Testing

[ ] ↳ 3 cells hidden

ML Exercise 1-1

Question 1

Loading Data

```
# Download dataset
!wget https://raw.githubusercontent.com/Gholamrezadar/machine-learning-exercises/main/dataset1/dataset1.csv
```

```
--2021-11-20 19:14:28--  https://raw.githubusercontent.com/Gholamrezadar/machine-learning-exercises/main/dataset1/dataset
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ..
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 12074 (12K) [text/plain]
Saving to: ‘dataset1.csv.2’

dataset1.csv.2      100%[===================>]  11.79K  --.-KB/s    in 0s

2021-11-20 19:14:28 (86.2 MB/s) - ‘dataset1.csv.2’ saved [12074/12074]
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tqdm.notebook import tqdm
import math

# Load the data using pandas
df = pd.read_csv("dataset1.csv")
```

```
# Show a sample of the data
df.head()
```

	x	y
0	0.097627	0.626964
1	0.430379	0.846452
2	0.205527	0.756017
3	0.089766	0.427504
4	-0.152690	-1.335228

```
# Show a description of the data (might be useful later)
df.describe()
```

	x	y
count	300.000000	300.000000

Helper functions

Loss Functions

```
# h(theta) =  theta transpose * X
def hypothesis(X, theta):
  y1 = theta*X
  return np.sum(y1, axis=1)

def MSE(X, y, theta):
  y_hat = hypothesis(X, theta)
  m = len(X)
  return sum((y_hat-y)**2)/(2*m)

def RMSE(X, y, theta):
  y_hat = hypothesis(X, theta)
  m = len(X)
  return np.sqrt(sum((y_hat-y)**2)/(2*m))

def MAE(X, y, theta):
  y_hat = hypothesis(X, theta)
  m = len(X)
  return sum(np.abs((y_hat-y)))/(2*m)

# Loss functions Derivatives
def MSE_prim(X, y, i, theta):
  y_hat = hypothesis(X, theta)
  Xi = X.iloc[:, i]
  m = len(X)
  return sum((y_hat-y) * Xi) / m

def RMSE_prim(X, y, i, theta):
  # src : https://math.stackexchange.com/questions/4065532/rmse-derivatives
  mse = MSE(X, y, theta)
  mse_prim = MSE_prim(X, y, i, theta)

  return mse_prim / 2 / np.sqrt(mse)

def MAE_prim(X, y, i, theta):
  # src : https://stats.stackexchange.com/questions/312737/mean-absolute-error-mae-derivative
  # src2 : https://github.com/chenxingwei/machine_learning_from_scratch/blob/master/algorithm/2.linearRegressionGradientDe
  y_hat = hypothesis(X, theta)
  # print(np.sum((X.T*(np.sign(y_hat-y)/len(X))), axis=1)[i])
  return np.sum((X.T*(np.sign(y_hat-y)/len(X))), axis=1)[i]

```

Gradient Descent

```
def gradientDescent(X, y, theta, lr, iteration, X_valid, y_valid, loss_fn = MSE, loss_fn_prim = MSE_prim, decay=0.0):
  # Training loss per iteration history
  train_loss_history = []
  # Validation loss per iteration history
  validation_loss_history = []
  # weights progression towards the optimal value
  theta_history = []

  # Progress bar
  with tqdm(total=iteration) as pbar:
    for iter in range(iteration):
      for i in range(0, len(X.columns)):
        # partial derivative of loss function with respect to Xi
        gradient = loss_fn_prim(X, y, i, theta)

        # TODO : Learning rate decay
        # lr = lr * 1/(1 + decay * iter)
```

```
        # Actual "Gradient Descent" !
        theta[i] -= lr * gradient

      # Calculating the loss after each iteration
      # of updating the weights using Gradient Descent
      loss = loss_fn(X, y, theta)
      if X_valid is not None and y_valid is not None:
        validation_loss = loss_fn(X_valid, y_valid, theta)

      # Save the history of loss and weights
      train_loss_history.append(loss)
      if X_valid is not None and y_valid is not None:
        validation_loss_history.append(validation_loss)
      theta_history.append(theta.copy())

      # Update progress bar
      pbar.update(1)

  history = {"train_loss":train_loss_history,
             "validation_loss":validation_loss_history,
             "weights":theta_history}
  # returns loss history, latest loss, weights
  return history, loss, theta
```

```
# Add the polynomial's terms as features
# so that the univariate non-linear regression
# becomes a multivariate linear regression
# where every polynomial term is a feature for
# the linear regression
def polynomial_to_linear_regression(X, polynomial_degree):
  for i in range(2, 2 + polynomial_degree):
    X['x'+str(i)] = X['x']**i
```

Normal Equation

```
def normalEquation(X, y):
  # (X^T X)^-1 X^T Y
  XTX = np.dot(X.T,X)
  XTX_inverse = np.linalg.inv(XTX)
  XTY = np.dot(X.T,y)
  theta = np.dot(XTX_inverse, XTY)
  return theta

def regularizedNormalEquation(X, y, lambd=0.1):
  # (X^T X + lambda I)^-1 X^T Y
  XTX = np.dot(X.T,X) + np.dot(np.identity(X.shape[1]),lambd)
  XTX_inverse = np.linalg.inv(XTX)
  XTY = np.dot(X.T,y)
  theta = np.dot(XTX_inverse, XTY)
  return theta
```

Plotting related

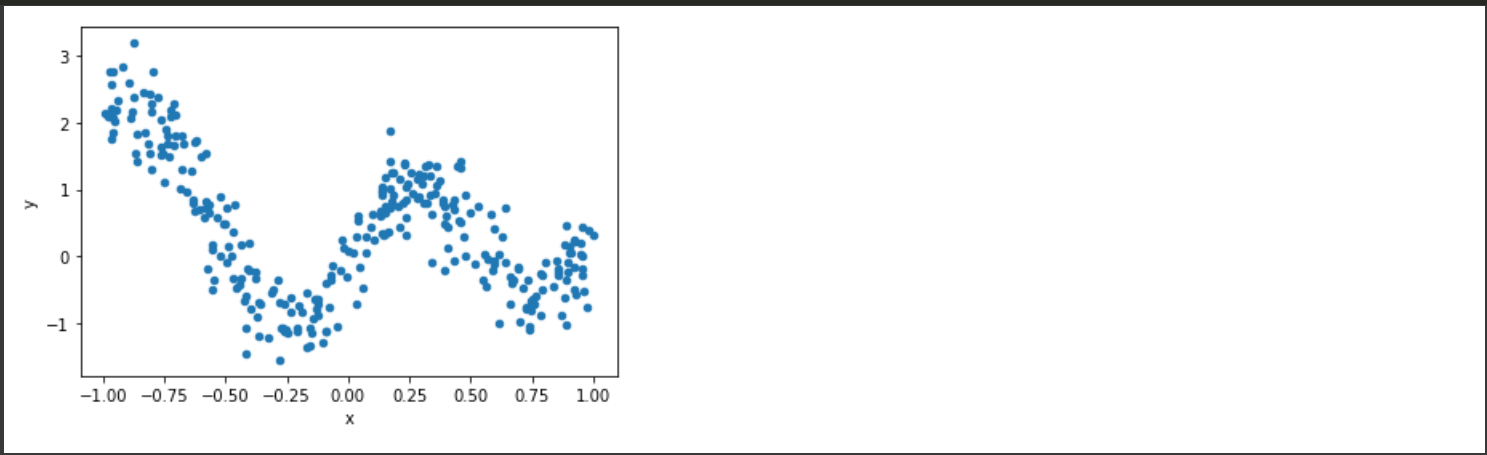
```
#  helper function used to plot a polynomial
def polyCoefficients(x, coeffs):
    o = len(coeffs)
    y = 0
    for i in range(o):
        y += coeffs[i]*x**i
    return y
```

```
# Plots a polynomial on top of the original data
def plot_curve(X, y, theta, c='r', title='', resolution=100):
  plt.figure()
  plt.title(title)
  # Plot the original data
  plt.scatter(x=X['x'],y= y)
```

```
7
8   x = np.linspace(-1, 1, resolution)
9   # Plot the fitted polynomial over the data
10  plt.plot(x, polyCoefficients(x, theta), c=c, linewidth=4)
11
12  plt.show()
```

Part 1 - Plotting the data

```
1 # Plot the data using matplotlib
2 df.plot(kind='scatter', x='x', y='y')
3 plt.show()
```



Part 2 - Shuffle

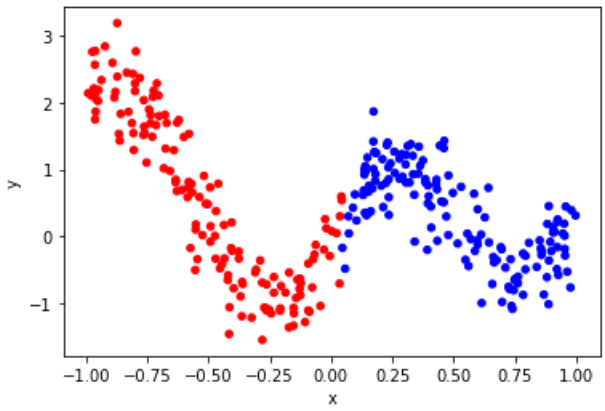
```
1 def plot_colorize(df):
2   '''
3   Assigns 'red' color to the first half of the data
4   and 'blue' to the rest
5
6   If the data is well shuffled we should see random red
7   and blue circles everywhere.
8
9   If the data is NOT well shuffled we might see a pattern between
10  circles' position and their color.
11  '''
12  col = {True:"red", False:"blue"}
13  colors = list(map(lambda x:col[x<df.shape[0]/2],df.index.tolist()))
14
15  df.plot(kind='scatter', x='x', y='y', color=colors)
16  plt.show()
```

Default data

```
1 plot_colorize(df)
2 print(" "*9,"Data seems to be well shuffled.")
```

Sorted data

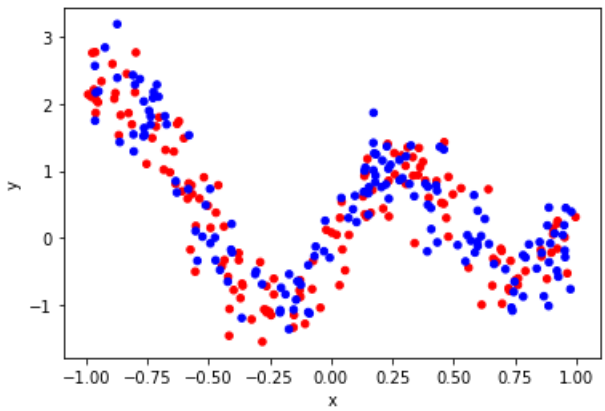
```
# Sort the data based on 'x' first, then do the
# previous part to see the result.
sorted_df = df.sort_values(by='x', ascending=True, ignore_index=True)

plot_colorize(sorted_df)
print(" "*6,"Data doesn't seem to be well shuffled.")
```




```
Data doesn't seem to be well shuffled.
```

Shuffled data

```
# Let's shuffle the data anyways (just in case)

# pandas Doc: specifying drop=True prevents .reset_index()
# from creating a column containing the old index entries.
shuffled_df = df.sample(frac=1).reset_index(drop=True)

plot_colorize(shuffled_df)
print(" "*9,"Data seems to be well shuffled.")
```




```
Data seems to be well shuffled.
```

Part 3 - Gradient Descent

Finding The optimal "Theta" values (weights)

Data Prepration

Add a column for bias

```
# Add a new column for simplicity of the calculations
# acts as the bias term
```

```
shuffled_df = pd.concat([pd.Series(1, index=shuffled_df.index, name='0'), shuffled_df], axis=1)
shuffled_df.head()
```

	0	x	y
0	1	0.963659	-0.527811
1	1	0.312659	1.348749
2	1	-0.125936	-0.631524
3	1	0.341276	-0.078979
4	1	0.923873	0.217419

Seperate X,y

```
# Split training data into X and y
X = shuffled_df.drop(columns="y")
y = shuffled_df.iloc[:, 2]

print(X.head(),end="\n\n")
print(y.head())
```

```
   0         x
0  1  0.963659
1  1  0.312659
2  1 -0.125936
3  1  0.341276
4  1  0.923873

0   -0.527811
1    1.348749
2   -0.631524
3   -0.078979
4    0.217419
Name: y, dtype: float64
```

```
# Split to train and valid
split = 0.7

X_train = X.iloc[ : int(len(X)*split),:].reset_index(drop=True)
X_valid = X.iloc[int(len(X)*split) : ,:].reset_index(drop=True)

y_train = y.iloc[ : int(len(X)*split)].reset_index(drop=True)
y_valid = y.iloc[int(len(X)*split) : ].reset_index(drop=True)

print(f"Train X size = {len(X_train)}")
print(f"Train y size = {len(y_train)}")
print(f"Valid X size = {len(X_valid)}")
print(f"Valid y size = {len(y_valid)}")
```

```
Train X size = 210
Train y size = 210
Valid X size = 90
Valid y size = 90
```

```
# Save a copy of X and y
# TODO might not need it
X_train_org = X_train.copy()
y_train_org = y_train.copy()
```

Polynomial Regression

a basic example

```
# polynomial degree
polynomial_degree = 12 #@param {type: "number"}
learning_rate = 0.030 #@param {type: "number"}
iterations = 1000 #@param {type: "number"}

```

polynomial_degree: 12

learning_rate: 0.030

iterations: 1000

Convert

aX + bX^2 + cX^3 + d

to

aX1 + bX2 + cX3 + d

```
# Add the polynomial's terms as features
# so that the univariate non-linear regression
# becomes a multivariate linear regression
# where every polynomial term is a feature for
# the linear regression
polynomial_to_linear_regression(X_train, polynomial_degree)
polynomial_to_linear_regression(X_valid, polynomial_degree)

X_train.head()
```

	0	x	x2	x3	x4	x5	x6	x7	x8	x9	x10	x1
0	1	0.963659	0.928638	0.894890	0.862369	0.831029	0.800829	7.717257e-01	7.436803e-01	7.166540e-01	6.906099e-01	6.655123e-0
1	1	0.312659	0.097756	0.030564	0.009556	0.002988	0.000934	2.920776e-04	9.132075e-05	2.855227e-05	8.927129e-06	2.791149e-0
2	1	-0.125936	0.015860	-0.001997	0.000252	-0.000032	0.000004	-5.024022e-07	6.327056e-08	-7.968048e-09	1.003465e-09	-1.263724e-1
								5.391851e-	1.840108e-	6.279842e-	2.143158e-	7.314078e

Training

```
# Initialize the weights with zero
theta = np.array([0.0]*len(X_train.columns))

# Initialize the weights with random values
theta = np.random.rand(len(X_train.columns),)

print("notice : takes approximately 3 minutes for 5k iters")

# Start the training
history, loss, theta = gradientDescent(X_train,
                                       y_train,
                                       theta,
                                       learning_rate,
                                       iterations,
                                       X_valid = X_valid,
                                       y_valid = y_valid,
                                       loss_fn=MSE,
                                       loss_fn_prim=MSE_prim,
                                       decay = 0.0)


```

```
notice : takes approximately 3 minutes for 5k iters
100%                                                1000/1000 [00:46<00:00, 21.89it/s]
```

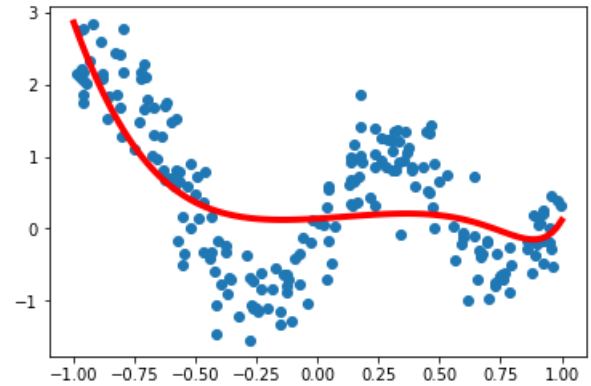
Plotting the fitted polynomials

```
# Predicting using the learned weights(theta)
# Not used here but useful
y_hat = theta*X_valid
y_hat = np.sum(y_hat, axis=1)
print(MSE(X_valid, y_valid, theta))
```

```
0.22926660232638735
```

```
plot_curve(X_train, y_train, theta)
```

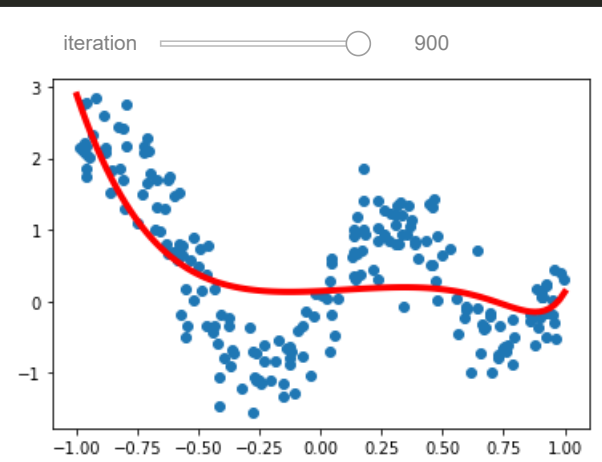
```
print(f"Training Loss : {RMSE(X_train, y_train, theta)}")
print(f"Validation Loss : {RMSE(X_valid, y_valid, theta)}\n")
```




```
Training Loss : 0.5101877849255999
Validation Loss : 0.4788179218934765
```

Interactive history viewer

```
from ipywidgets import interact
import ipywidgets as widgets

@interact(iteration = widgets.IntSlider(min=0, max=iterations-1, step=100, value=0))
def plot_weight_history(iteration):
  plot_curve(X_train, y_train, history["weights"][iteration])

print("Use the slider to see the algorithms progression")
```




```
Use the slider to see the algorithms progression
```

Part 4 - Plotting every Case!

Polynomial Degree :

- 5
- 8
- 10

Loss Functions :

- MSE
- RMSE
- MAE

Iterations :

- 5000
- 10000

```
# iterations = 1500 #TODO
polynomial_degrees = [5, 8, 10]
```

```
loss_functions = [(MSE, MSE_prim),
                  (RMSE, RMSE_prim),
                  (MAE, MAE_prim)]
fn_labels = ["MAE", "RMSE", "MSE"]
iterations = 2000
```

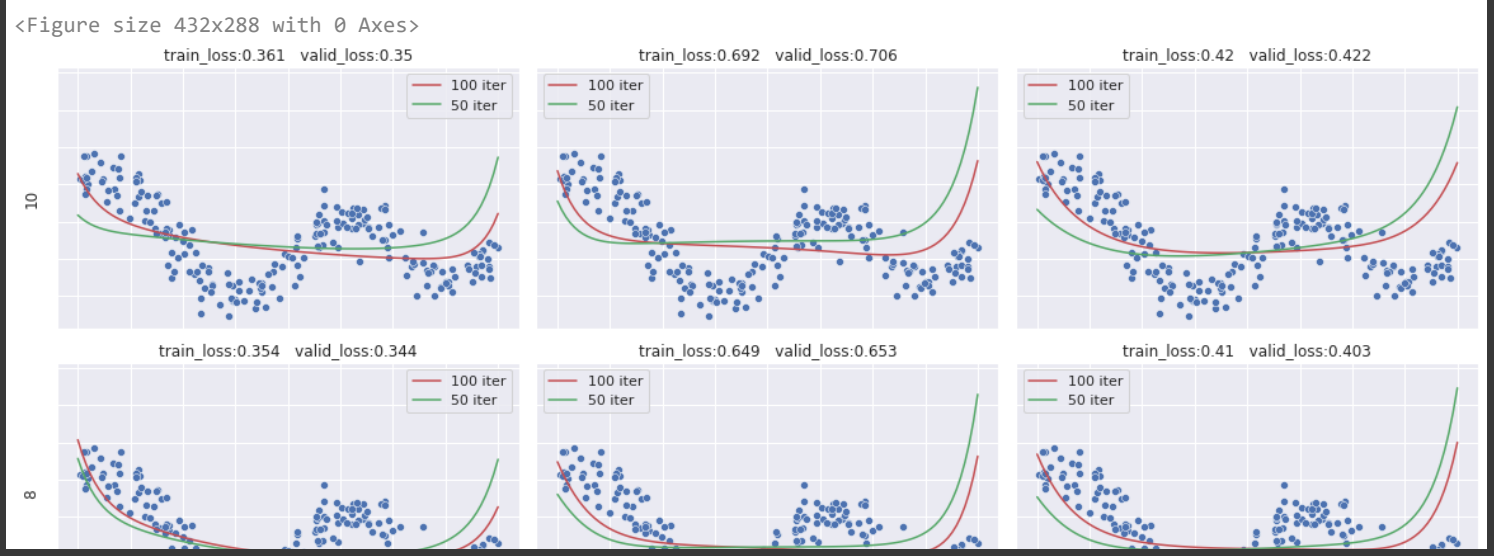
```
thetas = [[[],[],[]],
          [[],[],[]],
          [[],[],[]]]

losses = [[[],[],[]],
          [[],[],[]],
          [[],[],[]]]

for i,degree in enumerate(polynomial_degrees):
  for j, (loss_fn, loss_fn_prim) in enumerate(loss_functions):
    print(f"degree: {degree} | loss function: {fn_labels[j]}")

    # preprocess data
    X_train_copy = X_train.copy()
    X_valid_copy = X_valid.copy()

    polynomial_to_linear_regression(X_train_copy, degree)
    polynomial_to_linear_regression(X_valid_copy, degree)

    # TODO: gradient descent
    # Initialize the weights with random values
    theta = np.random.rand(len(X_train.columns),)
    # Start the training
    history, loss, theta = gradientDescent(X_train_copy,
                                           y_train,
                                           theta,
                                           learning_rate,
                                           iterations,
                                           X_valid = X_valid_copy,
                                           y_valid = y_valid,
                                           loss_fn=loss_fn,
                                           loss_fn_prim=loss_fn_prim,
                                           decay = 0.0)

    # latest iterations theta
    thetas[i][j].append(history["weights"][-1])
    # halfway theta
    thetas[i][j].append(history["weights"][int(iterations/2)-1])

    # Training loss
    losses[i][j].append(loss_fn(X_train_copy, y_train, theta))
    # Validation loss
    losses[i][j].append(loss_fn(X_valid_copy, y_valid, theta))

    print()
```

```
degree: 5 | loss function: MAE
100%                                          2000/2000 [00:51<00:00, 38.92it/s]

degree: 5 | loss function: RMSE
100%                                          2000/2000 [01:30<00:00, 22.86it/s]

degree: 5 | loss function: MSE
100%                                          2000/2000 [01:33<00:00, 21.39it/s]

degree: 8 | loss function: MAE
100%                                          2000/2000 [00:51<00:00, 38.12it/s]

degree: 8 | loss function: RMSE
30%                                           591/2000 [00:26<01:00, 23.15it/s]
```

```
# plt.subplot_tool()
```

```
def plot_every_curve(X, y, thetas, resolution=100):
  # Don't try this at home
  import warnings
  warnings.simplefilter(action="ignore", category=FutureWarning)

  plt.figure()
  fig, axes = plt.subplots(3, 3, sharex=True, sharey=True, figsize=(16,10), constrained_layout=True)
  # fig.subplots_adjust(hspace=0.6)

  # Helper lists for accessing the current config
  fn_labels = ["MAE", "RMSE", "MSE"]
  deg_labels = [10,8,5]

  # Used to plot the fitted polynomial in the range[-1,1]
  x = np.linspace(-1, 1, resolution)

  # Plotting every case in a 3 by 3 grid
  for i in range(3):
    for j in range(3):

      # 1. plot the original data (Blue)
      sns.scatterplot(x=X["x"], y=y, ax=axes[i,j])

      # 2. Plot a curve with Last iteration theta [0] (Red)
      theta = thetas[i][j][0]
      sns.lineplot(x, polyCoefficients(x, theta), color='r', ax=axes[i,j])

      # 3. Plot a curve with Middle iteration theta [1] (Green)
      theta = thetas[i][j][1]
      sns.lineplot(x, polyCoefficients(x, theta), color='g', ax=axes[i,j])

      # Legends and titles
      axes[i,j].legend(labels=[f"{iterations} iter", f"{iterations//2} iter"])
      axes[i,j].set_title(f"train_loss:{round(losses[i][j][0],3)}   valid_loss:{round(losses[i][j][1],3)}")

      # Matplotlib related code
      axes[i,j].xaxis.set_ticklabels([])
      axes[i,j].yaxis.set_ticklabels([])
      axes[i,j].set_xlabel(fn_labels[j])
      axes[i,j].set_ylabel(deg_labels[i])

  plt.show()
```

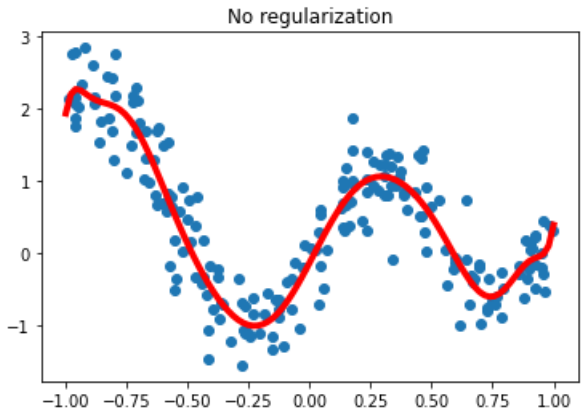
```
plot_every_curve(X_train, y_train, thetas)
```



Part 5 - Normal Equation

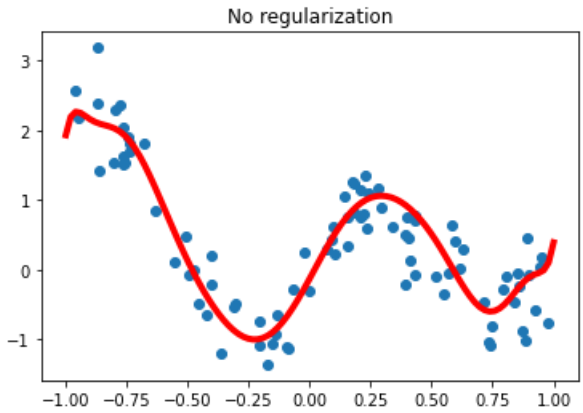
```
1 normal_theta = normalEquation(X_train, y_train)
```

```
1 # Plotting the fitted polynomial over training data
2 plot_curve(X_train, y_train, normal_theta, title="No regularization")
3 print(f"Training Loss : {RMSE(X_train, y_train, normal_theta)}")
```




```
Training Loss : 0.2750887262206953
```

```
1 # Plotting the fitted polynomial over validation data
2 plot_curve(X_valid, y_valid, normal_theta, title="No regularization")
3 print(f"Validation Loss : {RMSE(X_valid, y_valid, normal_theta)}\n")
```

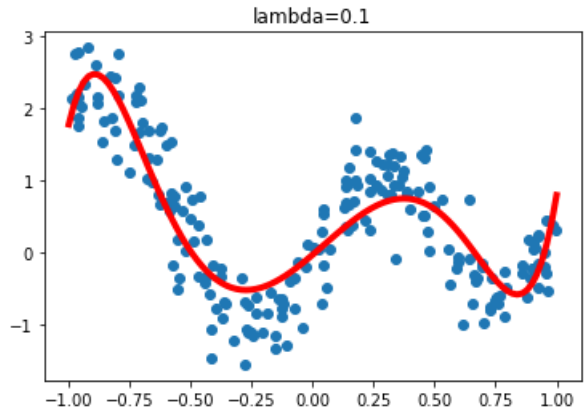



```
Validation Loss : 0.2855798993217868
```

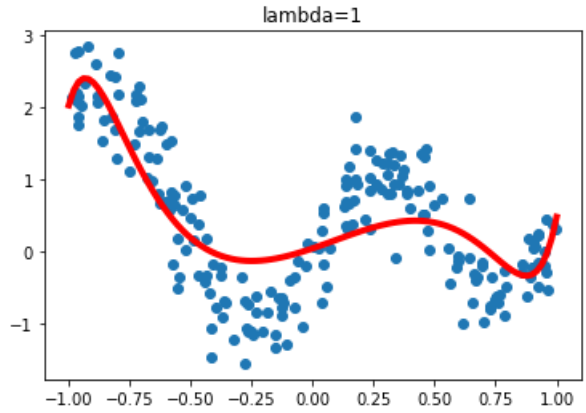
Part 6 - Regularized Normal Equation

```
1 reg_normal_theta = regularizedNormalEquation(X_train, y_train, lambd=.1)
2 plot_curve(X_train, y_train, reg_normal_theta, title="lambda=0.1")
```

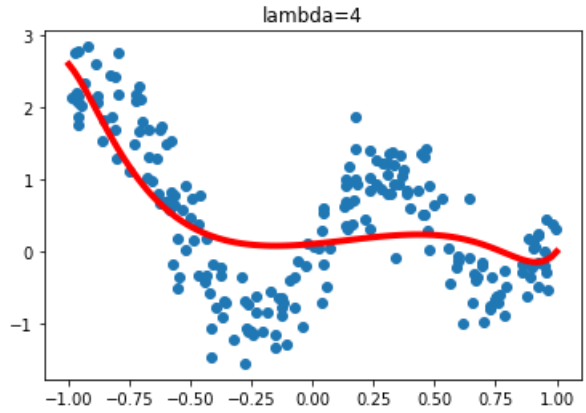
```
print(f"Training Loss : {RMSE(X_train, y_train, reg_normal_theta)}")
print(f"Validation Loss : {RMSE(X_valid, y_valid, reg_normal_theta)}\n")

reg_normal_theta = regularizedNormalEquation(X_train, y_train, lambd=1)
plot_curve(X_train, y_train, reg_normal_theta, title="lambda=1")
print(f"Training Loss : {RMSE(X_train, y_train, reg_normal_theta)}")
print(f"Validation Loss : {RMSE(X_valid, y_valid, reg_normal_theta)}\n")

reg_normal_theta = regularizedNormalEquation(X_train, y_train, lambd=4)
plot_curve(X_train, y_train, reg_normal_theta, title="lambda=4")
print(f"Training Loss : {RMSE(X_train, y_train, reg_normal_theta)}")
print(f"Validation Loss : {RMSE(X_valid, y_valid, reg_normal_theta)}\n")

```



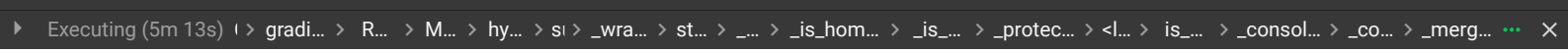

```
Training Loss : 0.34234945259970995
Validation Loss : 0.33329843645565466
```




```
Training Loss : 0.43260647240880507
Validation Loss : 0.4018612985008604
```




```
Training Loss : 0.5017630147189559
Validation Loss : 0.4717228178183581
```


Executing (5m 13s)
gradi…
R…
M…
hy…
sı
_wra…
st…
_…
_is_hom…
_is_…
_protec…
<l…
is_…
_consol…
_co…
_merg…