Name: Kiana Hadysadegh
Student ID: 230632224
Assignment Number 1(Regression)
Module number: ECS708P

Q5. 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]

From the weight values, we can find the importance or influence of each variable on the target variable. The sign of the weight and its magnitude for each variable show the direction and strength of its relationship with the target variable, respectively.

Our weights are
`tensor([[ 1.9400, -11.4488, 26.3047, 16.6306, -9.8810, -2.3179, -7.6995, 8.2121, 21.9769, 2.6065, 153.7365]])`

So -11.488 and 26.3047 are for gender and BMI variables in turn.

The weight values for these two variables are non-zero so they influence blood sugar levels.
BMI weight value is positive so an increase in this variable is associated with increased blood sugar levels and vice versa. So having a higher BMI is associated with higher blood sugar levels.
Since the gender weight value is negative, an increase in this variable is associated with a decrease in blood sugar levels. So gender influences blood sugar levels and being female is associated with lower blood sugar levels compared to being male. (male = 1 , female=2)
The magnitude of BMI weight is more than gender. So, changes in BMI have more influence on blood sugar levels compared to gender. Hence, the relation between BMI and Blood sugar level is stronger than it is between gender and Blood sugar level.

The estimated blood sugar levels for the given data are 43.5294, 232.2310 in turn.

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?

```
### your code here
test_prediction = model(x_test)
cost = mean_squared_error(test_prediction, y_test)
print(cost)
```

```
tensor(2885.6191)
```

The MSE error on the test dataset (2885.6191) is significantly lower than that on the training dataset (29711.3223). So, the model performs better on the test dataset. It has learned the patterns in data and it generalized well to new data(test data) and it has a good fit.

Overfitting happens when a model learns the training dataset too well. Overfitting causes weak performance on new and unseen data since it captures even noises and random fluctuations in the training dataset. In an overfitting situation, the error in training data is too lower than in test (unseen) data. So in this model, there is no sign of overfitting.

Underfitting happens when a model does not train well and can not capture underlying patterns in the training dataset. So, the model has weak performance in training and test datasets and has high errors in both of them. So in this model, there is no sign of underfitting too.

Q6. 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]

I defined a function (calculate_error) to test the influence of a given learning rate on training and test costs, and then I generated different learning rates from 0.01 to 0.3 and saved training and test costs into lists.

```
def calculate_error(lr):

  cost_lst = list()
  model = LinearRegression(x_train.shape[1])

  for it in range(100):
    prediction = model(x_train)
    cost = mean_squared_error(y_train, prediction)
    cost_lst.append(cost)
    gradient_descent_step(model, x_train, y_train, prediction, lr)

  test_prediction = model(x_test)
  test_cost = mean_squared_error(y_test, test_prediction)
  print("Learning_rate {}\n Min_train_cost {}\n test_cost {}\n--------\n".format(lr, min(cost_lst), test_cost ))
  return min(cost_lst), test_cost
```

```
### your code here
learning_rates = np.arange(0.01, 0.3 + 0.01, 0.01)
train_costs = []
test_costs = []
for lr in learning_rates:
  min_train_cost, test_cost = calculate_error(lr)
  test_costs.append(test_cost)
  train_costs.append(min_train_cost)
```

Learning_rate 0.01
 Min_train_cost 3356.77734375
 test_cost 3431.068359375

Learning_rate 0.02
 Min_train_cost 2906.434814453125
 test_cost 2906.97900390625

Learning_rate 0.03
 Min_train_cost 2897.025634765625
 test_cost 2885.867431640625

Learning_rate 0.04
 Min_train_cost 2895.79345703125
 test_cost 2884.759521484375

Learning_rate 0.05
 Min_train_cost 2894.800048828125
 test_cost 2884.92236328125

Learning_rate 0.060000000000000005
 Min_train_cost 2893.853515625
 test_cost 2885.09912109375

Learning_rate 0.06999999999999999
 Min_train_cost 2892.943115234375
 test_cost 2885.2470703125

Learning_rate 0.08
 Min_train_cost 2892.066162109375
 test_cost 2885.381591796875

Learning_rate 0.09
 Min_train_cost 2891.220703125
 test_cost 2885.5048828125

Learning_rate 0.09999999999999999
 Min_train_cost 2890.406494140625
 test_cost 2885.619140625

Learning_rate 0.11
 Min_train_cost 2889.62109375
 test_cost 2885.7255859375

Learning_rate 0.12
 Min_train_cost 2888.864990234375
 test_cost 2885.824951171875

Learning_rate 0.13
 Min_train_cost 2888.13525390625
 test_cost 2885.918701171875

Learning_rate 0.14
 Min_train_cost 2887.43212890625
 test_cost 2886.0087890625

Learning_rate 0.15000000000000002
 Min_train_cost 2886.754150390625
 test_cost 2886.0966796875

Learning_rate 0.16
 Min_train_cost 2886.10009765625
 test_cost 2886.1826171875

Learning_rate 0.17
 Min_train_cost 2885.469482421875
 test_cost 2886.267822265625

Learning_rate 0.18000000000000002
 Min_train_cost 2884.86181640625
 test_cost 2886.352783203125

Learning_rate 0.19
 Min_train_cost 2884.27587890625
 test_cost 2886.439208984375

Learning_rate 0.2
 Min_train_cost 2883.710693359375
 test_cost 2886.52587890625

Learning_rate 0.21000000000000002
 Min_train_cost 2883.165771484375
 test_cost 2886.61474609375

Learning_rate 0.22
 Min_train_cost 2882.640625
 test_cost 2886.70458984375

Learning_rate 0.23
 Min_train_cost 2882.134033203125

test_cost 2886.796630859375

Learning_rate 0.24000000000000002
Min_train_cost 2881.6455078125
test_cost 2886.8896484375

Learning_rate 0.25
Min_train_cost 2907.636962890625
test_cost 2907.63525390625

Learning_rate 0.26
Min_train_cost 5799.37841796875
test_cost 151478912.0

Learning_rate 0.27
Min_train_cost 7102.2880859375
test_cost 280081813471232.0

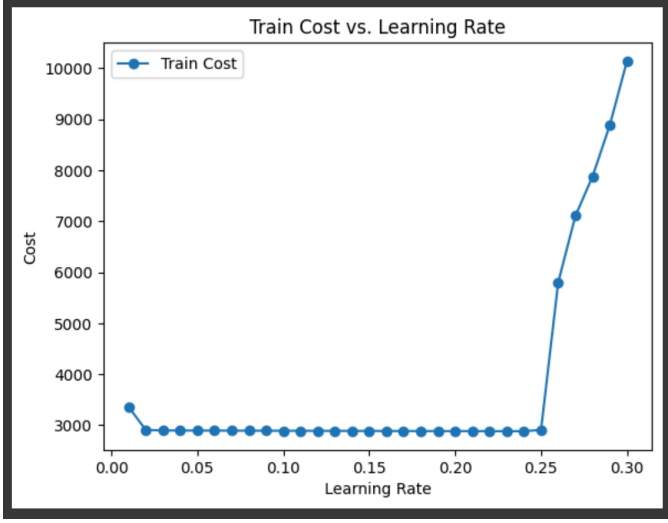
Learning_rate 0.28
Min_train_cost 7876.6005859375
test_cost 1.9601183783972032e+20

Learning_rate 0.29000000000000004
Min_train_cost 8879.5029296875
test_cost 5.869968407654236e+25

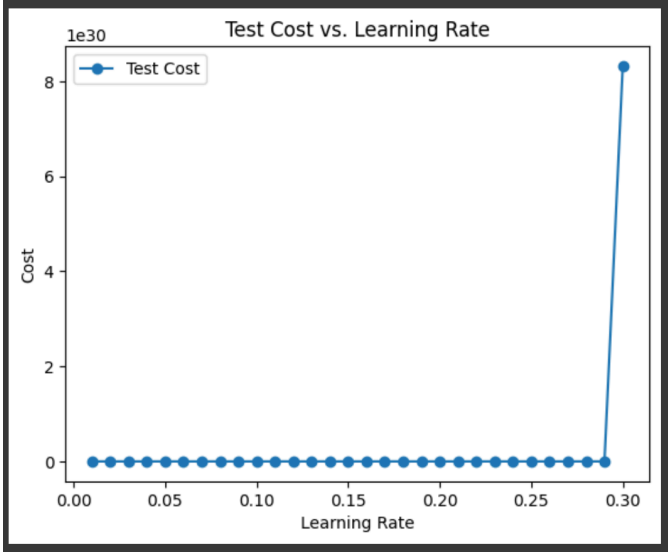
Learning_rate 0.3
Min_train_cost 10136.5400390625
test_cost 8.31950712524568e+30

Then I plotted train and test costs for different learning rates.
From the graph, we can see both training and test errors decrease by increasing the learning rate from 0 to 0.25. After 0.25 the training and test cost increased. This is because by increasing the learning rate value after 0.25, we take larger steps and skip the minimum.

```
plt.plot(learning_rates, train_costs, marker='o', label='Train Cost')

plt.xlabel('Learning Rate')
plt.ylabel('Cost')
plt.title('Train Cost vs. Learning Rate')
plt.legend()
plt.show()
```



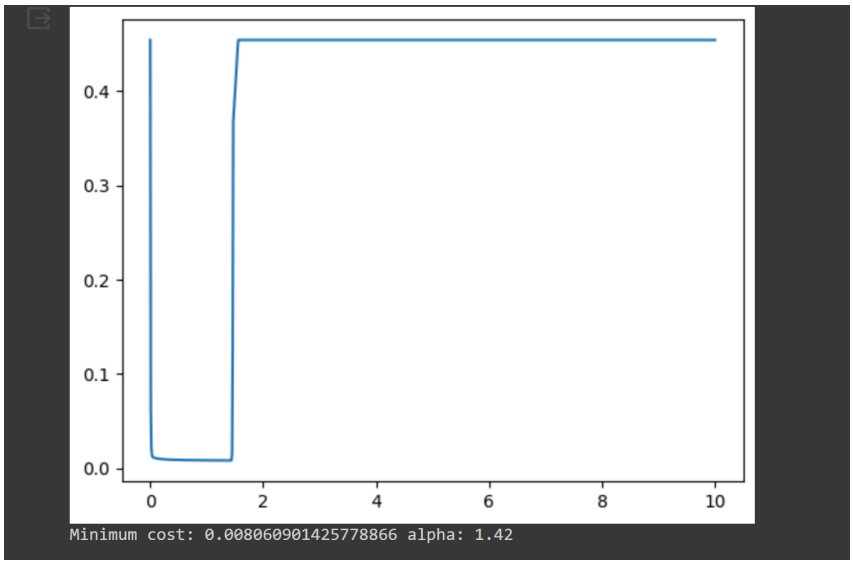

```
plt.plot(learning_rates, test_costs, marker='o', label='Test Cost')


plt.xlabel('Learning Rate')
plt.ylabel('Cost')
plt.title('Test Cost vs. Learning Rate')
plt.legend()
plt.show()
```



Q8. 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]

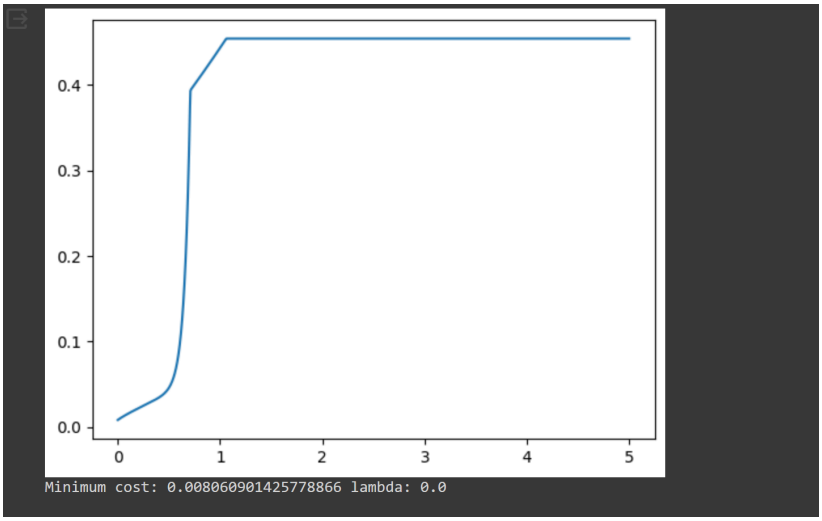
To find the best alpha, we repeat the process for each alpha and save the minimum cost in a list. At the end, we check which alpha gives us the lowest cost in the list and pick that alpha:

```
[105] #FINDING ALPHA
def find_alpha():
  alphas = np.arange(0, 10 + 0.01, 0.01) # select an appropriate alpha
  lam = 0 # select an appropriate lambda
  total_costs = []
  for alpha in alphas:
    cost_lst = list()
    model = LinearRegression(x3.shape[1])
    for it in range(100):
      prediction = model(x3)
      cost = mean_squared_error(y, prediction, lam, model.weight)
      cost_lst.append(cost)
      gradient_descent_step(model, x3, y, prediction, alpha, lam)

    display.clear_output(wait=True)
    total_costs.append(min(cost_lst))

  plt.plot(alphas, total_costs)
  plt.show()
  # print(model.weight)
  print('Minimum cost: {} alpha: {}'.format(min(total_costs), alphas[total_costs.index(min(total_costs))]))
find_alpha()
```



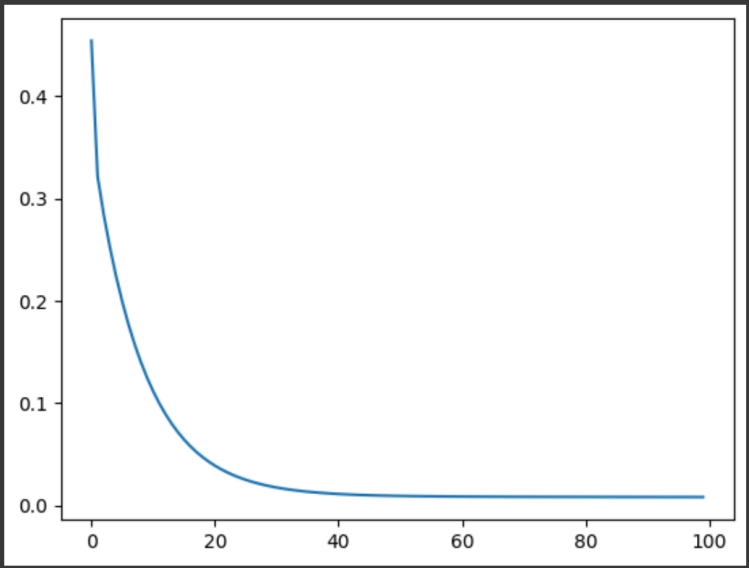
Then based on the best alpha I use the below code to find the best lambda: Now we save the minimum cost for each lambda in the list. In the end, we check which lambda gives us the lowest cost in the list and pick that one:

```
106] #FINDING LAMBDA BASED ON ALPHA
    def find_lambda(alpha):
      lambdas = np.arange(0, 5 + 0.001, 0.001) # select an appropriate lambda
      total_costs = []
      for lam in lambdas:
        cost_lst = list()
        model = LinearRegression(x3.shape[1])
        for it in range(100):
          prediction = model(x3)
          cost = mean_squared_error(y, prediction, lam, model.weight)
          cost_lst.append(cost)
          gradient_descent_step(model, x3, y, prediction, alpha, lam)

        display.clear_output(wait=True)
        total_costs.append(min(cost_lst))

      plt.plot(lambdas, total_costs)
      plt.show()
      # print(model.weight)
      print('Minimum cost: {} lambda: {}'.format(min(total_costs), lambdas[total_costs.index(min(total_costs))]))

    find_lambda(1.42)
```



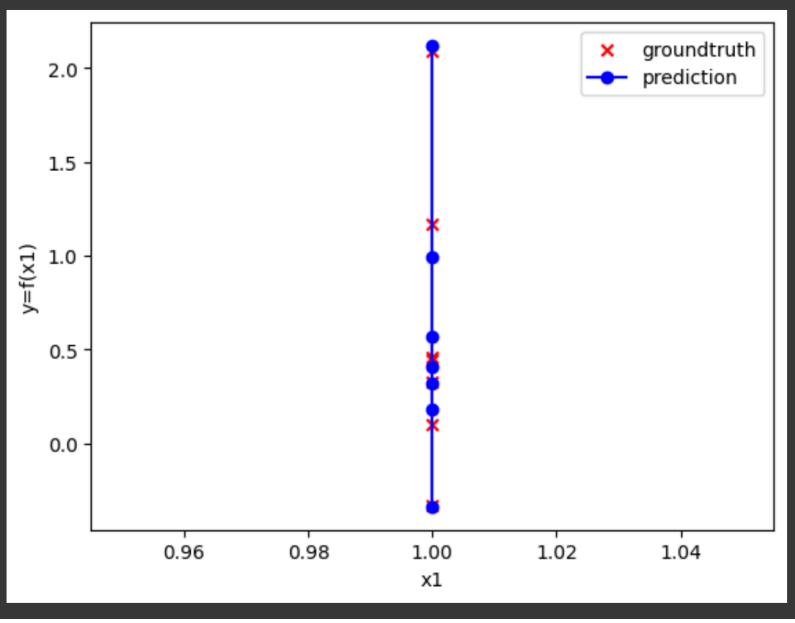
Figures are based on alpha 1.42, lambda 0:

```
cost_lst = list()
model = LinearRegression(x3.shape[1])
alpha = 1.42 # select an appropriate alpha
lam = 0 # select an appropriate lambda
for it in range(100):
  prediction = model(x3)
  cost = mean_squared_error(y, prediction, lam, model.weight)
  cost_lst.append(cost)
  gradient_descent_step(model, x3, y, prediction, alpha, lam)
display.clear_output(wait=True)
plt.plot(list(range(it+1)), cost_lst)
plt.show()
print(model.weight)
print('Minimum cost: {}'.format(min(cost_lst)))
```

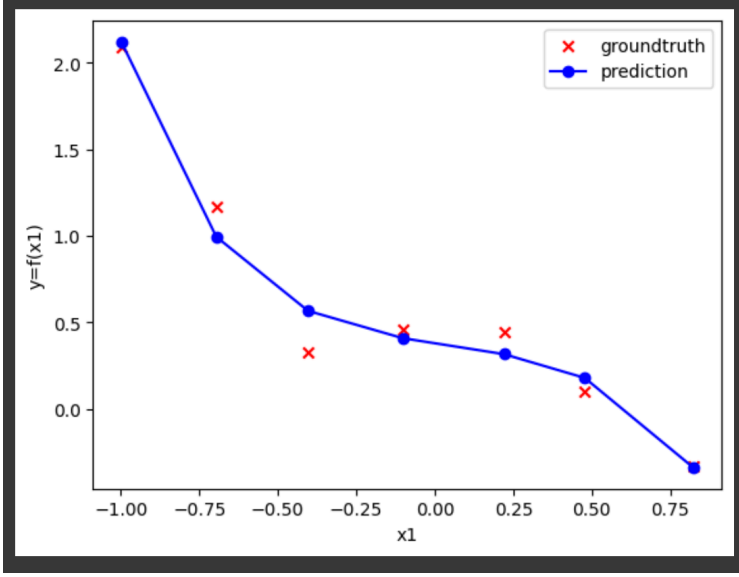



```
Parameter containing:
tensor([[ 0.3800, -0.2672,  0.0976, -0.8406,  0.1317, -0.4146]])
Minimum cost: 0.008060901425778866
```

```
plt.scatter(x3[:, 0], y, c='red', marker='x', label='groundtruth')
outputs = model(x3)
plt.plot(x3[:, 0], outputs, c='blue', marker='o', label='prediction')
plt.xlabel('x1')
plt.ylabel('y=f(x1)')
plt.legend()
plt.show()
```



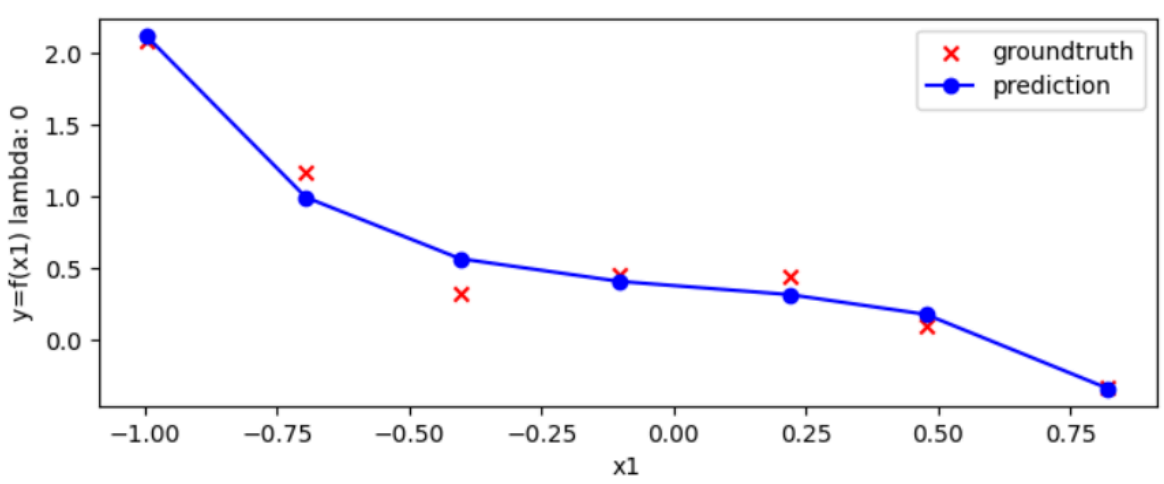

```
plt.scatter(x3[:, 1], y, c='red', marker='x', label='groundtruth')
outputs = model(x3)
plt.plot(x3[:, 1], outputs, c='blue', marker='o', label='prediction')
plt.xlabel('x1')
plt.ylabel('y=f(x1)')
plt.legend()
plt.show()
```

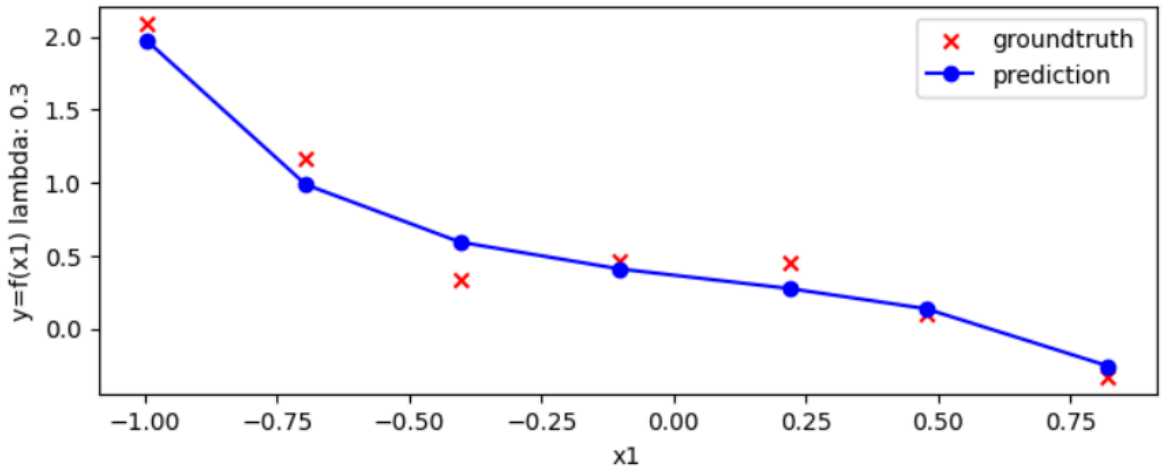


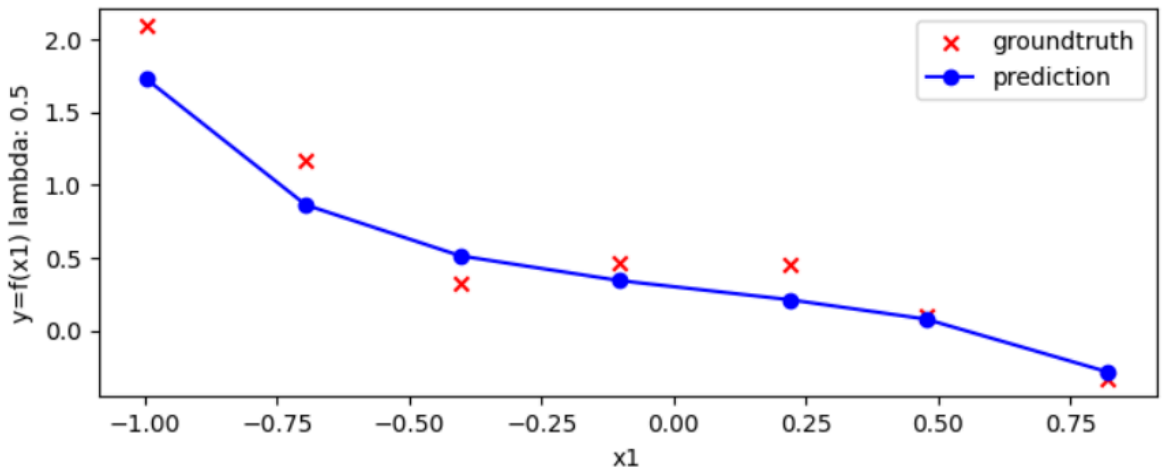
I defined a function called examine_lambda to see the influence of different lambdas on the model's fit. In this function, we test different values of lambda and plot the fit of the model:

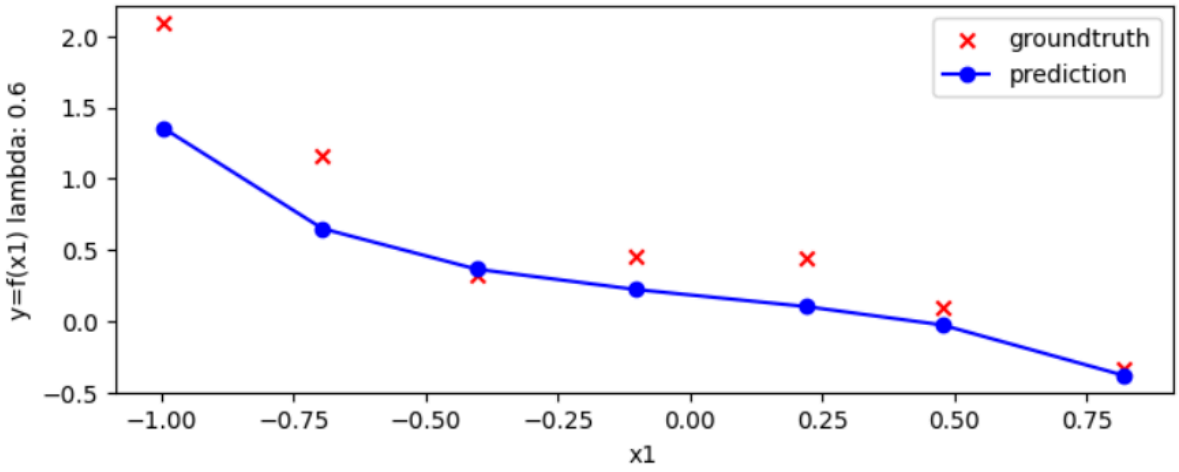
```python
def examine_lambda(lam, ax):
  cost_lst = list()
  model = LinearRegression(x3.shape[1])
  alpha = 1.42 # select an appropriate alpha
  for it in range(100):
    prediction = model(x3)
    cost = mean_squared_error(y, prediction, lam, model.weight)
    cost_lst.append(cost)
    gradient_descent_step(model, x3, y, prediction, alpha, lam)
  display.clear_output(wait=True)
  ax.scatter(x3[:, 1], y, c='red', marker='x', label='groundtruth')
  outputs = model(x3)
  ax.plot(x3[:, 1], outputs, c='blue', marker='o', label='prediction')
  ax.set_xlabel('x1')
  ax.set_ylabel('y=f(x1) lambda: {}'.format(lam))
  ax.legend()
```

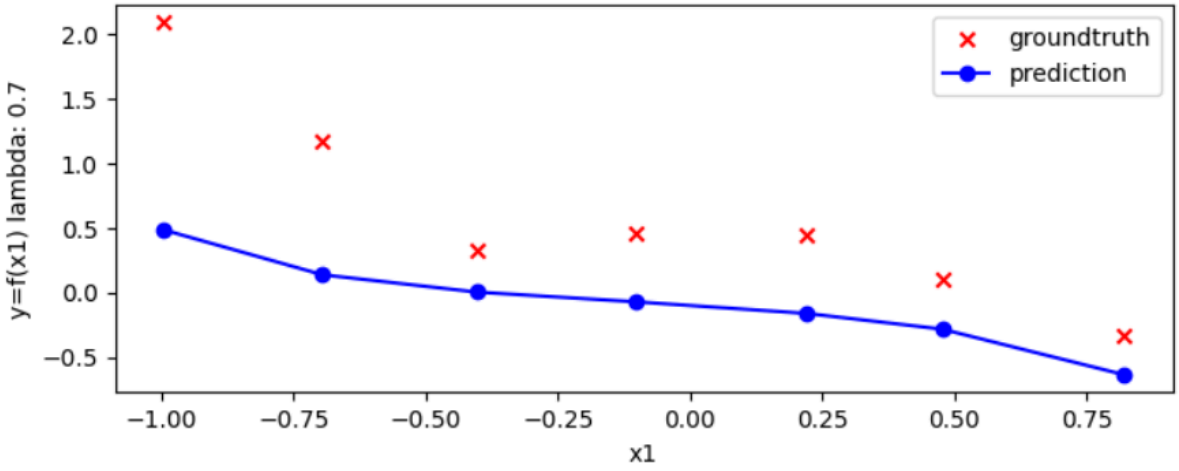
```python
lambdas = [0, 0.3, 0.5, 0.6, 0.7, 0.9, 1, 1.1, 2]

num_plots = len(lambdas)
num_rows = num_plots
num_cols = 1
fig, axes = plt.subplots(num_rows, num_cols, figsize=(8, 4 * num_plots))
if num_rows == 1:
    axes = [axes]

for i, input_value in enumerate(lambdas):
    examine_lambda(input_value, axes[i])

plt.subplots_adjust(hspace=0.4)
plt.show()
```

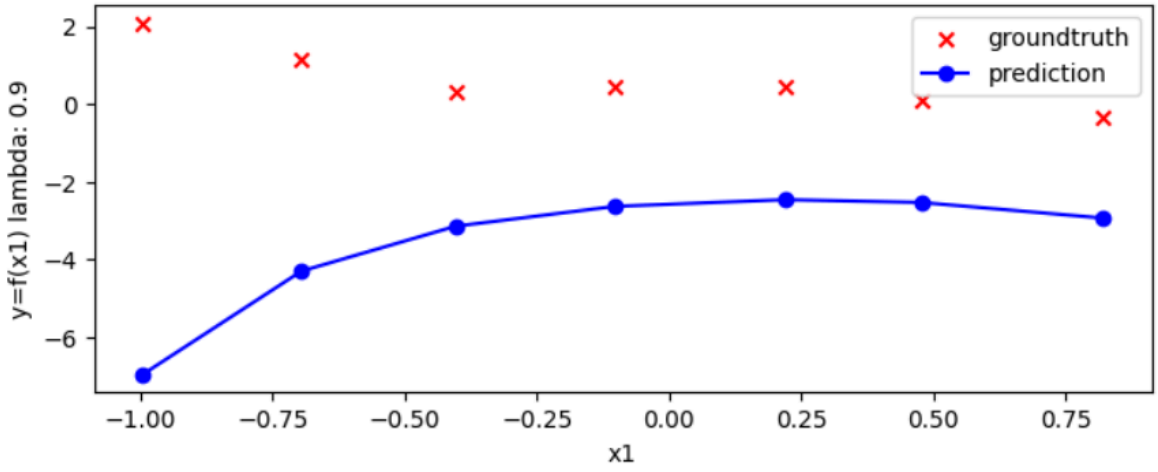

groundtruth
prediction
y=f(x1) lambda: 0
x1
2.0
1.5
1.0
0.5
0.0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75

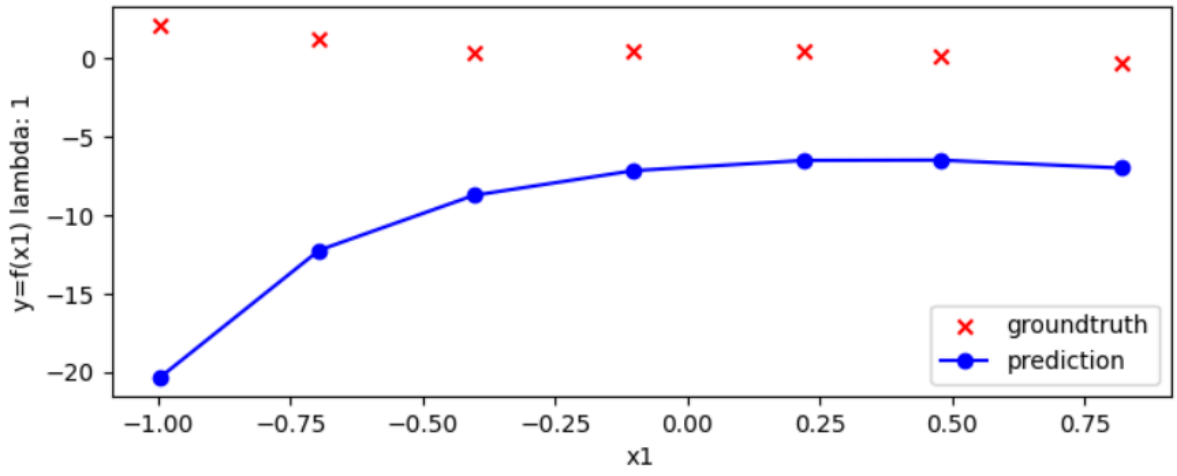

groundtruth
prediction
y=f(x1) lambda: 0.3
x1
2.0
1.5
1.0
0.5
0.0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75

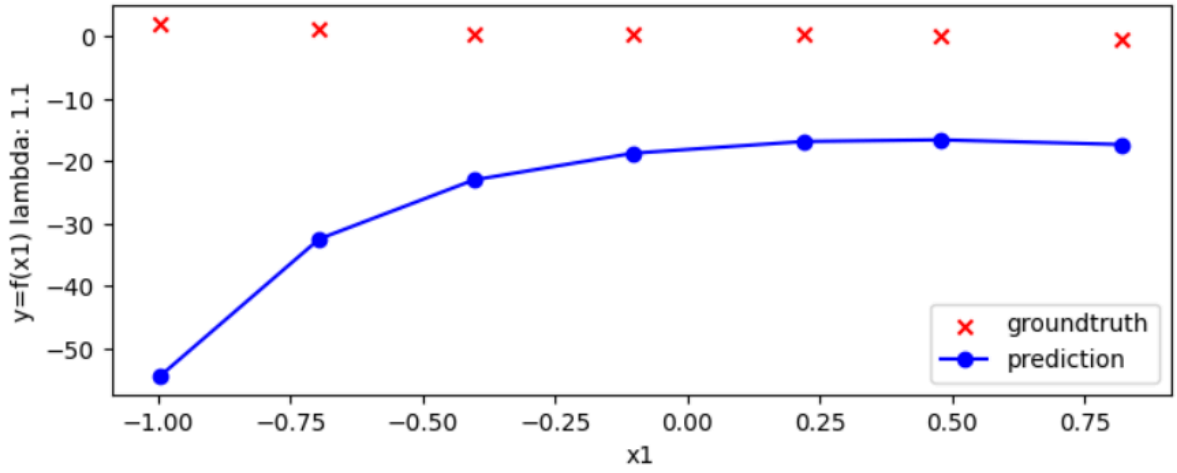

groundtruth
prediction
y=f(x1) lambda: 0.5
x1
2.0
1.5
1.0
0.5
0.0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75

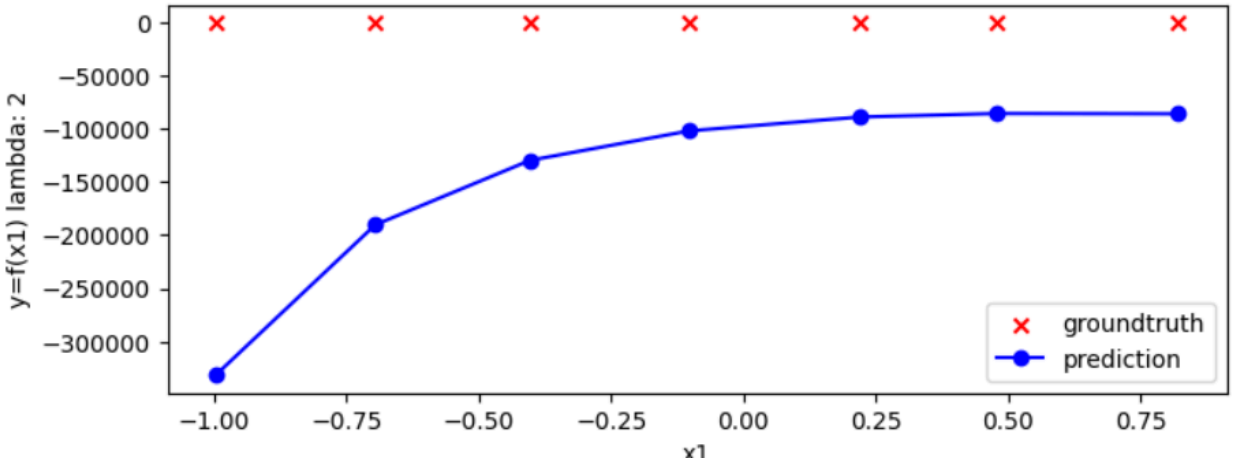

groundtruth
prediction
y=f(x1) lambda: 0.6
x1
2.0
1.5
1.0
0.5
0.0
−0.5
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75


groundtruth
prediction
y=f(x1) lambda: 0.7
x1
2.0
1.5
1.0
0.5
0.0
−0.5
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75


groundtruth
prediction
y=f(x1) lambda: 0.9
x1
2
0
−2
−4
−6
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75


y=f(x1) lambda: 1
0
−5
−10
−15
−20
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
x1
groundtruth
prediction


y=f(x1) lambda: 1.1
0
−10
−20
−30
−40
−50
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
x1
groundtruth
prediction


y=f(x1) lambda: 2
0
−50000
−100000
−150000
−200000
−250000
−300000
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
x1
groundtruth
prediction

It is obvious that by decreasing the lambda the fit of the model gets better. So lambdas between 0 and 0.5 fit well the data and regularized it. For the bigger lambdas, we see that the model does not fit well.