Optimization with Gradient Descent

This lesson will focus on how to implement gradient descent algorithm in Python.

WE'LL COVER THE FOLLOWING

In the previous lesson, we looked at the intuition behind the gradient descent algorithm and the update equation. In this lesson, we are going to implement it in Python. We are going to predict the tips paid by a customer at a restaurant. We will choose the best model using gradient descent.

Minimization with Gradient Descent

Recall that the gradient descent algorithm is

- Start with a random initial value of θ.
- Compute $\theta_t - \alpha \frac{\partial}{\partial \theta} L(\theta_t, Y)$ 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.

We will be using the *tips* dataset that has the following data.

```
# Tips Dataset
# total_bill : Total bill of the customer
# tip : Total tip paid by the customer
# gender : Tgender of the customer(Male/Female)
# smoker : smoker(yes/no)
# day : which day of the week
# time: time of visit (lunch/dinner)
# people: total people that came to dine in.
```

Our simple model said that for predicting tips we only need the amount of the total bill paid by the customer. Therefore, our prediction ($\hat{y}$) depends on the total bill (x) and the model parameter (θ). We have:

$$\hat{y} = \theta x$$

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

`gradient`

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

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

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

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

since we chose a simple model which said that for predicting tips we only need the amount of the total bill paid by the customer. Therefore, our prediction ($\hat{y}$) depends on the total bill (x) and the model parameter (θ). We have

$$\hat{y} = \theta x$$

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

Evaluating this expression gives us:

$$\frac{\partial}{\partial \theta} = \frac{-2}{} \sum^{n}$$

$$\frac{\partial}{\partial\theta}L(\theta_t, Y) = \frac{-2}{n}\sum_{i=1}^{n}(y_i - \theta_t x_i)(x_i)$$

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:

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

```
def gradient(theta,x,y):
  n = x.shape[0]
  temp = (y - (theta * x)) * x
  temp_sum = temp.sum()
  grad = (-2 / n) * temp_sum
  return grad
```

In **line 3**, we evaluate the expression inside the summation and take its sum in the next line. **In line 5**, we multiply it by $\frac{-2}{n}$ to compute the gradient. We return the gradient in the last line.

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.

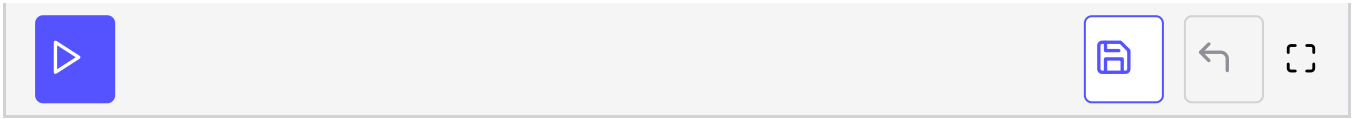
Learning rate

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

Implementation

```
import pandas as pd

# loss function to optimize
def mse_loss_func(theta,x,y):
    predictions = theta * x
    sq_error = (y - predictions)**2
    loss = sq_error.mean()
    return loss

# gradient of the loss function
def gradient(theta,x,y):
```

```
  n = x.shape[0]
  temp = (y - (theta * x)) * x
  temp_sum = temp.sum()

  grad = (-2 / n) * temp_sum
  return grad

# read data
df = pd.read_csv('tips.csv')

# intialize conditions
epsilon = 0.001
alpha = 0.001

# start optimization
theta = 0.0
iterations_completed = 0
print('Starting theta :', theta)

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

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

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

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

  # stopping condition
  diff = abs(new_theta - theta)
  if (diff < epsilon):
      break

  theta = new_theta
```



In **lines 4-8**, we write the loss function that we discussed above. In **lines 11-16**, we write the `gradient` function. We read the data and then give values to `epsilon` (**line 22**) and the learning rate `alpha` (**line 23**). We chose our initial $\theta = 0.0$ on **line26**. 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 30** 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 32**. We find the new value of theta (`new_theta`) in **line 35**. One iteration of gradient descent is complete until here. Therefore,

we increase the `iterations_completed` by 1.

We then find the loss with `new_theta` on **line 39** and print our findings. To test for the stopping condition, we compute the absolute difference between `new_theta` and `theta` on **line 47** and test the condition on the next line. If the stopping condition is satisfied, we exit the loop with a `break` statement in **line 49**. But if the condition is not satisfied, we move to **line 51** where the variable `theta` is given the value of `new_theta`, 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 $\theta \approx 0.143$. The mean squared loss for $\theta \approx 0.143$ is approximately 1.17 which is better than what we chose manually a few lessons back. This is the best model parameter that can be chosen for this model and gives the minimum error.

Types of Gradient Descent

Gradient descent is normally categorized into three types:

- Batch Gradient Descent
- Stochastic Gradient Descent
- Mini-batch Gradient Descent

We will study them one by one.

Batch Gradient Descent

This is the gradient descent that we implemented above. In **batch gradient descent**, we calculate the gradient of the loss function on the entire dataset. We update model parameters according to the errors introduced on the whole data set. However, this approach is not scalable with large datasets. A single iteration may take a lot of time if the dataset is very large.

Stochastic Gradient Descent

In **stochastic gradient descent**, we compute the gradients of the loss function on a single randomly chosen example form the dataset. Instead of using the

whole dataset in every iteration we use a single data point from the dataset to compute the gradient of the loss function.

Even though batch gradient descent takes big steps towards the minimum, stochastic gradient descent is faster for a single iteration and it takes steps towards the minimum very rapidly. It may reach the minimum faster for large datasets than batch gradient descent, as batch gradient descent takes steps after a long time.

Mini-batch Gradient Descent

Mini-batch gradient descent is in between the two approaches explained above. In **mini-batch gradient descent**, a set of random data points are chosen to calculate the gradient of the loss function at each iteration. It was named **mini-batch** because the chosen set is a very small subset of the entire dataset. This is used widely for optimizing functions as it provides a balance between the speed and size of steps taken towards the minimum.

Scratching the surface

So far in this chapter, we have learned how to use gradient descent to minimize a function. Gradient Descent is a very useful algorithm that we will keep coming back to as our problems and models become more and more complex. The current example used a very basic model for an easy problem. We can perform better at predicting tips from this dataset by using some other models with gradient descent.

In the next lesson, we will look at a fundamental concept for predictive analysis which is *linear regression*.