index

March 11, 2022

1 Gradient Descent - Lab

1.1 Introduction

In this lab, you'll continue to formalize your knowledge of gradient descent by coding the algorithm yourself. In the upcoming labs, you'll apply similar procedures to implement logistic regression on your own.

1.2 Objectives

In this lab you will:

- Implement gradient descent from scratch to minimize OLS

1.3 Use gradient descent to minimize OLS

To practice gradient descent, you'll investigate a simple regression case in which you're looking to minimize the Residual Sum of Squares (RSS) between the predictions and the actual values. Remember that this is referred to as Ordinary Least Squares (OLS) regression. You'll compare two simplistic models and use gradient descent to improve upon these initial models.

1.4 Load the dataset

- Import the file `'movie_data.xlsx'` using Pandas
- Print the first five rows of the data

  You can use the `read_excel()` function to import an Excel file.

[1]:
```
# Import the data
import pandas as pd
df = pd.read_excel('movie_data.xlsx')

# Print the first five rows of the data
df.head()
```

[1]:

	budget	domgross	title
0	13000000	25682380	21 & Over
1	45658735	13414714	Dredd 3D
2	20000000	53107035	12 Years a Slave
3	61000000	75612460	2 Guns

```
4   40000000   95020213                   42
```

1.5 Two simplistic models

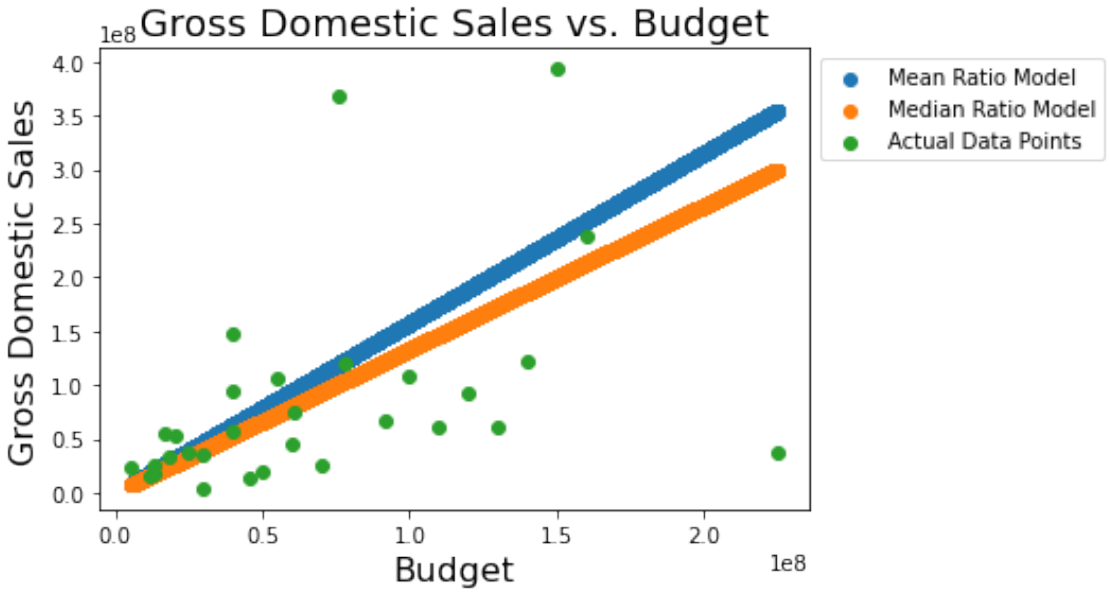
Imagine someone is attempting to predict the domestic gross sales of a movie based on the movie's budget, or at least further investigate how these two quantities are related. Two models are suggested and need to be compared.
The two models are:

$\text{domgross} = 1.575 \cdot \text{budget}$
$\text{domgross} = 1.331 \cdot \text{budget}$

Here's a graph of the two models along with the actual data:

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

x = np.linspace(start=df['budget'].min(), stop=df['budget'].max(), num=10**5)
plt.scatter(x, 1.575*x, label='Mean Ratio Model') # Model 1
plt.scatter(x, 1.331*x, label='Median Ratio Model') # Model 2
plt.scatter(df['budget'], df['domgross'], label='Actual Data Points')
plt.title('Gross Domestic Sales vs. Budget', fontsize=18)
plt.xlabel('Budget', fontsize=16)
plt.ylabel('Gross Domestic Sales', fontsize=16)
plt.legend(bbox_to_anchor=(1, 1))
plt.show()
```



1.6 Error/Loss functions

To compare the two models (and future ones), a metric for evaluating and comparing models to each other is needed. Traditionally, this is the residual sum of squares. As such you are looking to minimize

$$\sum(\hat{y} - y)^2$$

. Write a function `rss()` which calculates the residual sum of squares for a simplistic model:

$$\text{domgross} = m \cdot \text{budget}$$

[3]:
```
def rss(m, X=df['budget'], y=df['domgross']):
    dif = np.array(y - m*X)
    return np.dot(dif,dif)
```

1.7 Find the RSS for the two models

Which of the two models is better?

[4]:
```
# Your code here
print(rss(1.575, X=df['budget'], y=df['domgross']))
```

```
2.7614512142376128e+17
```

[5]:
```
print(rss(1.331, X=df['budget'], y=df['domgross']))
```

```
2.3547212057814554e+17
```

[6]:
```
# Your response here
## They are both bad
```

1.8 Gradient descent

Now that you have a loss function, you can use numerical methods to find a minimum to the loss function. By minimizing the loss function, you have achieved an optimal solution according to the problem formulation. Here's the outline of gradient descent from the previous lesson:

1. Define initial parameters:
    1. pick a starting point
    2. pick a step size α (alpha)
    3. choose a maximum number of iterations; the algorithm will terminate after this many iterations if a minimum has yet to be found
    4. (optionally) define a precision parameter; similar to the maximum number of iterations, this will terminate the algorithm early. For example, one might define a precision parameter of 0.00001, in which case if the change in the loss function were less than 0.00001, the algorithm would terminate. The idea is that we are very close to the bottom and further iterations would make a negligible difference
2. Calculate the gradient at the current point (initially, the starting point)
3. Take a step (of size alpha) in the direction of the gradient

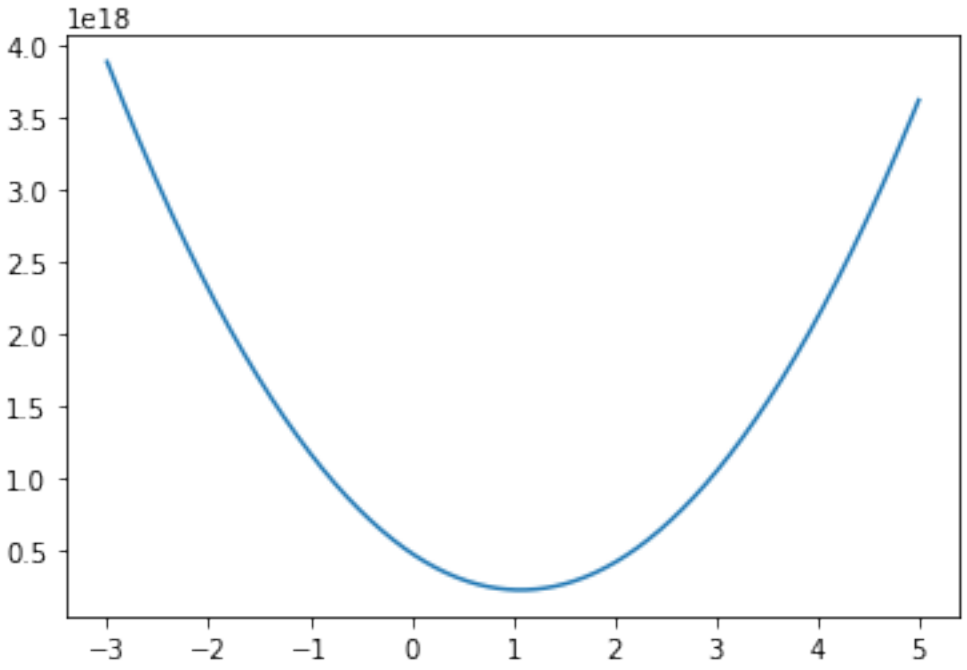
4. Repeat steps 2 and 3 until the maximum number of iterations is met, or the difference between two points is less then your precision parameter

To start, visualize the cost function. Plot the cost function output for a range of m values from -3 to 5.

[7]:
```
# Your code here
import seaborn as sns
m = np.linspace(-3, 5, num = 1000)

loss = [rss(i) for i in m]

sns.lineplot(x = m, y = loss);
```



As you can see, this is a simple cost function. The minimum is clearly around 1. With that, it's time to implement gradient descent in order to find the optimal value for m.

[10]:
```
# Set a starting point
cur_x = 1.5

# Initialize a step size
alpha = 10**(-7)

# Initialize a precision
precision = 0.0000001
```

```
# Helpful initialization
previous_step_size = 1

# Maximum number of iterations
max_iters = 10000

# Iteration counter
iters = 0

for i in enumerate(range(max_iters)):

# Create a loop to iterate through the algorithm until either the
# max_iteration or precision conditions is met
# Your code here; create a loop as described above
    # Calculate the gradient. This is often done by hand to reduce
    # computational complexity.
    # For here, generate points surrounding your current state, then calculate
    # the rss of these points
    # Finally, use the np.gradient() method on this survey region.
    # This code is provided here to ease this portion of the algorithm
    # implementation

    x_pre = cur_x
    x_survey_region = np.linspace(start = cur_x - previous_step_size ,
                                  stop = cur_x + previous_step_size ,
                                  num = 101)
    rss_survey_region = [np.sqrt(rss(m)) for m in x_survey_region]
    gradient = np.gradient(rss_survey_region)[50]

    # Update the current x, by taking an "alpha sized" step in the direction
    # of the gradient
    cur_x -= alpha * gradient
    iters +=1
    previous_step_size = abs(x_pre - cur_x)
    if previous_step_size <= precision:
        break

    # Update the iteration number


# The output for the above will be: ('The local minimum occurs at', 1.
 ↪1124498053361267)
print("The local minimum occurs at", cur_x)
print(iters)
```

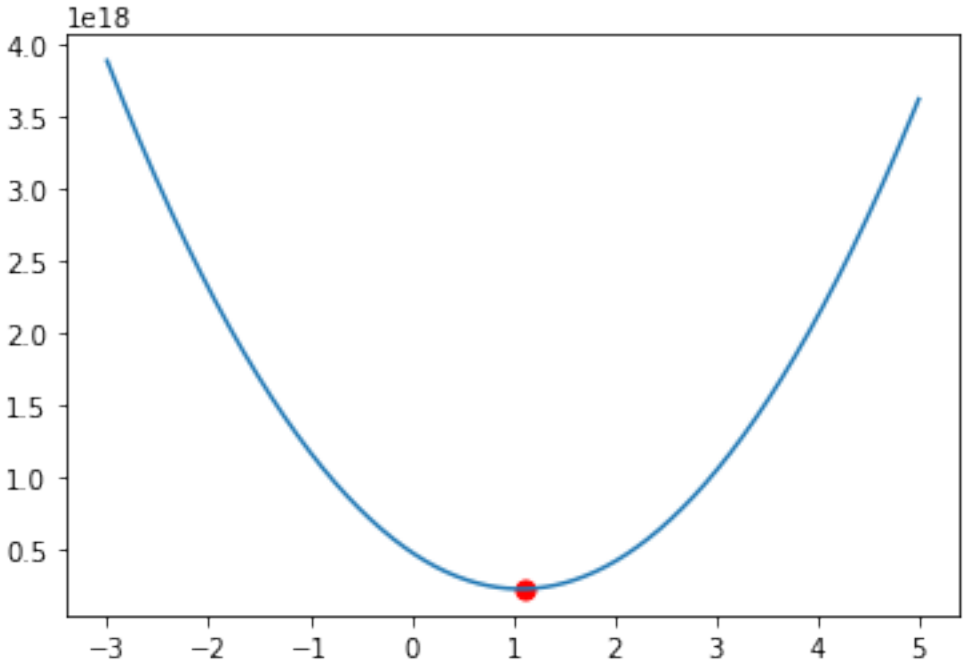
```
The local minimum occurs at 1.112449806423876
6
```

1.9 Plot the minimum on your graph

Replot the RSS cost curve as above. Add a red dot for the minimum of this graph using the solution from your gradient descent function above.

[26]:
```
# Your code here
m = np.linspace(-3, 5, num = 10**3)
loss = [rss(i) for i in m]
sns.lineplot(x = m, y = loss);
print(cur_x)
print(rss(cur_x))
plt.scatter(cur_x, rss(cur_x), marker='o', s=50, color = "red");
```

```
1.112449806423876
2.213449943121513e+17
```



1.10 Summary

In this lab, you coded up a gradient descent algorithm from scratch! In the next lab, you'll apply this to logistic regression in order to create a full implementation yourself!