ECS708: *Machine Learning*
Lab Assignment 1 Part 1: REPORT

Student details...

- **Number**: 231052045
- **Name**: Pranav Narendra Gopalkrishna

0. Introduction

For this lab we will use the diabetes dataset. Furthermore, we shall make the following libraries and modules to support our implementation...

- `torch`
- `nn` module from `torch`
- `model_selection` module from `sklearn`
- `pandas` (aliased as `pd`)
- `matplotlib.pyplot` (aliased as `plt`)`

For reference, here are a few rows of the dataset...

```
diabetes_db.head(10)
```

	AGE	SEX	BMI	BP	S1	S2	S3	S4	S5	S6	Y
0	59	2	32.1	101.0	157	93.2	38.0	4.00	4.8598	87	151
1	48	1	21.6	87.0	183	103.2	70.0	3.00	3.8918	69	75
2	72	2	30.5	93.0	156	93.6	41.0	4.00	4.6728	85	141
3	24	1	25.3	84.0	198	131.4	40.0	5.00	4.8903	89	206
4	50	1	23.0	101.0	192	125.4	52.0	4.00	4.2905	80	135
5	23	1	22.6	89.0	139	64.8	61.0	2.00	4.1897	68	97
6	36	2	22.0	90.0	160	99.6	50.0	3.00	3.9512	82	138
7	66	2	26.2	114.0	255	185.0	56.0	4.55	4.2485	92	63
8	60	2	32.1	83.0	179	119.4	42.0	4.00	4.4773	94	110

	AGE	SEX	BMI	BP	S1	S2	S3	S4	S5	S6	Y
9	29	1	30.0	85.0	180	93.4	43.0	4.00	5.3845	88	310

We first split the data into test and training sets. For consistency and to allow for meaningful comparison the same splits are maintained in the remainder of the lab. The identifiers are as follows:

- `x_train` : Training data's feature (independent variable) columns
  - *This will be normalized*
- `x_test` : Testing data's feature (independent variable) columns
  - *This will be normalized*
- `y_train` : Training data's target (dependent variable) columns
- `y_test` : Testing data's target (dependent variable) columns

In the above data, we see that all the independent variables are on different scales. This can affect gradient descent, we therefore need to normalize all features to zero mean, and unit standard deviation. The normalized value z_i of x_i is obtained through $z_i = \frac{x_i - \mu}{\sigma}$ where μ is the mean and σ is the standard deviation of X and $x_i, \mu, \sigma \in \mathbb{R}^D$.

1.1 Linear Regression

Introduction

We define hypothesis function...

$$y = f(x) = w^T x$$

... so we need to learn weight vector w.

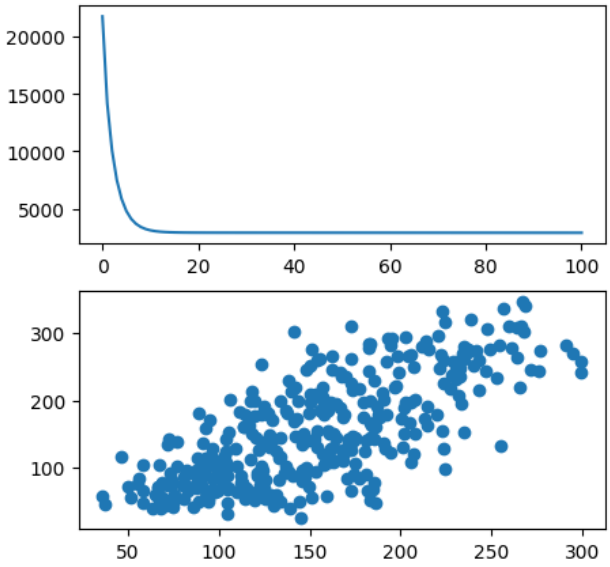
Question 5

What conclusion if any can be drawn from the weight values? How does gender and BMI affect blood sugar levels? What are the estimated blood sugar levels for the below examples? [2 marks]

AGE	SEX	BMI	BP	S1	S2	S3	S4	S5	S6
25	F	18	79	130	64.8	61	2	4.1897	68
50	M	28	103	229	162.2	60	4.5	6.107	124

Performing gradient descent iteratively (100 times), we get the following results. The top graph shows the cost function's curve, specifically showing how the cost reduces with each gradient descent step. The second graph is a scatterplot where the x-axis denotes the predicted values of the model while the

y-axis denotes the true y-values (target variable values) observed in the training data after a 100 gradient descent steps.




```
MODEL WEIGHTS
w0      :  153.736526 --> bias
w1      :  1.939769 --> AGE
w2      : -11.449014 --> SEX
w3      :  26.303600 --> BMI
w4      :  16.631128 --> BP
w5      : -9.944833 --> S1
w6      : -2.269463 --> S2
w7      : -7.669748 --> S3
w8      :  8.223666 --> S4
w9      :  22.000544 --> S5
w10     :  2.605969 --> S6

COSTS
Minimum : 2890.245849609375
Final   : 2890.245849609375
```

CONCLUSIONS: **Part 1**:

Judging by the weights alone, we can draw the following conclusions:

1. The magnitude of a variable's impact on the predicted target value

- *This indicates the strength of the relationship between the given predictor and target variables*

2. Whether the impact is negative (i.e. higher the predictor value, lower the predicted target value, all else equal) or positive (i.e. higher the predictor value, higher the predicted target value, all else equal)

- *This indicates the kind of relationship (proportional or inverse) between the given predictor and target variables*

Hence, we see that BMI (body mass index), BP (blood pressure) and S5 are the biggest factors contributing to the predicted blood sugar level, with sex, S1, S3 and S4 also having a moderate to high impact. Furthermore, we see that the impact of BMI, BP and S5 is positive, while the impact of sex (discussed later below), S1 and S3 is negative. Looking at the less significantly weighted variables, we see that age and S6 has a relatively small and positive impact, while S2 have a low and negative impact.

CONCLUSIONS: **Part 2**:
BMI has a relatively high positive weight, indicating that there is a strong positive relationship between blood sugar levels and BMI; the higher the BMI, the more the blood sugar levels tend to be (with significant correlation). This matches common knowledge, since people have BMI's due to having higher body weight relative to their height (such is the calculation of BMI), which tends to be a result of the higher consumption energy-dense foods such as carbohydrates (which break down to sugar) and sugars.

Gender (given as "SEX" in the data) has a relatively high negative weight (*high in terms of absolute magnitude*). To interpret this, we must first note the way gender is encoded in the data; 2 for female, 1 for male (with the normalised values being 1.0599 and -0.9408 respectively). The high negative weight implies that the higher the value of the sex "SEX", the lower the blood sugar level tends to be (with significant correlation). This indicates that being a female (i.e. being given a higher value in this encoding) has a negative impact on the predicted blood sugar level, tending to lower it. Reviewing some research on gender differences in blood sugar (or glucose) levels, we see that women have a higher risk of hypoglycaemia (low blood sugar) than men and tend to have a lower blood sugar level than men on average. This matches the weightage given to the variable "SEX" in our linear regression model.

References:

- "Gender differences in foods uptakes, glycemic index, BMI, and various plasma parameters between young men and women in Japan"
  - Link: https://www.oatext.com/Gender-differences-in-foods-uptakes-glycemic-index-BMI-and-various-plasma-parameters-between-young-men-and-women-in-Japan.php
  - Quote: "*Basic levels of blood glucose levels are lower in women than men*."
- "The effects of baseline characteristics, glycaemia treatment approach, and glycated haemoglobin concentration on the risk of severe hypoglycaemia: post hoc epidemiological analysis of the ACCORD study"
  - Link: https://www.bmj.com/content/340/bmj.b5444.long

- Quote: "*We found significantly increased risks for hypoglycaemia among women*"

Estimating blood sugar levels for:

AGE	SEX	BMI	BP	S1	S2	S3	S4	S5	S6
25	F	18	79	130	64.8	61	2	4.1897	68
50	M	28	103	229	162.2	60	4.5	6.107	124

```
# Function to normalize new data & get the result
"""
As our model was trained on a normalised distrribution, it can predict reliably
only for features lying in the same distribution; thus, we normalise the given
feature values (x-values) using the mean & standard deviation of the training
set.

NOTE: We normalised only for x-values, not y-values (i.e. target values).
"""
def predict(model, x):
  x = norm_set(x, mu["x_train"], sigma["x_train"])
  # Adding the bias after normalisation
  x = torch.cat([torch.tensor([1]), x], dim=0)
  y = model.forward(x).item()
  return y


x1 = torch.tensor([25, 2, 18, 79, 130, 64.8, 61, 2, 4.1897, 68])
x2 = torch.tensor([50, 1, 28, 103, 229, 162.2, 60, 4.5, 6.107, 123])
prediction1 = predict(model, x1)
prediction2 = predict(model, x2)
print("ESTIMATED BLOOD SUGAR LEVELS")
print(f"Person 1: {prediction1:.5f}")
print(f"Person 2: {prediction2:.5f}")
```

```
ESTIMATED BLOOD SUGAR LEVELS
Person 1: 43.54755
Person 2: 232.10165
```

Now estimate the error on the test set. Is the error on the test set comparable to that of the train set? What can be said about the fit of the model? When does a model over/under fits?

```
prediction_test = model.forward(x_test)
print(mean_squared_error(prediction_test, y_test))
```

```
2885.6298828125
```

```
# Printing the target value's training data's mean & standard deviation
# (for getting an idea about the shift & scale of the target values)
print(mu["y_train"].item())
print(sigma["y_train"].item())
```

```
153.73654174804688
78.06190490722656
```

CONCLUSIONS:
We see that the mean squared errors on the test set ≈ 2885.609 is very close to the mean squared errors on the training set ≈ 2890.339. Given the scale of the variation in the target variable's values in the training set (*standard deviation* ≈ 78.062), the difference between the mean squared errors of the training and testing sets is insignificant (0.06% *of the standard deviation of the target variable's values in the training set*).

This indicates that the model is not overfitted, i.e. the model's weights are not fine-tuned to the training set and are generalised enough to make predictions for new data with reasonable accuracy. An underfitted model is indicated by the mean squared errors being high in any set, training or testing. An overfitted model is indicated by the mean squared error being low in the training set but high in the testing set.

Question 6

Try the code with a number of learning rates that differ by orders of magnitude and record the error of the training and test sets. What do you observe on the training error? What about the error on the test set? [3 marks]

```
# FUNCTION TO PERFORM GRADIENT DESCENT LOOP
# (... to perform gradient descent steps iteratively)
def gradient_descent_loop(x, y, alpha, nIterations, getAllCosts=False):
  # NOTE: nIterations ==> Number of loop iterations

  # Initializing the model:
  model = LinearRegression(x.shape[1])
  # The future list of costs obtained:
  costList = []
  #------------------------
  # We will run this loop for `nIterations` iterations
  for i in range(nIterations+1):
    # Doing gradient descent & getting updated model's predictions:
    prediction = model.forward(x)
    gradient_descent_step(model, x, y, prediction, alpha)
    if getAllCosts: costList.append(mean_squared_error(prediction, y))
  #------------------------
  # Returning obtained mean squared error (cost)
  if getAllCosts: return costList
  return mean_squared_error(prediction, y)
```

Defining the α values in a list...

```
alphaList = [0.1**(n) for n in range(-2, 4)]
```

Obtaining all the cost-related data for training and testing data...

```
def getCostData(x, y):
  costs, costsPerLoop = [], []
  for alpha in alphaList:
    costs.append(gradient_descent_loop(x, y, alpha, 5))
    costsPerLoop.append(gradient_descent_loop(x, y, alpha, 50, True))
    # NOTE: costsPerLoop is for plotting purposes later
    """
    IMPLEMENTATION NOTE:
    We are running fewer iterations for the numerical data because a few
    iterations is enough to show whether the cost converges or diverges.

    We are running many more iterations for plotting to get smoother cost
    function curves and better portray their convergence or divergence
    graphically.
    """
  return costs, costsPerLoop

# Obtaining the cost-related data
train = getCostData(x_train, y_train)
test = getCostData(x_test, y_test)
```

Displaying the final costs obtained for both training and testing data for the different α values...

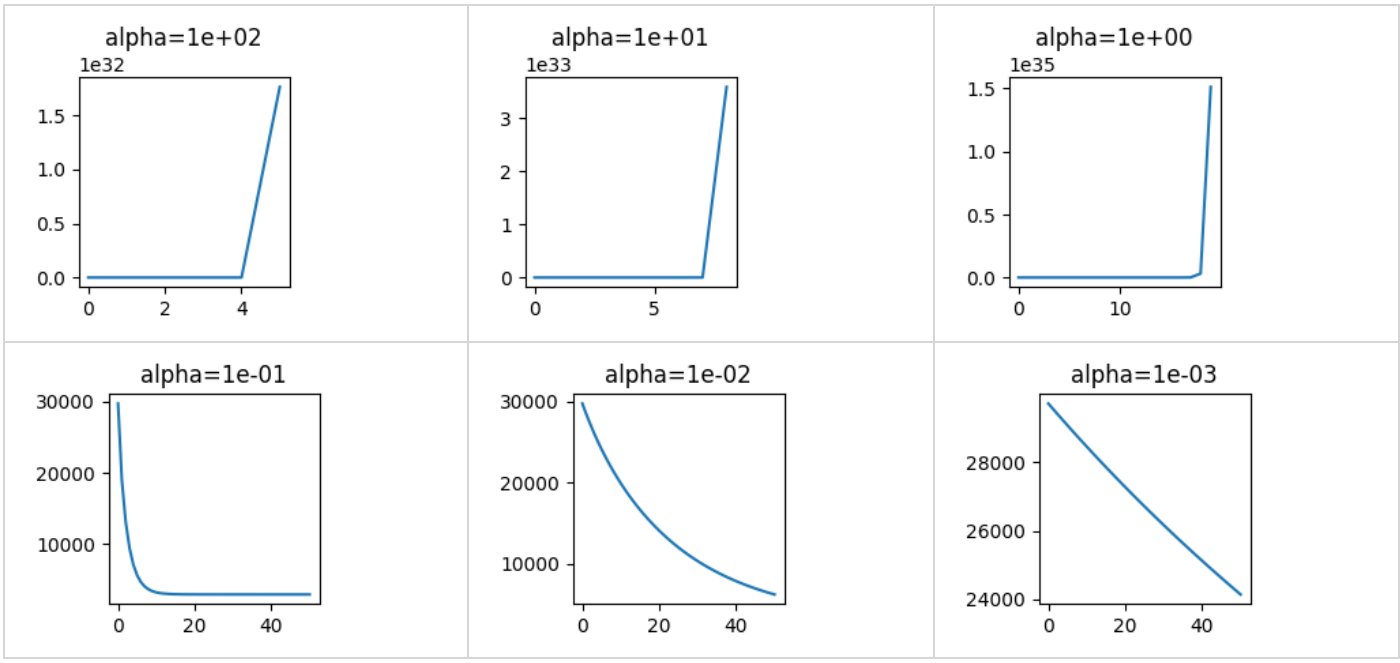
```
pd.DataFrame(
    data = {"cost (in training)": train[0], "cost (in testing)": test[0]},
    index = [f"alpha={alpha:.0e}" for alpha in alphaList])
```

	cost (in training)	cost (in testing)
alpha=1e+02	1.760926e+32	7.406606e+31
alpha=1e+01	1.570402e+22	6.665720e+21
alpha=1e+00	4.620044e+11	2.176643e+11
alpha=1e-01	5.562790e+03	5.840854e+03
alpha=1e-02	2.408285e+04	2.204420e+04
alpha=1e-03	2.907237e+04	2.605022e+04

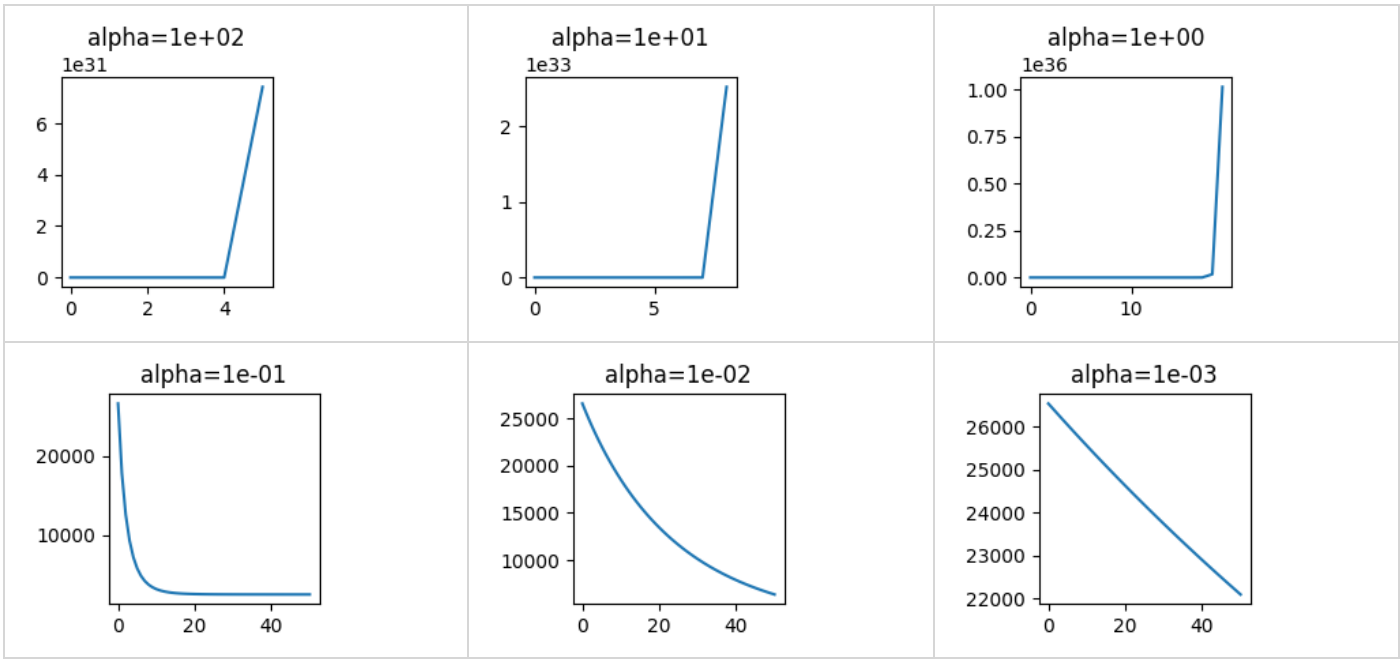
Plotting the cost curves to get a better idea about their convergence or divergence...

```
def plot_costs(costs, title):
  plt.figure(figsize=(2, 2))
  plt.plot(range(len(costs)), costs)
  plt.title(title)


# Plotting cost data for training
for i, alpha in enumerate(alphaList):
  plot_costs(train[1][i], f"alpha={alpha:.0e}")
```




```
# Plotting cost data for testing
for i, alpha in enumerate(alphaList):
  plot_costs(test[1][i], f"alpha={alpha:.0e}")
```



CONCLUSIONS:

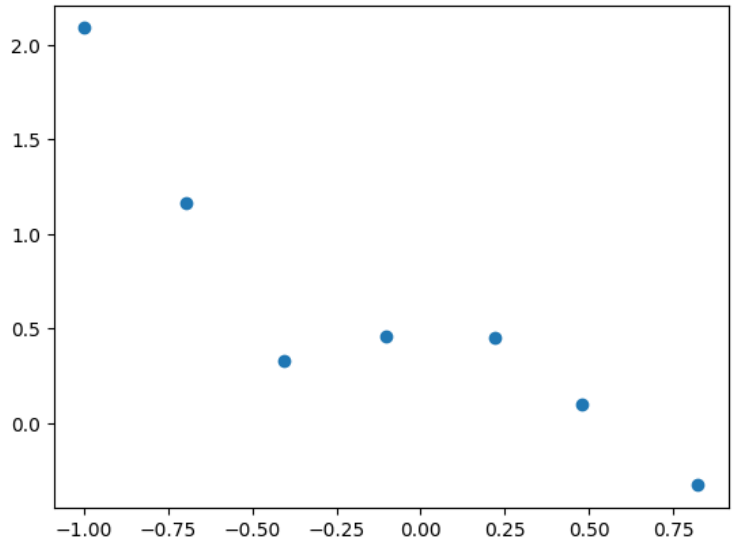
The errors (*mean squared errors between predictions and true values, which is the cost function in our case*) for the training and testing sets change similarly for both sets, since the fit is well-generalised. The α values (*i.e. learning rates used*) determine whether and how fast the errors (*i.e. mean squared errors*) converge or diverge with gradient descent (*note that the gradient being calculated is that of the cost function with respect to the weight vector, i.e. for the the mean squared errors with respect to the weight vector*). From the above result, it is clear that at the magnitude level 10^0 (i.e. 1) is the lowest magnitude level at which the errors diverge. From level 10^{-1} onwards, the errors only converge. We also see that for higher α values at and above the magnitude level 10^0, the error diverges faster the higher α is, whereas for lower α values at and below the magnitude level 10^{-1}, the error converges slower the lower α is.

1.2 Regularized Linear Regression

Introduction

In this exercise, we will be trying to create a model that fits data that is clearly not linear. We will be attempting to fit the data points seen in the graph below:

```
x = torch.tensor([-0.99768, -0.69574, -0.40373, -0.10236,
                  0.22024, 0.47742, 0.82229])
y = torch.tensor([2.0885, 1.1646, 0.3287, 0.46013,
                  0.44808, 0.10013, -0.32952]).reshape(-1, 1)
# This reshaping ensures y is a column vector instead of a row vector
plt.scatter(x, y)
plt.show()
```



In order to fit this data we will create a new hypothesis function, which uses a fifth-order polynomial:

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3 + \theta_4 x_1^4 + \theta_5 x_1^5$$

As we are fitting a small number of points with a high order model, there is a danger of overfitting. To attempt to avoid this we will use regularization. Our cost function becomes:

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} \theta_j^2\right]$$

Here, m is the number of observations (i.e. number of rows in `x`), while n is the number predictors/features (i.e. number of columns in `x`). Now, adjusting variable `x` to include the higher order polynomials...

```
# Adding higher order terms:
x = torch.tensor([list(x**i) for i in range(1, 6)]).T
# We transponse as we want each x^i to be a separate column not a separate row

# Adding bias field as the first column:
x = torch.cat([torch.ones(x.shape[0], 1), x], dim=1)
```

Here, the first column contains the bias term, while the column with index i contains the values of x^{i-1} (note that indices start from 0). So, column with index 1 contains values of x, column with index 2 contains values of x^2, etc.

Question 8

First of all, find the best value of alpha to use in order to optimize best. Next, experiment with different values of λ and see how this affects the shape of the hypothesis. [3 marks]

Optimizing α

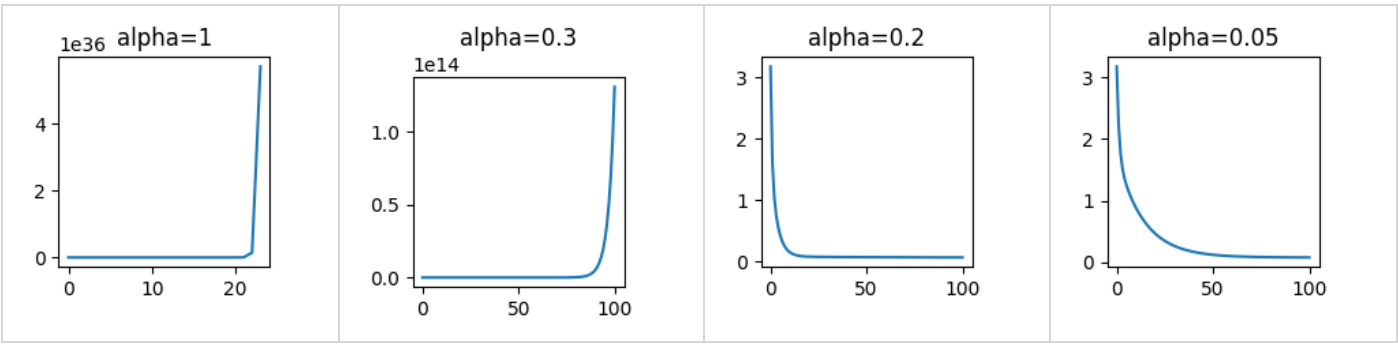
```
# NEW GRADIENT DESCENT STEP LOOP FUNCTION DEFINITION
# (... to perform gradient descent steps iteratively)
def gradient_descent_loop(x, y, alpha, lam, nIterations,
                          getModel=False, getCosts=False):
  # NOTE: nIterations ==> Number of loop iterations

  # Initializing the model:
  model = LinearRegression(x.shape[1])
  # The future list of costs obtained:
  costList = []
  #------------------------
  # We will run this loop for `nIterations` iterations
  for i in range(nIterations+1):
    # Doing gradient descent & getting updated model's predictions:
    prediction = model.forward(x)
    gradient_descent_step(model, x, y, prediction, alpha, lam)
    if getCosts:
      costList.append(mean_squared_error(y, prediction, lam, model.weight))
  #------------------------
  # Returning obtained mean squared error (cost)
  if getModel: return model
  if getCosts: return costList
  return mean_squared_error(prediction, y, lam, model.weight)


# Example of an alpha leading to diverging costs
# (... useful for explaining the alpha-finding algorithm)
print(f"For alpha={2}, cost={gradient_descent_loop(x, y, 2, 0, 2)}")
```

```
For alpha=2, cost=47177.31005859375
```

```
# Plotting cost function curve for different alpha values
# (... useful for explaining the alpha-finding algorithm)
alphaList = [1, 0.3, 0.2, 0.05]
for alpha in alphaList:
  costsGenerated = gradient_descent_loop(x, y, alpha, 0, 100, False, True)
  plot_costs(costsGenerated, f"alpha={alpha}")
```


alpha=1
1e36
4
2
0
0
10
20
alpha=0.3
1e14
1.0
0.5
0.0
0
50
100
alpha=0.2
3
2
1
0
0
50
100
alpha=0.05
3
2
1
0
0
50
100

```
# Finding the alpha that leads to minimum cost
# (for 100 gradient step iterations)

"""
This algorithm will obtain the fastest converging alpha value
for N gradient descent step iterations (here, N is fixed as 100).
"""

# NOTE 1: We are fixing lambda as 0 for now
# NOTE 2: alpha must be positive, since it is the cost convergence factor.

# Small helper function
def switch(a, b): return b, a

# Initializing lower & upper bounds of optimal alpha & their costs
"""
Here, we are using three pieces of prior knowledge
1. alpha must be above zero
2. alpha = 2 leads to diverging cost
3. If an alpha leads to a cost exceeding 3.1786, the cost diverges
"""
L, U = 0, 2
costL = gradient_descent_loop(x, y, L, 0, 100)
costU = gradient_descent_loop(x, y, U, 0, 100)

# Doing the search
for i in range(50):
  if costU < costL:
    costL, costU = switch(costL, costU)
    L, U = switch(L, U)

  cost = gradient_descent_loop(x, y, (L + U)/2, 0, 100)
  if cost <= 3.1786 and cost < costL: L, costL = (L + U)/2, cost
  else: U, costU = (L + U)/2, cost

# Displaying the upper & lower bounds for best alpha (with upper & lower costs)
print(f"Best alpha value lies in [{L:.6f}, {U:.6f}]")
print(f"Lowerst cost lies in [{costL:.6f}, {costU:.6f}]")
```

```
Best alpha value lies in [0.268557, 0.268557]
Lowerst cost lies in [0.064086, 0.064086]
```

TECHNICAL NOTE: **Justifying the above algorithm:**
Assumption on the cost function:

- The cost function is an exponential curve of the form $e^{f(x)}$, where $f(x)$ is a function of the number of gradient descent steps x
- Thus, the cost curve is either monotonically increasing (in case of a diverging curve, where $f(x)$ is positive) or monotonically decreasing (in case of a converging curve, where $f(x)$ is negative)

These observations were made based on the plots of cost function curves for different α values (note again that α is the factor that determines the rate of convergence or divergence of the cost function). Based on these assumptions, consider the following...

Given the fact that our initial cost (calculated when testing the new mean squared function) is a little over 3.1786, any α that does not lead to a converging cost will present cost values larger than $3.1786.5$ after enough (*more than 5*) gradient descent steps (*and will diverge to infinity as the number of iterations of the gradient descent step increases*). Thus, any α value leading to costs less than or equal to 3.1786 after the end of a sufficient number of iterations will certainly converge.

Now, if the cost converges for $\alpha = k$, then it must converge for all $\alpha \leq k$ and for any $\alpha < k$. Furthermore, it is logical to conclude that given N gradient step iterations, if both α_1 and α_2 converge and α_1 leads to a lower cost than α_2, then for N iterations, α converges more toward the minimum cost (which is asymptotic in our cost function).

Let us define the optimal α value as the one that converges the most in N gradient step iterations. Thus, if we have that optimal α value's lower bound is L and the upper bound is U, then, if some α value between L and U, say $\frac{L+U}{2}$ leads to converging cost (i.e. cost less than or equal to 3.1786), then any α value between L and $\frac{L+U}{2}$ is suboptimal (i.e. converges more slowly). Thus, the optimal α must lie in the range $[\frac{L+U}{2}, U]$, which means it is now valid to make the lower bound $\frac{L+U}{2}$.

On the other hand, if $\frac{L+U}{2}$ does not lead to a lower or faster converging cost than L, then the optimal α must lie in the range $[L, \frac{L+U}{2}]$, which means it is now valid to make the upper bound $\frac{L+U}{2}$.

Thus, the algorithm followed by the above code will indefinitely narrow down the bounds of the optimal α value (unless an exact value is found, in which case the lower and upper bounds will become equal).

SIDE NOTE 1: **Degree of convergence for different numbers of iterations**:
N gradient steps have always been specified when talking about optimal α values, and this is due to

the following fact observed about different α values. Some α values converge more than others in fewer iterations, but if the number of iterations are increased, these same α values may converge lesser than other α values it previously outperformed. For example, $\alpha = 0.25$ converges more than $\alpha = 0.27$ for 10 gradient step iterations, but $\alpha = 0.27$ converges more than $\alpha = 0.25$ for 100 iterations. For this reason, the "optimal" α was defined with respect to the number of gradient descent step iterations.

SIDE NOTE 2:
Based on different observations with different α values and number of gradient step iterations (the results are not presented in this document), a conjecture can be made (for the given cost function) that if both α_1 and α_2 lead to converging costs, and if $\alpha_1 > \alpha_2$, then for a sufficiently high (but finite) number of gradient step iterations, α_1 will always converge more than α_2.

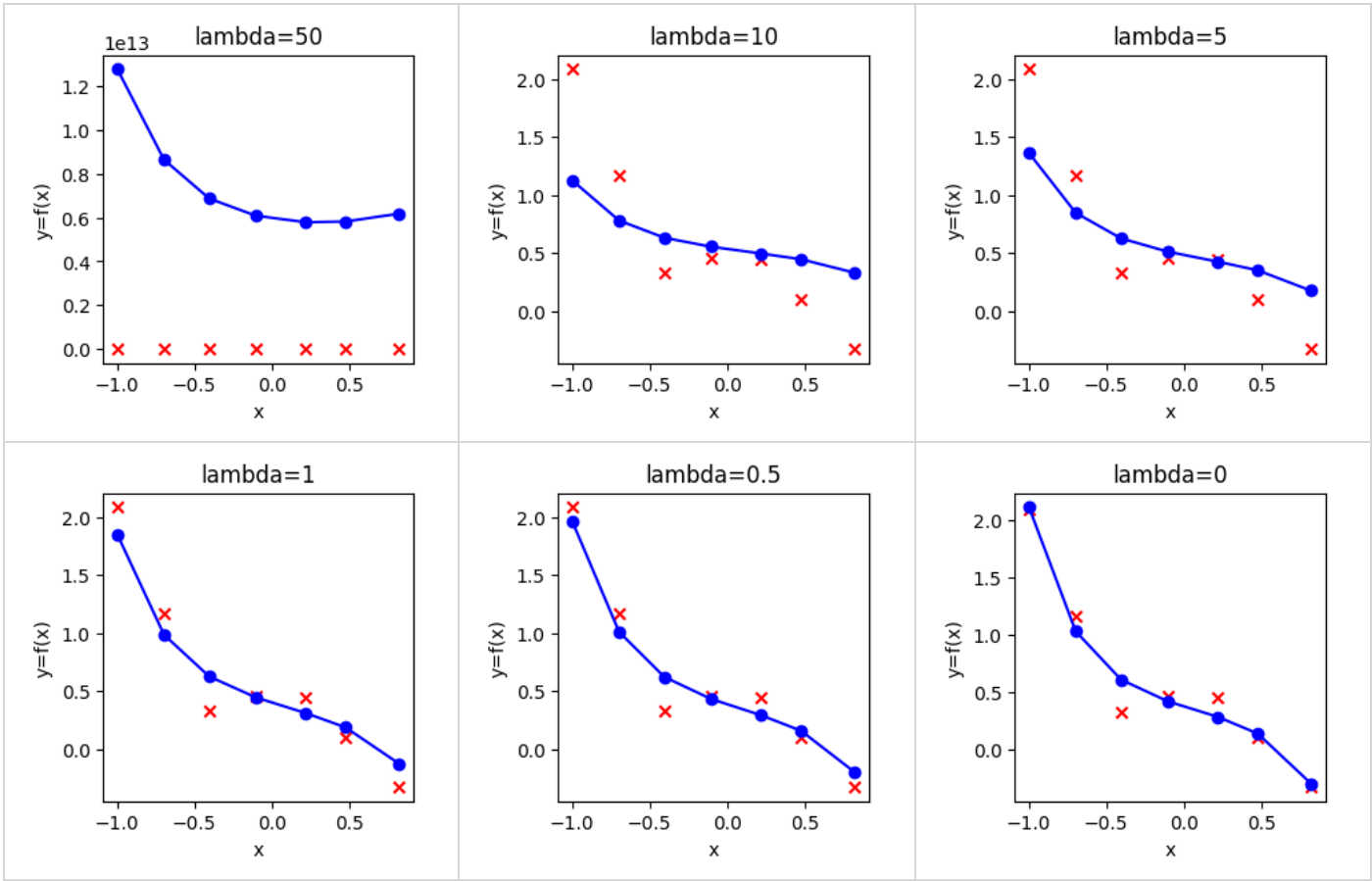
Experimenting with different λ values

```
# Generating data for different lambda values
"""
NOTE:
lambda must be positive, since it is the factor of the extra cost added to
the cost function that penalizes high weight magnitudes that tend to overfit
the model.
"""
lamList, predictionList = [50, 10, 5, 1, 0.5, 0], []
for lam in lamList:
  # Training the model with the given lambda value
  model = gradient_descent_loop(x, y, 0.2685, lam, 100, True)
  # Getting & saving predictions
  prediction = model.forward(x)
  predictionList.append([p.item() for p in prediction])
```

Plotting the hypothesis curves for different λ values...

```
# Function to plot a given set of predictions against true values
def plot_hypothesis(prediction, title):
  plt.figure(figsize=(3, 3))
  # Scatter plot to see the original data
  plt.scatter(x[:, 1], y, c='red', marker='x', label='Ground truth')
  # NOTE: x[:, 0] gives the 1st column, which consists of the original x values
  plt.plot(x[:, 1], prediction, c='blue', marker='o', label='Prediction')
  plt.title(title), plt.xlabel('x'), plt.ylabel('y=f(x)')


# Creating a plot per lambda value
for i in range(len(lamList)):
  # Using the `plot_costs` function defined before
  plot_hypothesis(predictionList[i], f"lambda={lamList[i]}")
```



CONCLUSIONS: **The effect of different λ values**:

In general, the higher the λ value (i.e. the regularization parameter), the less exact or accurate the fit of the predictions is with respect to the training data. We see that up to a point, the higher the λ value, the flatter and further away on average from the true values the hypothesis curve gets, reflecting the fact that all the errors (cost function values) are pushed to high values due to the regularization term. Beyond a point, i.e. for sufficiently high λ values, the fit becomes completely inaccurate and even

changes its shape from a flat curve vaguely passing through the points to a curve away from any point in the same direction.

CONCEPTUAL NOTE: **Finding the right λ value**:
λ, as the regularization parameter, is the value that controls the level of overfitting, i.e. the difference between the costs (mean squared errors in our case) of the training and testing sets. In this context, the testing set is the validation set. In our case, there is no validation set obtainable, hence we cannot determine the λ value required to correct for overfitting (if at all).