Simple Linear Regression

This lesson will focus on what linear regression is and why we need it.

During the last stage of the data science lifecycle, we are faced with the question of which model to choose to make predictions. We can decide what kind of model to use by looking at the relationship between the data variables that we have.

Let’s take the example of predicting tips paid to waiters.

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('tips.csv')
print(df.describe())
# plot between total_bill and tip
df.plot(kind='scatter',x = 'total_bill',y='tip')
plt.show()
```

From the plot, we can see that there is a direct linear relationship between `total_bill` and `tips` where increasing/decreasing `total_bill` results in an increase/decrease in `tips` as well. Looking at the linear relationship, we need a linear model for this problem. If we represent `total_bill` values by x then our prediction ($\hat{y}$) becomes

$$\hat{y} = \theta_0 x + \theta_1$$

This is the form of a linear equation. The term $\theta_1 x$ implies that increasing/decreasing total bill(x) will increase/decrease the tip($\hat{y}$). Now that we have our model, we need to fit it to the data using gradient descent optimization. We will be using the *mean squared error* loss function.

Optimizing the Simple linear model

If we denote our predictions by $\hat{y}$ and the actual values by y, then loss function will be calculated as:

$$L(\theta, X, Y) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

$$L(\theta, X, Y) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (\theta_0 x + \theta_1))^2$$

We will be minimizing this loss function with gradient descent. Recall that in gradient descent we:

- Start with random initial value of θ.
- Compute $\theta_t - \alpha \frac{\partial}{\partial \theta} L(\theta, X, Y)$ to update the value of θ.
- Keep updating the value of θ until it stops changing values. This can be the point where we have reached the minimum of the error function.

Simple linear model fitting in Python

We will need a function that gives us the partial derivative of the loss function.

`gradient`

Before we can implement this in code on our predicting weights example, we need to evaluate the gradient term in the update expression

$$\theta_{t+1} = \theta_t - \alpha \frac{\partial}{\partial \theta} L(\theta_t, X, Y)$$

We know that if our Loss function is *mean squared error* then

$$\frac{\partial}{\partial \theta} L(\theta, X, Y) = \frac{\partial}{\partial \theta_t} [\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2]$$

$$= \frac{1}{n} \sum_{i=1}^{n} -2(y_i - \hat{y}_i) \frac{\partial}{\partial \theta_t} (\hat{y}_i)$$

We know that $\hat{y}_i = \theta_1 x_i + \theta_2$

$$= \frac{1}{n} \sum_{i=1}^{n} -2(y_i - \hat{y}_i) \frac{\partial}{\partial \theta_t} (\theta_0 x_i + \theta_1)$$

Now since we have two parameters (θ_0 and θ_1), we need to differentiate with respect to both. So

Derivating w.r.t θ_0

$$\frac{\partial}{\partial \theta} L(\theta, X, Y) = \frac{1}{n} \sum_{i=1}^{n} -2(y_i - \hat{y}_i) \frac{\partial}{\partial \theta_0} (\theta_0 x_i + \theta_1)$$

$$= \frac{1}{n} \sum_{i=1}^{n} -2(y_i - \hat{y}_i) x_i$$

Derivating w.r.t θ_1

$$\frac{\partial}{\partial \theta} L(\theta, X, Y) = \frac{1}{n} \sum_{i=1}^{n} -2(y_i - \hat{y}_i) \frac{\partial}{\partial \theta_1} (\theta_0 x_i + \theta_1)$$

$$= \frac{1}{n} \sum_{i=1}^{n} -2(y_i - \hat{y}_i)$$

Therefore, we will be coding this expression to compute the gradient. `gradient` function will need the same arguments as the `mse_loss_func` which are:

- `thetas`: The model parameters θ

- `x` : The dataset needed to make predictions
- `y` : The actual values with which to compare

```
def gradient(thetas,x,y):
  n = x.shape[0]
  grad_1 = ( (y - (thetas[0] * x + thetas[1])) * x ).sum()
  grad_2 =  ( y - (thetas[0] * x + thetas[1]) ).sum()
  temp = np.array([grad_1,grad_2])
  grad = (-2 / n) * temp
  return grad
```

In **line 3**, we evaluate the expression inside the summation for θ_0 and take its sum. We do the same in the next line for θ_1. Then in **line 5** we make a numpy array to store both gradients. In **line 6**, we multiply it by $\frac{-2}{n}$ to compute the gradient. We return the gradient in the last line. Note that the returned variable `grad` is also a numpy array of size 2.

Stopping condition

Now we need to decide a stopping condition for gradient descent. We will define a number `epsilon` and say that we will stop updating our model parameter when the change in the model parameter is below `epsilon`. In our case, we can set it to 0.001.

Learning rate

We will call this `alpha`. Its value should be between 0 and 1.

Implementation

```
import pandas as pd
import numpy as np

# loss function to optimize
def mse_loss_func(thetas,x,y):
    predictions = thetas[0] * x + thetas[1]
    sq_error = (y - predictions)**2
    loss = sq_error.mean()
    return loss

# gradient of the loss function
def gradient(thetas,x,y):
  n = x.shape[0]
  grad_1 = ( (y - (thetas[0] * x + thetas[1])) * x ).sum()
  grad_2 =  ( y - (thetas[0] * x + thetas[1]) ).sum()
  temp = np.array([grad_1,grad_2])
  grad = (-2 / n) * temp
  return grad

# read data
```

```
df = pd.read_csv('tips.csv')

# intialize conditions

epsilon = 0.001
alpha = 0.001

# start optimization
thetas = np.array([0.0,0.1])
iterations_completed = 0
print('Starting theta :', thetas)

while(1):
  # compute gradient
  grad = gradient(thetas,x = df['total_bill'],y=df['tip'])

  # update theta
  new_thetas = thetas - (alpha * grad)
  iterations_completed +=1

  # loss on new theta
  loss = mse_loss_func(new_thetas,x = df['total_bill'],y=df['tip'])

  print('\n\niteration: ',iterations_completed)
  print('grad :',grad)
  print('new_theta :', new_thetas)
  print('loss :',loss)

  # stopping condition
  diff = abs(new_thetas - thetas)
  if (diff[0] < epsilon and diff[1] < epsilon ):
      break

  thetas = new_thetas
```

In **lines 5-9**, we write the loss function that we discussed above. In **lines 12-18**, we write the `gradient` function. We read the data and then give values to `epsilon` (**line 24**) and the learning rate `alpha` (**line 25**). We chose our initial $\theta_0 = 0.0$ and $\theta_1 = 0.1$ on **line 28**. We make a variable, `iterations_completed`, which will keep track of the number of iterations of gradient descent at all times.

We start a `while` loop on **line 32** which will keep running until we break it with a `break` statement. Then we start the gradient descent algorithm and compute the gradients in **line 34**. We find the new value of theta (`new_thetas`) in **line 37**. One iteration of gradient descent is complete here. Therefore, we increase the `iterations_completed` by 1.

We then find the loss with `new_thetas` on **line 41** and print our findings. To test for the stopping condition, we compute the absolute difference between `new_thetas` and `thetas` on **line 49** and test the condition on the next line. If the stopping condition is satisfied, we exit the loop with the `break` statement in **line 51**. But if the condition is not satisfied, we move to **line 53** where the variable `thetas` is given the value of `new_thetas`, so that the same variables can be used in the next iteration of the loop.

Results

From the output, we can see that it takes only 3 iterations for gradient descent to reach the best model. It chooses $\theta_0 \approx 0.139$ and $\theta_1 \approx 0.107$. The mean squared loss is approximately 1.14, which is better than what we had gotten in the previous lesson. These are the best model parameters that can be chosen for this model and give the minimum error.

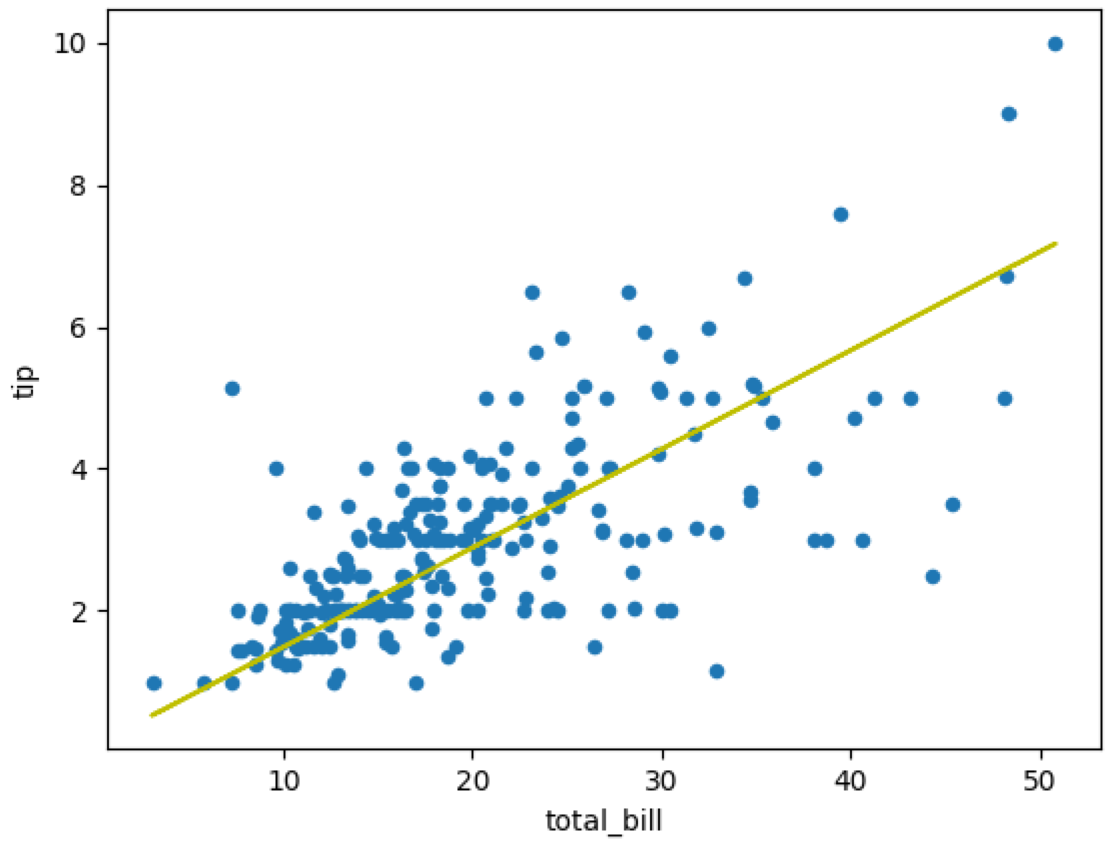
Best fit line

The equation of our model $\hat{y} = \theta_0 x + \theta_1$ is the equation of a line with θ_0 as the slope of the line and θ_1 as the intercept. The line produced by this equation is called the **best fit** line. We can plot this line alongside the scatter plot we plotted above.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

df = pd.read_csv('tips.csv')

# Plot actual vlues
df.plot(kind='scatter',x = 'total_bill',y='tip')

# Plot best fit line of predictions
thetas = np.array([0.139,0.107])
predictions = thetas[0] * df['total_bill'] + thetas[1]
plt.plot(df['total_bill'], predictions,color = 'y')
```

In **line 8** we plot the scatter plot between `total_bill` and `tip`. In **line 11**, we initialize our thetas to the best fit thetas we found above by gradient optimization. We retrieve predictions using these thetas in the next line. Then we plot the predictions at the y-axis and `total_bill` at x-axis on the same scatter plot.

scatter plot.



From the plot, we can see that line fits the data quite well. The whole process that we did above for finding this best fit line is called **linear regression**. It gives us the best fit line, i.e., the line with the minimum error.

This is a very fundamental concept that we will extend in the next lesson. Can you think of a way to improve or extend this model?