# ML Exercise 1-1

## Question 1

## Hyper parameters

change before running the note book

**learning_rate:** 0.030

**iterations:** 10000

Show code

## Loading Data

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

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

# use seaborn
sns.set()

# 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 |
|---|---|---|

# Helper functions

std 0.580948 1.021100

## Loss Functions

[ ] ↳ 1 cell hidden

50% 0.045096 0.316442

## Gradient Descent

max 0.997694 3.186153

```
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 = []

  lr0 = lr
  # Progress bar
  with tqdm(total=iteration) as pbar:
    for itera in range(iteration):
      # TODO : Learning rate decay
      lr = lr0 * 1/(1 + decay * itera)

      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)

        # 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 = {"training_loss":train_loss_history,
             "validation_loss":validation_loss_history,
             "weights":theta_history}
  # returns loss history, latest loss, weights
  print(f"training_loss : {round(train_loss_history[-1],4)} | validation_loss : {round(validation_loss_history[-1],4)}")
  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)

  x = np.linspace(-1, 1, resolution)
  # Plot the fitted polynomial over the data
  plt.plot(x, polyCoefficients(x, theta), c=c, linewidth=4)

  plt.show()
```

```
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 = ["degree 10","degree 8","degree 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
```

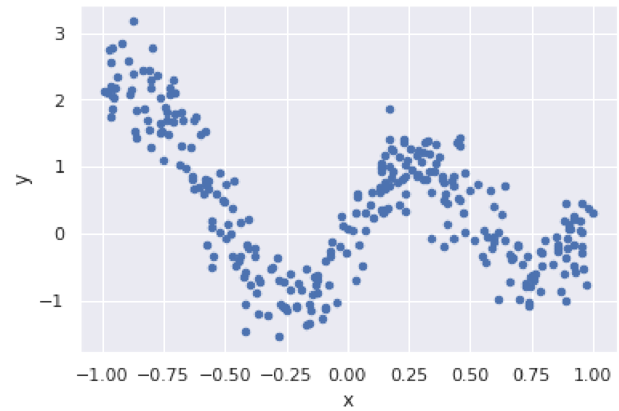
```
      axes[i,j].legend(labels=[f"{iterations} iter", f"{iterations//2} iter"])
      axes[i,j].set_title(f"train_loss:{round(losses[i][j][0],3)}   valid_loss:{round(losses[i][j][1],3)}")

      # 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()
```

```
def plot_every_case_loss(histories, starting_iter=0):
  # 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=False, figsize=(16,10), constrained_layout=True)

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

  # X = iterations range
  x = np.linspace(0, iterations, iterations)

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

      # 1. Training_loss - iteration curve (Red)
      sns.lineplot(x[starting_iter:], histories[i][j][0]["training_loss"][starting_iter:], color='r', ax=axes[i,j])
      # 2. Validation_loss - iteration curve (green)
      sns.lineplot(x[starting_iter:], histories[i][j][0]["validation_loss"][starting_iter:], color='g', ax=axes[i,j])

      # Legends
      axes[i,j].legend(labels=[f"training loss", f"validation loss"])

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

  plt.show()
```

## Part 1 - Plotting the data

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

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have preced
```

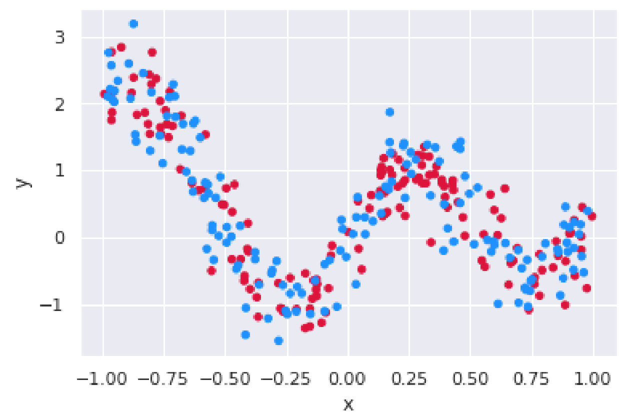



## Part 2 - Shuffle

```
def plot_colorize(df):
  '''
  Assigns 'red' color to the first half of the data
  and 'blue' to the rest

  If the data is well shuffled we should see random red
  and blue circles everywhere.

  If the data is NOT well shuffled we might see a pattern between
  circles' position and their color.
  '''
  # col = {True:"indianred", False:"cornflowerblue"}
  col = {True:"crimson", False:"dodgerblue"}

  colors = list(map(lambda x:col[x<df.shape[0]/2],df.index.tolist()))

  df.plot(kind='scatter', x='x', y='y', color=colors)
  plt.show()
```

## Default data

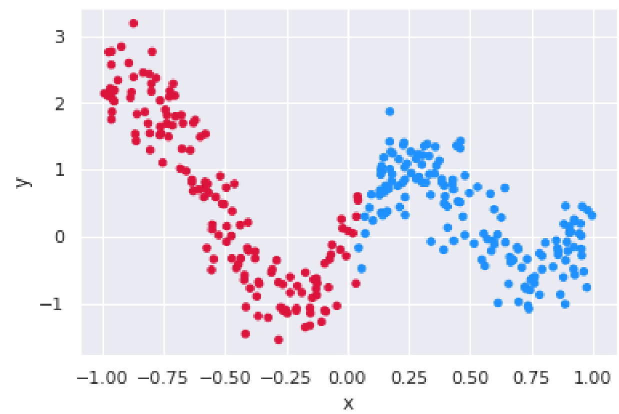
```
plot_colorize(df)
print(" "*9,"Data seems to be well shuffled.")
```




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

## Sorted data

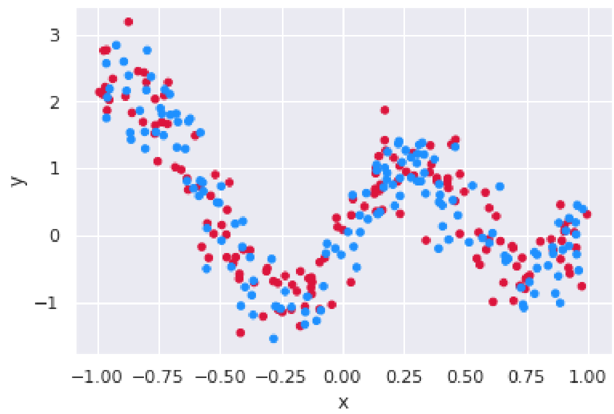
```
# Sort the data based on 'x' first, then do the
# previous part to see the result.
sorted_df = df.sort_values(by='x', ascending=True, ignore_index=True)

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 | 0.392687 | -0.202069 |
| **1** | 1 | -0.891324 | 2.592220 |
| **2** | 1 | 0.957237 | 0.444572 |
| **3** | 1 | 0.763471 | -0.603117 |
| **4** | 1 | -0.762545 | 1.637844 |

## 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  0.392687
1  1 -0.891324
2  1  0.957237
3  1  0.763471
4  1 -0.762545

0   -0.202069
```

```
1    2.592220
2    0.444572
3   -0.603117
4    1.637844
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:** 10

Show code

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 | x11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0.392687 | 0.154203 | 0.060554 | 0.023779 | 0.009338 | 0.003667 | 0.001440 | 0.000565 | 0.000222 | 0.000087 | 0.000034 |
| 1 | 1 | -0.891324 | 0.794459 | -0.708120 | 0.631164 | -0.562572 | 0.501434 | -0.446940 | 0.398368 | -0.355075 | 0.316487 | -0.282093 |
| 2 | 1 | 0.957237 | 0.916302 | 0.877118 | 0.839609 | 0.803705 | 0.769336 | 0.736437 | 0.704944 | 0.674798 | 0.645942 | 0.618319 |
| 3 | 1 | 0.763471 | 0.582888 | 0.445018 | 0.339758 | 0.259395 | 0.198041 | 0.151198 | 0.115435 | 0.088132 | 0.067286 | 0.051371 |
| 4 | 1 | -0.762545 | 0.581474 | -0.443400 | 0.338112 | -0.257826 | 0.196604 | -0.149919 | 0.114320 | -0.087174 | 0.066474 | -0.050689 |

## 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")

# tip : nice way to find decay (https://machinelearningmastery.com/using-learning-rate-schedules-deep-learning-models-pyth
# Decay = LearningRate / Epochs
# Decay = 0.1 / 1000
# Decay = 0.0001

# 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)
```

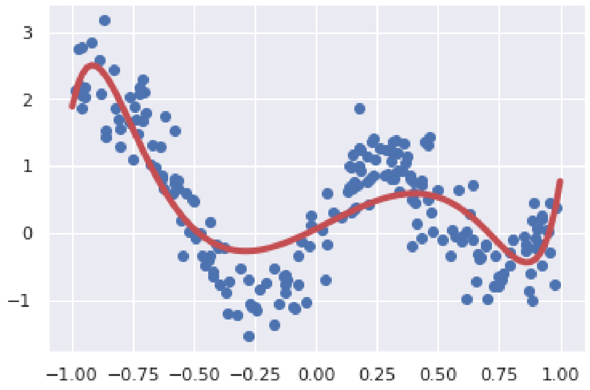
```
notice : takes approximately 3 minutes for 5k iters
100%                                   10000/10000 [06:14<00:00, 26.80it/s]
training_loss : 0.1491 | validation_loss : 0.1609
```

## 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))
```

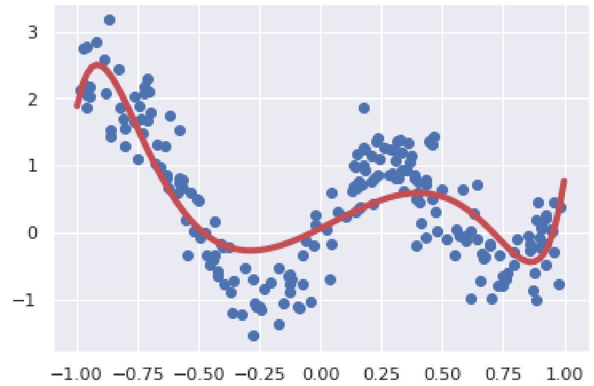
```
0.16093676345655536
```

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




```
Training Loss : 0.14911607013673991
Validation Loss : 0.16093676345655536
```

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

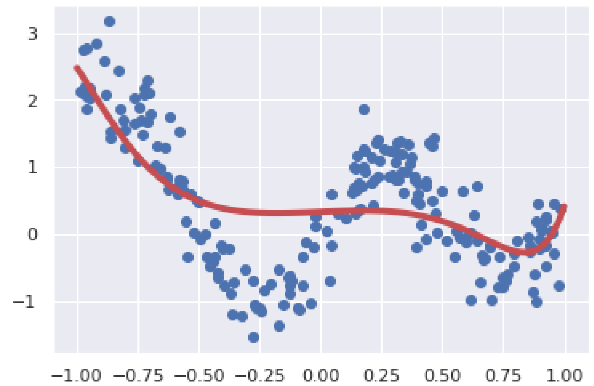



```
Training Loss : 0.14911607013673991
```

## 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")
```

iteration 1400




```
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 = 100
polynomial_degrees = [5, 8, 10]
loss_functions = [(MSE, MSE_prim),
                  (RMSE, RMSE_prim),
                  (MAE, MAE_prim)]
fn_labels = ["MAE", "RMSE", "MSE"]
```

## Training all 9 models

```
# Containers used for storing results about each model
# Used later by the plotting functions
thetas = [[[],[],[]],
          [[],[],[]],
          [[],[],[]]]

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

histories = [[[],[],[]],
             [[],[],[]],
             [[],[],[]]]

# start the training process for each model
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 (univariate non-linear to multivariate linear)
    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)

    # 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)

    # Saving latest iteration's theta for each model
    thetas[i][j].append(history["weights"][-1])
    # Saving halfway theta for each model
    thetas[i][j].append(history["weights"][int(iterations/2)-1])

    # Saving Training loss for each model
    losses[i][j].append(loss_fn(X_train_copy, y_train, theta))
    # Saving Validation loss for each model
    losses[i][j].append(loss_fn(X_valid_copy, y_valid, theta))
    # Saving Histories for each model (used for plotting loss per iteration)
    histories[i][j].append(history)

    print()
```

```
degree: 5 | loss function: MAE
```

100% 10000/10000 [06:12<00:00, 27.36it/s]

```
training_loss : 0.1473 | validation_loss : 0.1593

degree: 5 | loss function: RMSE
```

100% 10000/10000 [11:12<00:00, 15.22it/s]

```
training_loss : 0.3774 | validation_loss : 0.3922

degree: 5 | loss function: MSE
```

100% 10000/10000 [06:47<00:00, 24.75it/s]

```
training_loss : 0.2056 | validation_loss : 0.2186

degree: 8 | loss function: MAE
```

100% 10000/10000 [06:12<00:00, 27.32it/s]

```
training_loss : 0.1475 | validation_loss : 0.1592

degree: 8 | loss function: RMSE
```

100% 10000/10000 [11:09<00:00, 14.46it/s]

```
training_loss : 0.3778 | validation_loss : 0.3927

degree: 8 | loss function: MSE
```

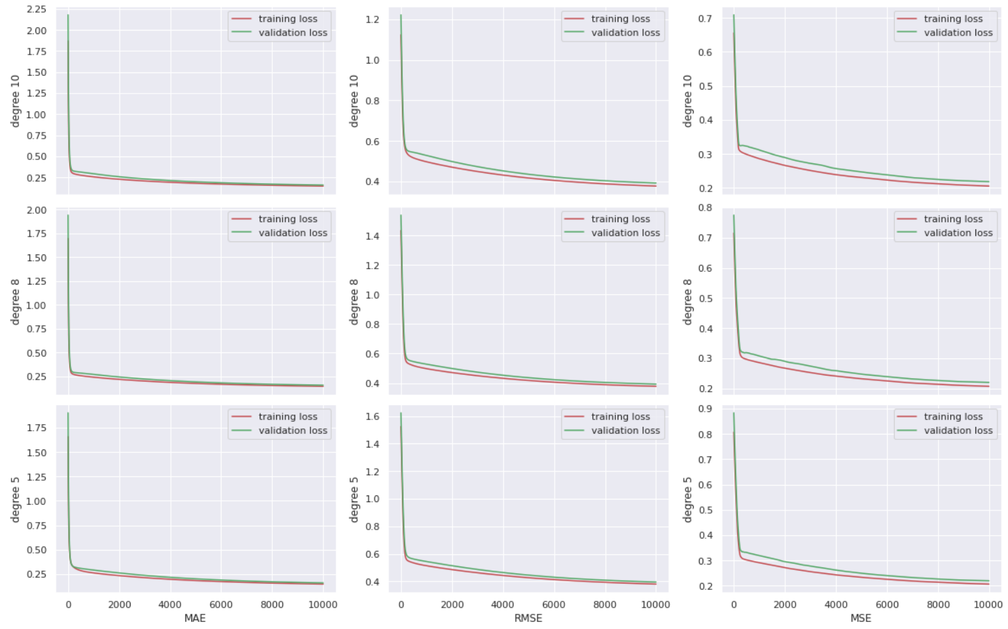
```
plot_every_curve(X_train, y_train, thetas)
```

```
<Figure size 432x288 with 0 Axes>
```


## Plotting train/valid loss

```
plot_every_case_loss(histories)
```

<Figure size 432x288 with 0 Axes>




## Plotting Learning Rate

```
_lrs = []
# Iterations
_iterations = 100
# Initial lr
_lr0 = 0.1
# Decay
_decay = _lr0/_iterations

# Simulate gradient descents main loop
_lr = _lr0
for i in range(_iterations):
  _lr = _lr * 1/(1 + _decay * i)
  _lrs.append(_lr)

_x = list(range(_iterations))
_y = _lrs

plt.plot(_x, _y)
```

```
[<matplotlib.lines.Line2D at 0x7f0646089f50>]
```

## Part 5 - Normal Equation

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

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

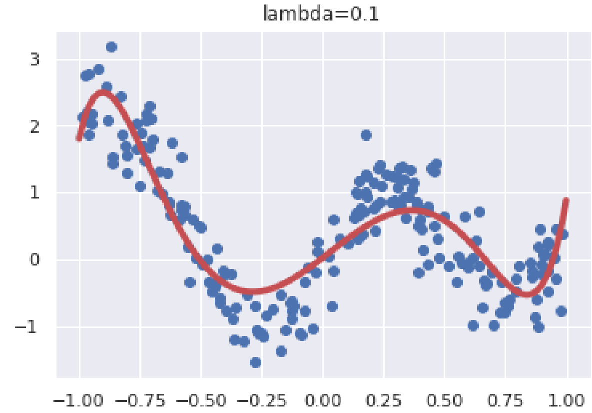

```
Training Loss : 0.26567087402148526
```

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

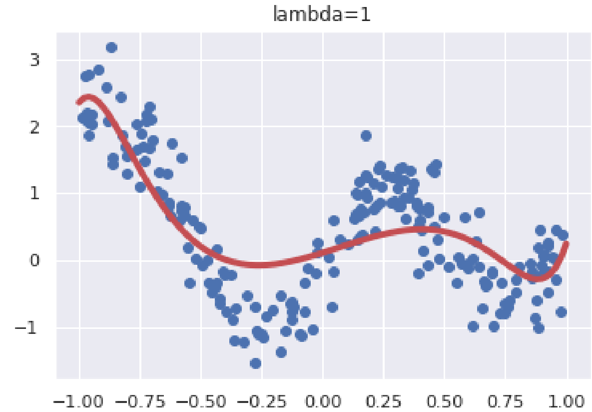

```
Validation Loss : 0.3095475085283209
```

## Part 6 - Regularized Normal Equation

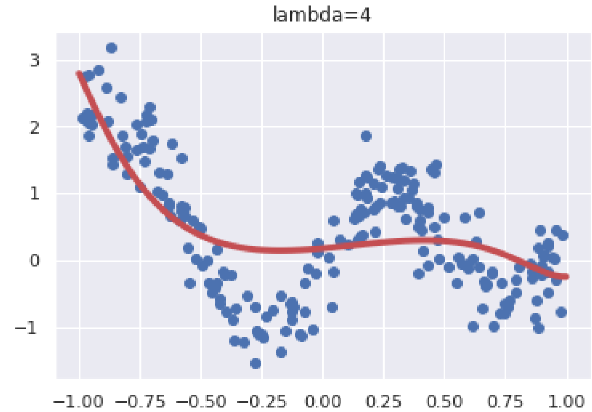
```
reg_normal_theta = regularizedNormalEquation(X_train, y_train, lambd=.1)
plot_curve(X_train, y_train, reg_normal_theta, title="lambda=0.1")
print(f"Training Loss : {RMSE(X_train, y_train, reg_normal_theta)}")
print(f"Validation Loss : {RMSE(X_valid, y_valid, reg_normal_theta)}\n")

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 : 0.34126275359191854
Validation Loss : 0.35843601252690255
```




```
Training Loss : 0.4250887587842727
Validation Loss : 0.44552564006365325
```




```
Training Loss : 0.4910232709927601
Validation Loss : 0.5202602652456745
```

✓ 0s completed at 7:20 AM