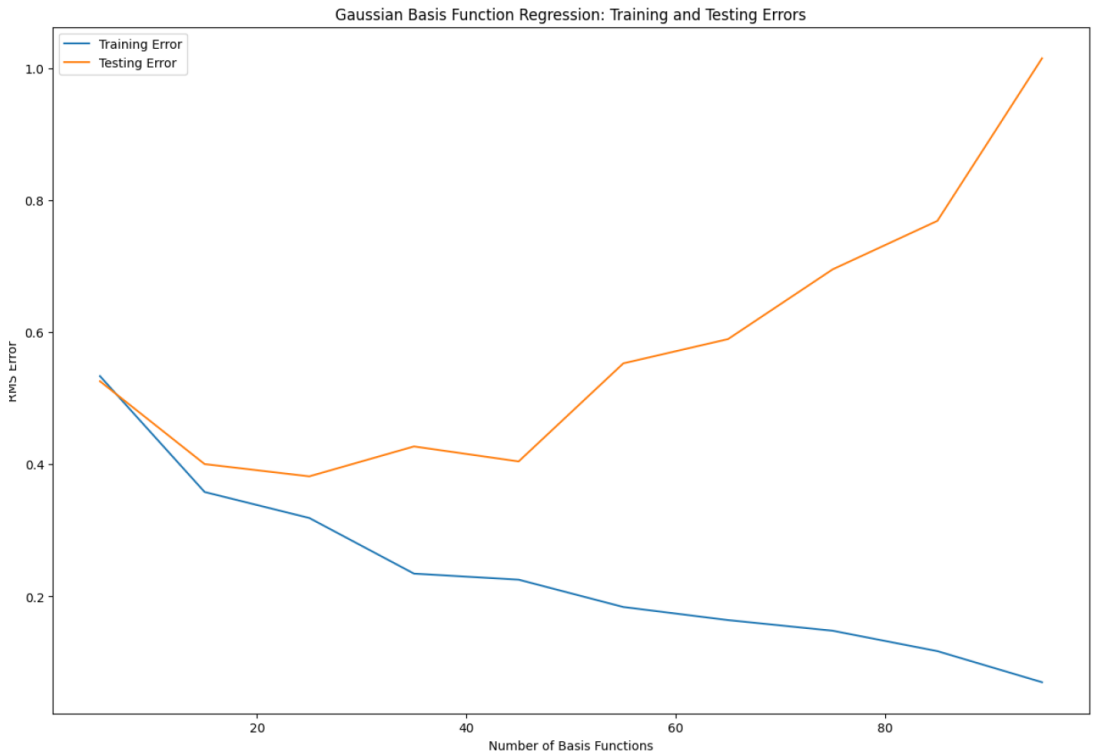


M.Tamjidi

Part B

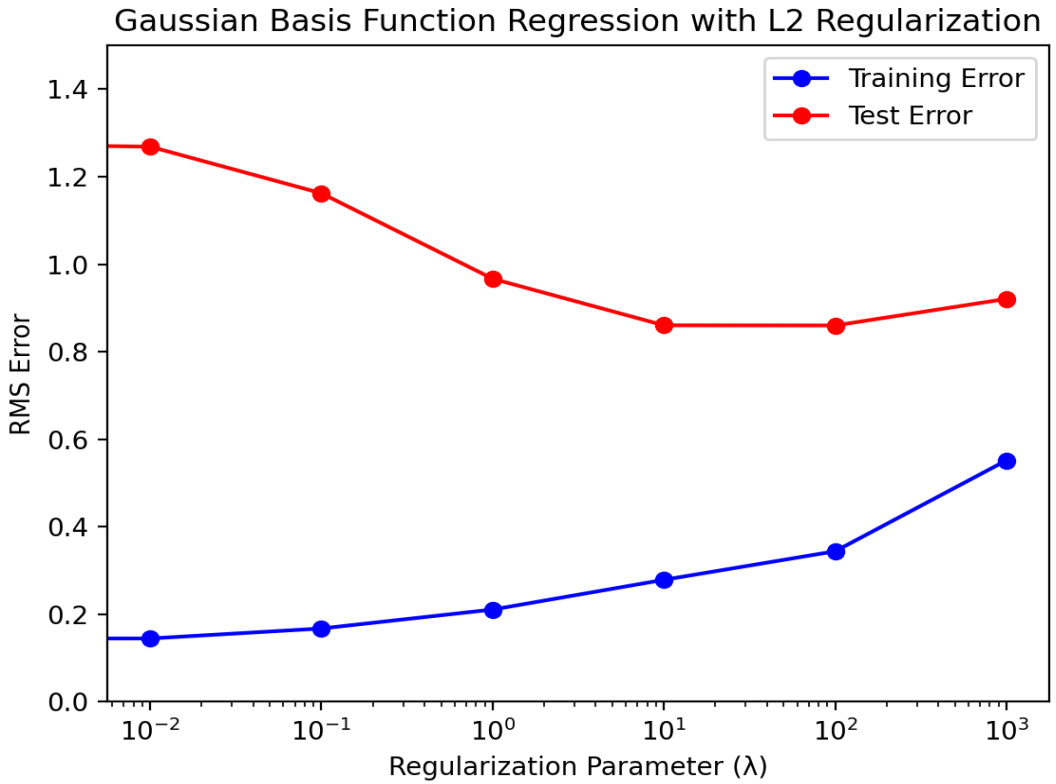
```python
import numpy as np

def dist2(x, c):

    if x.shape[1] != c.shape[1]:
        raise ValueError("Data dimension does not match dimension of centers")

    n2 = np.add.outer(np.sum(x**2, axis=1), np.sum(c**2, axis=1)) - 2 * np.dot(x, c.T)

    n2[n2 < 0] = 0

    return n2
train_errors = []
test_errors = []
s = 2
Range = range(5, 96, 10)

for number in Range:

    random_indices = np.random.permutation(X_train.shape[0])[:number]
    centers = X_train[random_indices]

    dist_train = dist2(X_train, centers)
    dist_test = dist2(X_test, centers)

    Phi_train = np.hstack([np.ones((X_train.shape[0], 1)), np.exp(-dist_train / (2 * s**2))])
    Phi_test = np.hstack([np.ones((X_test.shape[0], 1)), np.exp(-dist_test / (2 * s**2))])

    w = np.linalg.pinv(Phi_train) @ t_train

    t_train_pred = Phi_train @ w
    train_errors.append(np.sqrt(np.mean((t_train_pred - t_train) ** 2)))

    t_test_pred = Phi_test @ w
    test_errors.append(np.sqrt(np.mean((t_test_pred - t_test) ** 2)))

plt.figure(figsize=(15, 10))
plt.plot(Range, train_errors, label="Training Error")
plt.plot(Range, test_errors, label="Testing Error")
plt.xlabel("Number of Basis Functions")
plt.ylabel("RMS Error")
plt.title("Gaussian Basis Function Regression: Training and Testing Errors")
plt.legend()
plt.show()
```

$$K(x, c) = \exp\left(-\frac{\|x - c\|^2}{2\sigma^2}\right)$$



M.Tamjidi

```python
train_errors_reg = []
test_errors_reg = []
Lamb = [0, 0.01, 0.1, 1, 10, 100, 1000]
number = 90
s = 2

random_indices = np.random.permutation(X_train.shape[0])[:number]
centers = X_train[random_indices]

Phi_train = np.hstack([np.ones((X_train.shape[0], 1)), np.exp(-dist_train / (2 * s**2))])
Phi_test = np.hstack([np.ones((X_test.shape[0], 1)), np.exp(-dist_test / (2 * s**2))])

for lambdas in Lamb:

    I = np.eye(Phi_train.shape[1])
    w = np.linalg.inv(Phi_train.T @ Phi_train + lambdas * I) @ Phi_train.T @ t_train

    t_train_prediction = Phi_train @ w
    train_errors_reg.append(np.sqrt(np.mean((t_train_prediction - t_train) ** 2)))


    t_test_prediction = Phi_test @ w
    test_errors_reg.append(np.sqrt(np.mean((t_test_prediction - t_test) ** 2)))



plt.figure(figsize=(15, 10))
plt.plot(Lamb, train_errors_reg, label="Training Error")
plt.plot(Lamb, test_errors_reg, label="Testing Error")
plt.xscale('log')
plt.ylim(0, 1)
plt.xlabel("Regularization Parameter (λ)")
plt.ylabel("RMS Error")
plt.title("Gaussian Basis Function Regression with L2 Regularization")
plt.legend()
plt.show()
```



M.Tamjidi

Question 5

Question 5 (20 marks) (implementation)

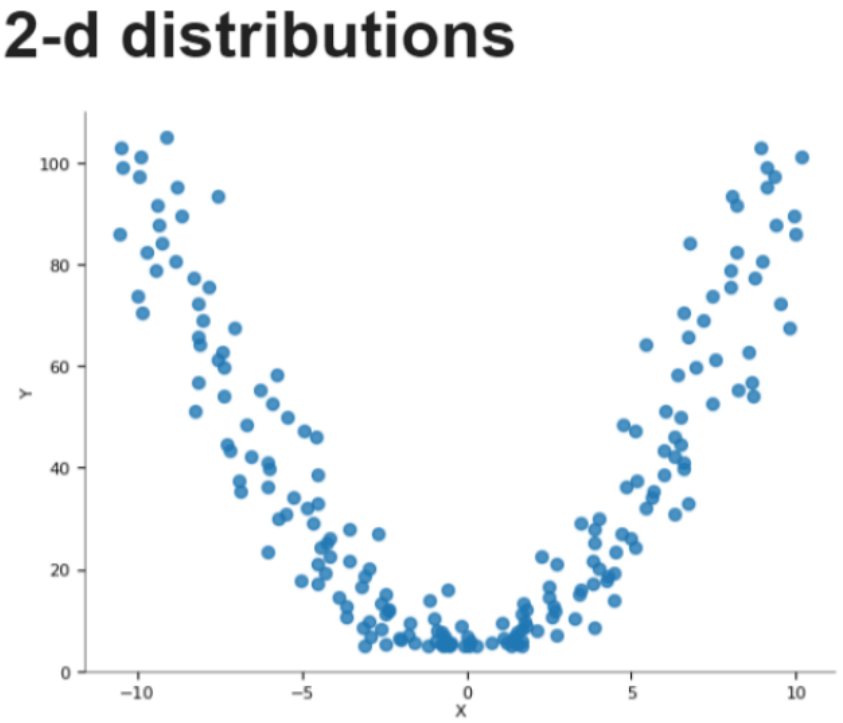
You have been given a one-dimensional dataset **(q4_dataset.csv)**. Consider column ***X*** as input features and column ***Y*** as labels and visualize the data given in a two-dimensional space.

a) If you want to fit a Linear Regression model on this dataset, there would be a basis (kernel) function that maps the features to a better space. Can you guess which one?

b) Divide this dataset into two **train** and **test** subsets by random sampling such as the test set includes 20% of the total dataset. Train a Linear Regression model using the kernel function asked in part (a) and evaluate it on the test set by **computing MSE (Minimum Squared Error)**. **Notice That in this task you are not allowed to use any ML libraries.**

Question 5

Our synthetic data looks like a quadratic function

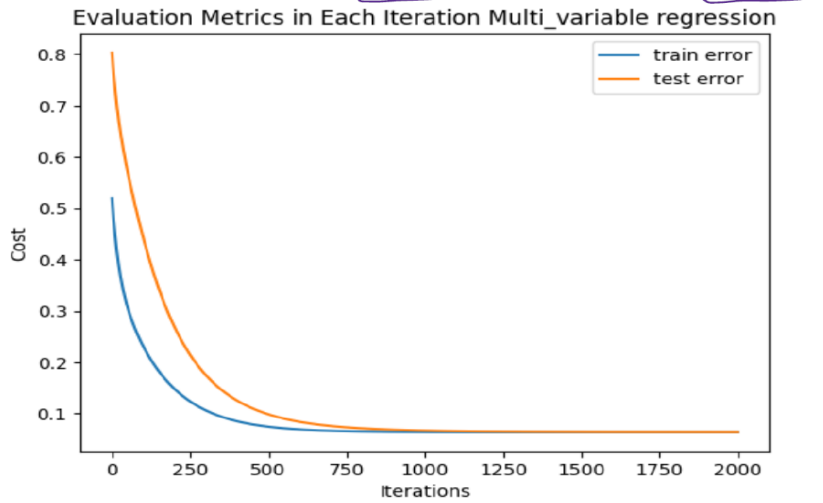


```python
class Regression:
  J=0
  def __init__(self, X, Y, W):
    self.X = X
    self.Y = Y
    self.W = W
  def cost(self):
    m=self.X.shape[0]
    J=(1 / (2 * m)) * np.sum ((np.dot(self.X , self.W) - self.Y) ** 2)
    return(J)

  def batch_GD(self, alpha, num_iters, X_test, Y_test):
    cost_list=list()
    eval_cost_list=list()
    m = self.X.shape[0]
    n = self.X.shape[1]
    grad = np.zeros (n)
    for i in range (0,num_iters+1):
      prediction = np.dot (self.X, self.W)
      J=(1 / (2 * m)) * np.sum ((prediction - self.Y) ** 2)
      MSE,_=self.evaluation(X_test, Y_test)

      if i%200==0:
        print('epoch=%i and the train_cost is= %f and evaluation error is %f'%(i,J, MSE))
      grad = (1 / m) * np.dot (self.X.T, (prediction - self.Y))
      self.W = self.W - alpha * grad
      cost_list.append([i,J])
      eval_cost_list.append([i,MSE])
    return cost_list, eval_cost_list, self.W

  def evaluation(self, X_test, Y_test):
    m = self.X.shape[0]
    Y_prediction = np.dot(X_test, self.W)
    mse= np.mean((Y_test - Y_prediction)**2)
    mae = np.mean(Y_test- Y_prediction)
    return mse, mae

Multi_variable = Regression(X_train, Y_train, W = np.zeros((X_train.shape[1], 1)))
cost_list, eval_cost_list, Weights = Multi_variable.batch_GD( 0.01, 2000, X_test, Y_test)
plt.plot(np.array(cost_list)[:,0], np.array(cost_list)[:,1], label='train error')
plt.plot(np.array(eval_cost_list)[:,0], np.array(eval_cost_list)[:,1], label='test error')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.title('Evaluation Metrics in Each Iteration Multi_variable regression')
plt.legend()
plt.show()
```

Simple regression model can fit it!

```python
quadratic = Regression(X_train, Y_train, W = np.zeros((X_train.shape[1], 1)))
cost_list, eval_cost_list, Weights = quadratic.batch_GD( 0.01, 2000, X_test, Y_test)
plt.plot(np.array(cost_list)[:,0], np.array(cost_list)[:,1], label='train error')
plt.plot(np.array(eval_cost_list)[:,0], np.array(eval_cost_list)[:,1], label='test error')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.title('Evaluation Metrics in Each Iteration Multi_variable regression')
plt.legend()
plt.show()
```

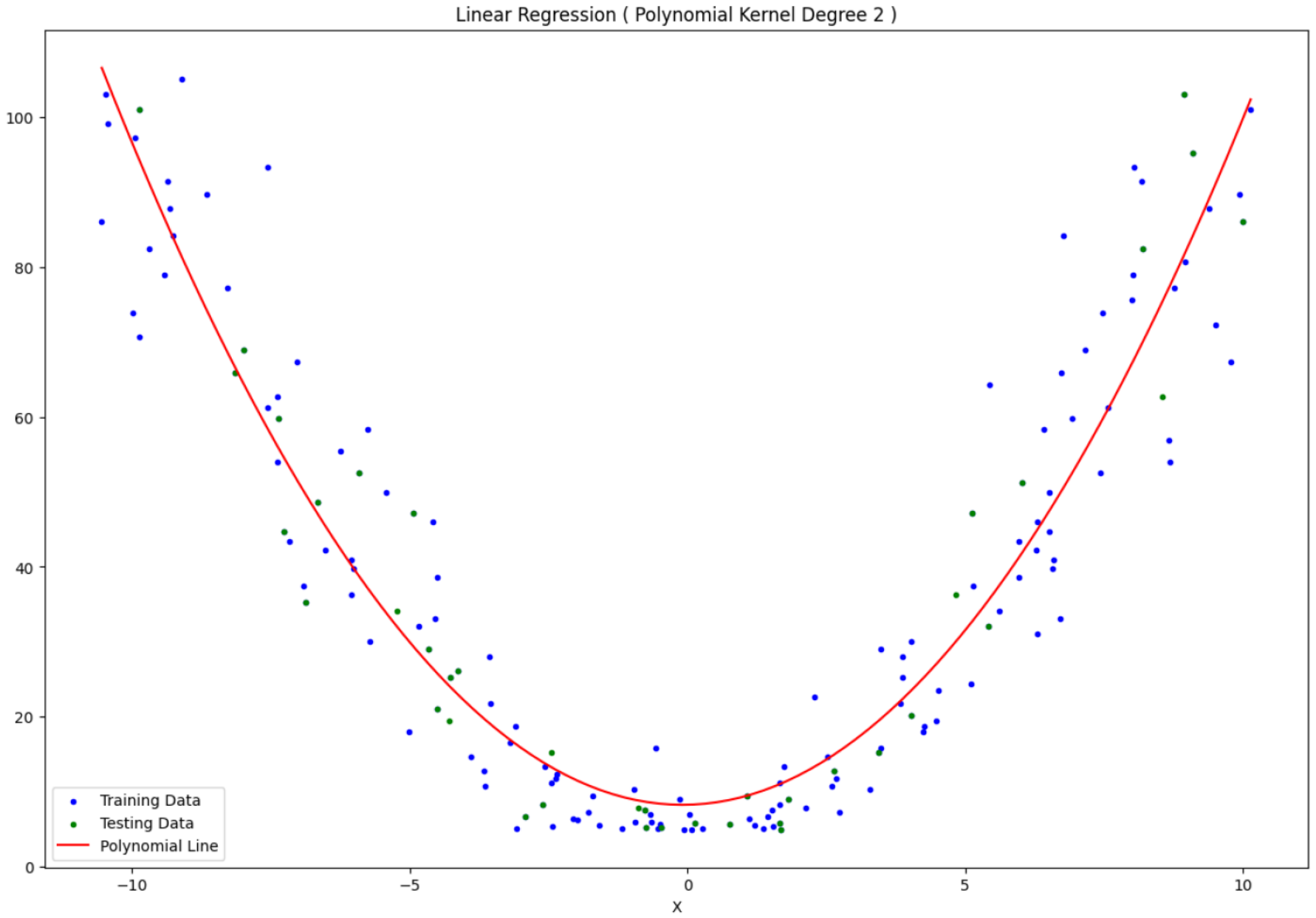
```
epoch=0 and the train_cost is= 0.519438 and evaluation error is 0.803262
epoch=200 and the train_cost is= 0.147934 and evaluation error is 0.268296
epoch=400 and the train_cost is= 0.085317 and evaluation error is 0.125313
epoch=600 and the train_cost is= 0.069294 and evaluation error is 0.084205
epoch=800 and the train_cost is= 0.065194 and evaluation error is 0.071433
epoch=1000 and the train_cost is= 0.064145 and evaluation error is 0.067024
epoch=1200 and the train_cost is= 0.063876 and evaluation error is 0.065319
epoch=1400 and the train_cost is= 0.063807 and evaluation error is 0.064592
epoch=1600 and the train_cost is= 0.063790 and evaluation error is 0.064258
epoch=1800 and the train_cost is= 0.063785 and evaluation error is 0.064098
epoch=2000 and the train_cost is= 0.063784 and evaluation error is 0.064019
```



So we reach with a good and acceptable train error only with one term and only with one term of quadratic feature!

M.Tamjidi

Here is our prediction!



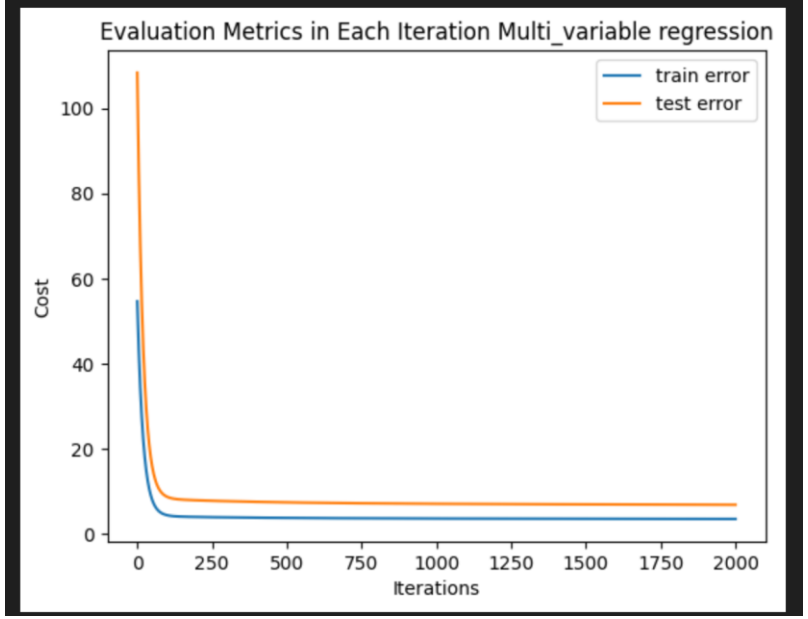
Question 6

We used the same multivariate regression class

linear regression

```
Multi_variable = Regression(X_train, Y_train, W = np.zeros((X_train.shape[1], 1)))
cost_list, eval_cost_list, Weights = Multi_variable.batch_GD( 0.01, 2000, X_test, Y_test)
plt.plot(np.array(cost_list)[:,0], np.array(cost_list)[:,1], label='train error')
plt.plot(np.array(eval_cost_list)[:,0], np.array(eval_cost_list)[:,1], label='test error')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.title('Evaluation Metrics in Each Iteration Multi_variable regression')
plt.legend()
plt.show()
```

```
epoch=0 and the train_cost is= 54.623282 and evaluation error is 108.363418
epoch=200 and the train_cost is= 3.947226 and evaluation error is 7.867307
epoch=400 and the train_cost is= 3.779119 and evaluation error is 7.500924
epoch=600 and the train_cost is= 3.682632 and evaluation error is 7.281932
epoch=800 and the train_cost is= 3.623628 and evaluation error is 7.143749
epoch=1000 and the train_cost is= 3.584746 and evaluation error is 7.050800
epoch=1200 and the train_cost is= 3.556720 and evaluation error is 6.983583
epoch=1400 and the train_cost is= 3.534599 and evaluation error is 6.931322
epoch=1600 and the train_cost is= 3.515740 and evaluation error is 6.887991
epoch=1800 and the train_cost is= 3.498728 and evaluation error is 6.850174
epoch=2000 and the train_cost is= 3.482809 and evaluation error is 6.815899
```



Polynomial degree=2

```python
polynomial_degree=2
def poly_features(X1, degree):
  poly = np.array(X1.copy(), dtype=np.float64) # for not overflowing
  for pow in range(2, degree+1):
    # use X instead of X_train
    # assign the result to poly_f
    poly = np.concatenate((poly, np.power(X1.copy()[:,1:], pow)), axis=1)
  return poly

poly=poly_features(X.to_numpy(),polynomial_degree)
print(poly.shape)

#train_test split
mask=np.random.uniform(size=(4177,1))>0.2
df = pd.DataFrame(poly)
X_train=df[mask]
Y_train=Y[mask]

X_test=df[~mask]
Y_test=Y[~mask]


X_train = X_train.to_numpy()
Y_train = Y_train.to_numpy()
X_test = X_test.to_numpy()
Y_test = Y_test.to_numpy()
```

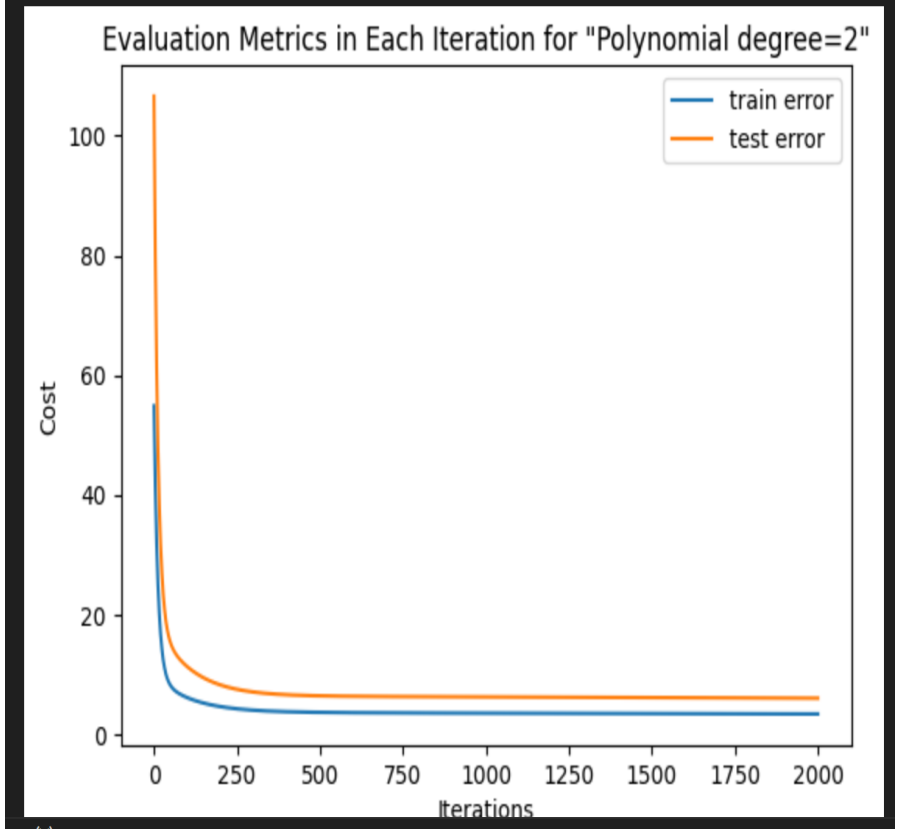
Applying kernel

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1.0	0.365	0.095	0.5140	0.2245	0.1010	0.1500	0.455	0.133225	0.009025	0.264196	0.050400	0.010201	0.022500	0.207025
1	1.0	0.265	0.090	0.2255	0.0995	0.0485	0.0700	0.350	0.070225	0.008100	0.050850	0.009900	0.002352	0.004900	0.122500
2	1.0	0.420	0.135	0.6770	0.2565	0.1415	0.2100	0.530	0.176400	0.018225	0.458329	0.065792	0.020022	0.044100	0.280900
3	1.0	0.365	0.125	0.5160	0.2155	0.1140	0.1550	0.440	0.133225	0.015625	0.266256	0.046440	0.012996	0.024025	0.193600
4	1.0	0.255	0.080	0.2050	0.0895	0.0395	0.0550	0.330	0.065025	0.006400	0.042025	0.008010	0.001560	0.003025	0.108900
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
4172	1.0	0.450	0.165	0.8870	0.3700	0.2390	0.2490	0.565	0.202500	0.027225	0.786769	0.136900	0.057121	0.062001	0.319225
4173	1.0	0.440	0.135	0.9660	0.4390	0.2145	0.2605	0.590	0.193600	0.018225	0.933156	0.192721	0.046010	0.067860	0.348100
4174	1.0	0.475	0.205	1.1760	0.5255	0.2875	0.3080	0.600	0.225625	0.042025	1.382976	0.276150	0.082656	0.094864	0.360000
4175	1.0	0.485	0.150	1.0945	0.5310	0.2610	0.2960	0.625	0.235225	0.022500	1.197930	0.281961	0.068121	0.087616	0.390625
4176	1.0	0.555	0.195	1.9485	0.9455	0.3765	0.4950	0.710	0.308025	0.038025	3.796652	0.893970	0.141752	0.245025	0.504100

Building new object of the regression class

```
polynomial_2 = Regression(X_train, Y_train, W = np.zeros((X_train.shape[1], 1)))
cost_list, eval_cost_list, Weights = polynomial_2.batch_GD( 0.01, 2000, X_test, Y_test)
plt.plot(np.array(cost_list)[:,0], np.array(cost_list)[:,1], label='train error')
plt.plot(np.array(eval_cost_list)[:,0], np.array(eval_cost_list)[:,1], label='test error')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.title('Evaluation Metrics in Each Iteration for "Polynomial degree=2"')
plt.legend()
plt.show()
```

```
epoch=0 and the train_cost is= 54.858222 and evaluation error is 106.613349
epoch=200 and the train_cost is= 4.595836 and evaluation error is 8.217610
epoch=400 and the train_cost is= 3.742093 and evaluation error is 6.573170
epoch=600 and the train_cost is= 3.581082 and evaluation error is 6.319130
epoch=800 and the train_cost is= 3.530438 and evaluation error is 6.255509
epoch=1000 and the train_cost is= 3.498252 and evaluation error is 6.211646
epoch=1200 and the train_cost is= 3.469973 and evaluation error is 6.167911
epoch=1400 and the train_cost is= 3.443250 and evaluation error is 6.124023
epoch=1600 and the train_cost is= 3.417663 and evaluation error is 6.080997
epoch=1800 and the train_cost is= 3.393099 and evaluation error is 6.039358
epoch=2000 and the train_cost is= 3.369497 and evaluation error is 5.999287
```



M.Tamjidi

Polynomial degree=3

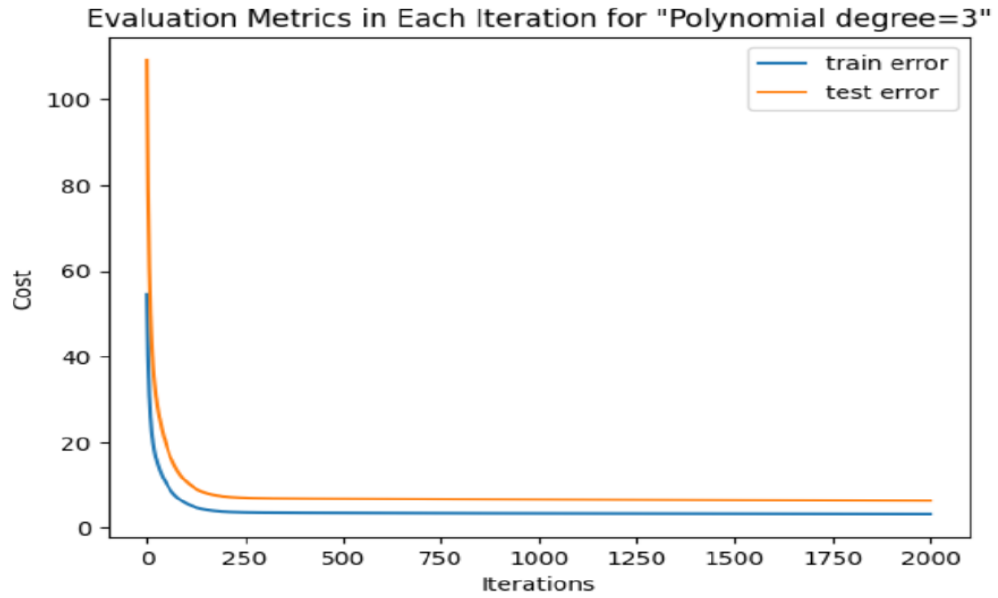
We just repeat the previous sections

C : Polynomial with degree=3

More terms further enhance the result.

```
polynomial_3 = Regression(X_train, Y_train, W = np.zeros((X_train.shape[1], 1)))
cost_list, eval_cost_list, Weights = polynomial_3.batch_GD( 0.01, 2000, X_test, Y_test)
plt.plot(np.array(cost_list)[:,0], np.array(cost_list)[:,1], label='train error')
plt.plot(np.array(eval_cost_list)[:,0], np.array(eval_cost_list)[:,1], label='test error')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.title('Evaluation Metrics in Each Iteration for "Polynomial degree=3"')
plt.legend()
plt.show()
```

```
epoch=0 and the train_cost is= 54.556760 and evaluation error is 108.905374
epoch=200 and the train_cost is= 3.812574 and evaluation error is 7.339771
epoch=400 and the train_cost is= 3.560359 and evaluation error is 6.867640
epoch=600 and the train_cost is= 3.512988 and evaluation error is 6.780217
epoch=800 and the train_cost is= 3.473210 and evaluation error is 6.705991
epoch=1000 and the train_cost is= 3.437815 and evaluation error is 6.638574
epoch=1200 and the train_cost is= 3.405851 and evaluation error is 6.576465
epoch=1400 and the train_cost is= 3.376641 and evaluation error is 6.518653
epoch=1600 and the train_cost is= 3.349691 and evaluation error is 6.464423
epoch=1800 and the train_cost is= 3.324636 and evaluation error is 6.413257
epoch=2000 and the train_cost is= 3.301201 and evaluation error is 6.364778
```



But in this case the train error decreases but dev set error increases a little which indicated that with increasing the polynomial degrees it trends to overfit our data.

To further enhance we will use the 5 gaussian kernel base function (they are incredible!)

M.Tamjidi

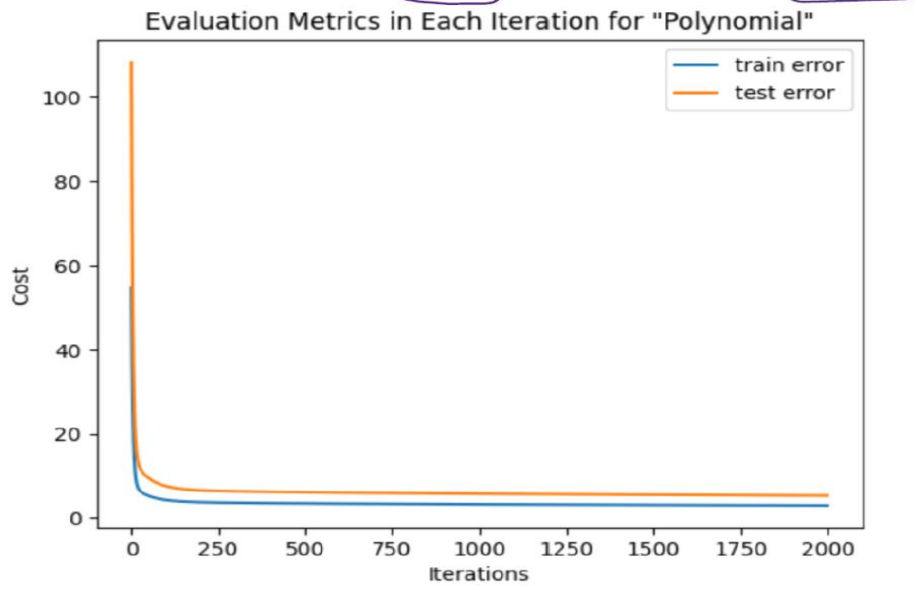
RBF

We just repeat the previous sections

D: RBF kernels

```
RBF= Regression(X_train, Y_train, W = np.zeros((X_train.shape[1], 1)))
cost_list, eval_cost_list, Weights = RBF.batch_GD( 0.01, 2000, X_test, Y_test)
plt.plot(np.array(cost_list)[:,0], np.array(cost_list)[:,1], label='train error')
plt.plot(np.array(eval_cost_list)[:,0], np.array(eval_cost_list)[:,1], label='test error')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.title('Evaluation Metrics in Each Iteration for "Polynomial"')
plt.legend()
plt.show()
```

```
epoch=0 and the train_cost is= 54.634557 and evaluation error is 108.300585
epoch=200 and the train_cost is= 3.737548 and evaluation error is 6.537384
epoch=400 and the train_cost is= 3.518734 and evaluation error is 6.177964
epoch=600 and the train_cost is= 3.403969 and evaluation error is 5.995619
epoch=800 and the train_cost is= 3.315504 and evaluation error is 5.853602
epoch=1000 and the train_cost is= 3.241129 and evaluation error is 5.733014
epoch=1200 and the train_cost is= 3.176164 and evaluation error is 5.626966
epoch=1400 and the train_cost is= 3.118355 and evaluation error is 5.532307
epoch=1600 and the train_cost is= 3.066381 and evaluation error is 5.447188
epoch=1800 and the train_cost is= 3.019342 and evaluation error is 5.370302
epoch=2000 and the train_cost is= 2.976568 and evaluation error is 5.300623
```



So as indicated the RBFs can bring the best accuracy and lower train and evaluation error.
We use the five term of gaussian kernels which it's details are in the provided code.

Thank you!

Feel free to contact me

Mehrant.0611@gmail.com
Telegram: @Mttnt