# **REPORT**

ECS708P (Machine Learning) - Lab 3 - Regression (Assignment 1 part 1)
Soumya Snigdha Kundu - 221001323

## Q5) What conclusion if any can be drawn from the weight values?

```
Parameter containing:
tensor([[  1.9400, -11.4488,  26.3047,  16.6306,  -9.8810,  -2.3179,  -7.6995,
           8.2121,  21.9769,   2.6065, 153.7365]])
Minimum Training cost: tensor([2890.4067])
```

These are my acquired weight values. I can immediately point out that the BMI, with an associated weight of 26.3047 values heavily into our final prediction. Mail and female values exactly oppositely with a value of negative (-) 11.4488. The other values are accounted similarly, but just with less importance, as their weights are comparatively lower.

## How does gender and BMI affect blood sugar levels?

As gathered from the weight, higher the BMI, more chance the person has for an elevated blood sugar level. They are **directly proportional** to each other.

## What are the estimated blood sugar levels for the below examples?

The below screenshot has the values, attached with the code for the answers.

**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]

| 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 |

```
[14] ### your code here
     first_entry = torch.tensor([[25, 2, 18, 79, 130, 64.8, 61, 2, 4.1897, 68]])
     second_entry = torch.tensor([[50, 1, 28, 103, 229, 162.2, 60, 4.5, 6.107, 124]])

     norm_first = norm_set(first_entry, X_mean, X_std)
     norm_second = norm_set(second_entry, X_mean, X_std) #Normalising the entries

     ones_first = torch.cat([norm_first, torch.ones(norm_first.shape[0], 1)], dim=1)
     ones_second = torch.cat([norm_second, torch.ones(norm_second.shape[0], 1)], dim=1)

     print("Prediction on the first entry: ", model(ones_first))
     print("Prediction on the second entry: ", model(ones_second))

     Prediction on the first entry:  tensor([[43.5294]])
     Prediction on the second entry:  tensor([[232.2310]])
```

### Now estimate the error on the test set.

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?

```
[15] prediction = model(x_test)
     mean_squared_error(y_test, prediction)

     tensor([2885.6194])
```

```
print('Minimum Training cost: {}'.format(min(cost_lst)))
print("This is the test Error:", mean_squared_error(y_test, prediction))

Minimum Training cost: tensor([2890.4067])
This is the test Error: tensor([2885.6194])
```

### Is the error on the test set comparable to that of the train set?

**Yes,** it is comparable to that of the train set as the values are quite close to each other.

### What can be said about the fit of the model?

Based on the given information, The model has a **good fit** because:

A. <u>errors from the training and testing sets are comparable.</u>
B. There is <u>clearly no overfitting</u> as the errors are comparable.
C. No provided baseline, and the model is maintaining its cost as the number of epochs increases (in 100s).
D. There is also a significant decrease in the cost as the alpha value increases.

### When does a model over/under fits?

One notices overfitting when the model does well on the training data but does not perform well on the testing data. It hints towards memorisation rather than learning. Hence, results in poor generalizability.

When the model performs poorly on the train data and is unable to capture the connection between the features and targets, the model is underfitting.

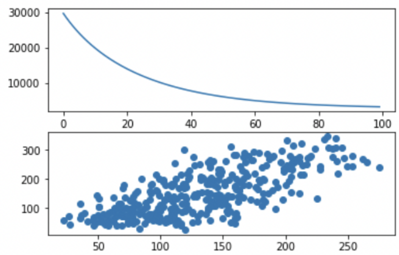
**Q6.** Try the code with a number of learning rates that differ by orders of magnitude and record the error of the training and test sets.

| Alpha Value | Training Error | Test Error |
|---|---|---|
| `0.1` | `2890.4067` | `2885.6194` |
| `0.01` | `3356.7759` | `3431.0684` |
| `0.001` | `20040.5820` | `18534.3066` |
| `0.0001` | `28468.0801` | `25530.2246` |

<u>Sample Code/Graph example with different alphas</u>

ALPHA = 0.01

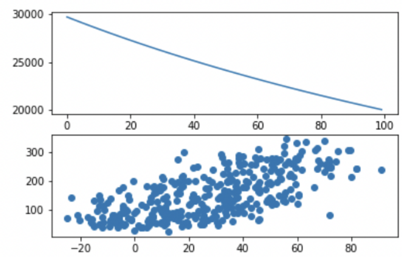
```
### your code here
#Train Value

cost_lst = list()
model = LinearRegression(x_train.shape[1])
alpha = .01
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, alpha)
fig, axs = plt.subplots(2)
axs[0].plot(list(range(it+1)), cost_lst)
axs[1].scatter(prediction, y_train)
plt.show()
print(model.weight)
print('Minimum Training cost: {}'.format(min(cost_lst)))

prediction = model(x_test)
print("Test Error:", mean_squared_error(y_test, prediction))
```



```
Parameter containing:
tensor([[  2.0607,  -9.1801,  23.7039,  15.3157,  -2.9941,  -5.4496, -10.2138,
          6.7962,  17.5502,   5.7594, 133.3481]])
Minimum Training cost: tensor([3356.7759])
Test Error: tensor([3431.0684])
```

ALPHA = 0.001

```
### your code here
#Train Value

cost_lst = list()
model = LinearRegression(x_train.shape[1])
alpha = .001
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, alpha)
fig, axs = plt.subplots(2)
axs[0].plot(list(range(it+1)), cost_lst)
axs[1].scatter(prediction, y_train)
plt.show()
print(model.weight)     def mean_squared_error(y_true: torch.Tensor, y_pred: torch.
print('Minimum Traini   View source

prediction = model(x_   <function mean_squared_error at 0x7fd55ae29ef0>
print("Test Error:", mean_squared_error(y_test, prediction))
```



```
Parameter containing:
tensor([[ 1.9955, -0.5323,  7.3662,  5.3021,  1.6007,  0.9778, -4.3713,  4.4398,
          6.3970,  4.2180, 27.8929]])
Minimum Training cost: tensor([20040.5820])
Test Error: tensor([18534.3066])
```

What do you observe on the training error? What about the error on the test set?

<u>**As the alpha value increases exponentially, the training error increases as well.**</u> It tends to perform worse and worse, especially a huge jump when the alpha decreases from `0.01` to `0.001`. This <u>nature is mimicked from the testing error as well.</u>

# Context image for question 8

```
[22] x = torch.tensor([-0.99768, -0.69574, -0.40373, -0.10236, 0.22024, 0.47742, 0.82229]).reshape(-1, 1)
     x3 = torch.cat([x, x**2, x**3, x**4, x**5, torch.ones(x.shape[0], 1)], dim=1)
     print(x3)

     tensor([[-9.9768e-01,  9.9537e-01, -9.9306e-01,  9.9075e-01, -9.8845e-01,
               1.0000e+00],
             [-6.9574e-01,  4.8405e-01, -3.3678e-01,  2.3431e-01, -1.6302e-01,
               1.0000e+00],
             [-4.0373e-01,  1.6300e-01, -6.5807e-02,  2.6568e-02, -1.0726e-02,
               1.0000e+00],
             [-1.0236e-01,  1.0478e-02, -1.0725e-03,  1.0978e-04, -1.1237e-05,
               1.0000e+00],
             [ 2.2024e-01,  4.8506e-02,  1.0683e-02,  2.3528e-03,  5.1818e-04,
               1.0000e+00],
             [ 4.7742e-01,  2.2793e-01,  1.0882e-01,  5.1952e-02,  2.4803e-02,
               1.0000e+00],
             [ 8.2229e-01,  6.7616e-01,  5.5600e-01,  4.5719e-01,  3.7595e-01,
               1.0000e+00]])
```

**Q7.** Update the cost and gradient descent methods to use the regularised cost, as shown above. [4 marks]

Note that the punishment for having more terms is not applied to the bias. This means that we use a different update technique for the partial derivative of $\theta_0$, and add the regularization to all of the others:

$\theta_j = \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}, j = 0$

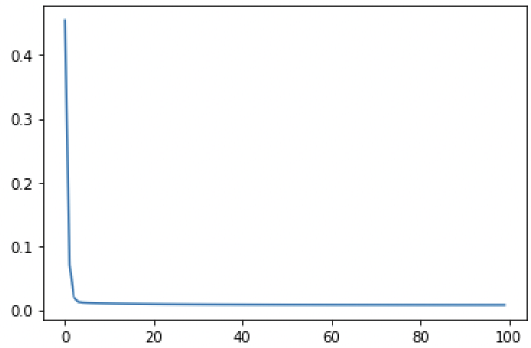
$\theta_j = \theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}, j > 0$

```
[23] def mean_squared_error(y_true: torch.Tensor, y_pred: torch.Tensor, lam: float, theta: torch.tensor) -> torch.Tensor:
       ### your code here
       m = x.shape[0]
       # cost = sum(((y_pred - y_true)**2)/len(y_pred))
       cost_new = (1/(2*m)) * (torch.sum((y_pred-y_true)**2) + (lam * torch.sum(theta**2)))

       return cost_new

     def gradient_descent_step(model: nn.Module, X: torch.Tensor, y: torch.Tensor, y_pred: torch.Tensor, lr: float, lam) -> None:
       weight = model.weight
       N = X.shape[0]
       ### your code here
       ###
       call = (lr/N) * torch.sum((2*(y_pred - y))*X,dim=0)
       weight[:,0] -= call[0]
       weight[:,1:] = weight[:,1:]*(1-(lr*lam)/N) - call[1:]
       # print(weight.shape)

       model.weight = nn.Parameter(weight, requires_grad=False)
```

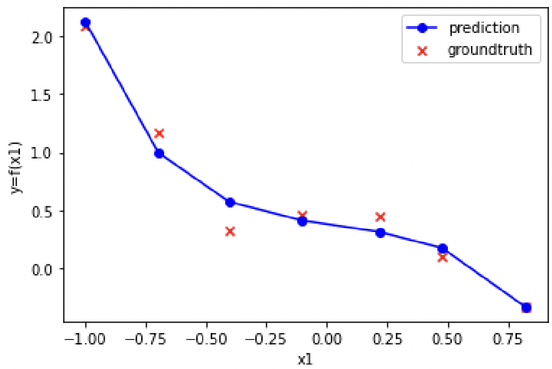
**Q8.** First of all, find the best value of alpha to use in order to optimise best.

The best value of alpha which I managed to find is `0.4975`.

```
cost_lst = list()
model = LinearRegression(x3.shape[1])
alpha = 0.4975 # select an appropriate alpha
lam = 0.0 # 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)))

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()
```




```
Parameter containing:
tensor([[-0.2923,  0.0906, -0.7654,  0.1418, -0.4628,  0.3817]])
Minimum cost: 0.008264507167041302
```



Next, experiment with different values of $\lambda$ and see how this affects the shape of the hypothesis.

Here is a table of values of lambda for the fixed best value of alpha.

| Lambda Value | Cost |
|---|---|
| 0.1 | 0.015416899695992 47 |
| 0.01 | 0.009045101702213287 |
| 0.001 | 0.008343462832272053 |
| 0.0001 | 0.008272412233054638 |

As the lambda value decreases, the cost decreases as well. This is owing to the constricted dataset and how it is mostly better without a form of regularisation.

As for the curves (depicted below) There is no major change with the exponential decrease of the lambda value in this case.

Shape of hypothesis (:Lambda value increases left to right)