$Univariate\ \ Linear\ \ Regresion$

Definition

Given a data set (training set), contains elements which consists of paired values of the independent variable (input) and the dependent variable (output)
D = { (x_1 , y_1), (x_2 , y_2), ... , (x_n , y_n) } where x_i is called features and y_i is the **label** on the i^{th} data point. Linear Regression aims to estimate a function f_θ with little or no knowledge about the function form. This can be used to predict a label y given a new x.

General Form of Linear Regression

$$f_\theta(x) = \theta^T \phi(x) = \sum_{i=1}^{m} \theta_i \phi_i(x)$$

- $f_\theta(x)$ is called the hypothesis function.
- $\phi(x)$ is a basis function (by analogy with the concept of vectors are composed of a linear combination of basis vectors).
- θ is the coefficient of the linear combination. In neural network it is often referred as '**Weight**'

Cost Function

Definition

Cost Function (Loss Function), sometimes called Sum of Squared estimate of Errors (SSE). This is calculated by summing up all the squared differences between predicted and actual label y . This is the important criteria to evaluate a Linear Regression model. The general formula for calculating the Cost Function is:

$$J(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} (y_i - f_\theta(x_i))^2$$

- $J(\theta_0, \theta_1, \ldots, \theta_n)$ is the Cost Function.
- $f_\theta(x)$ is the hypothesis.
- m is the size of the data set.

Usage

This can be used to calculate how fit the hypothesis is with the given set of data. The hypothesis will be the fittest when Cost Function is at its extreme minimum. This can be achieved by using Gradient Descent algorithm.

Gradient Descent

Definition

A first-order iterative optimization algorithm for finding a local minimum of a differentiable function. This idea is to take repeated steps in the opposite direction of the gradient of the cost function $J(\theta_0, \theta_1, \ldots, \theta_n)$ at the current point. We will use the θ as a matrix vector containing all θ.

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \ldots \\ \theta_n \end{bmatrix}$$

Algorithm

Repeat until converges:

$$\theta_{i+1} = \theta_i - \alpha \nabla J(\theta)$$

With

$$\nabla J(\theta) = \begin{bmatrix} \frac{\partial J(\theta_0)}{\partial \theta_0} \\ \\ \frac{\partial J(\theta_1)}{\partial \theta_1} \\ \\ \ldots \\ \\ \frac{\partial J(\theta_n)}{\partial \theta_n} \end{bmatrix}$$

- α is the learning rate.
- $\nabla J(\theta)$ is the gradient of the cost function $J(\theta)$.

```
 repeat until converge
{
```

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_j)$$

$$\theta_j = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} [(h_\theta(x_i) - y_i)\phi_j(x_i)]$$

```
 }
```

(simutaneously update θ_j for $j = 0, 1, 2, \ldots, m$)

Demo Model

Read input, build DataFrame and plot the data points

In [ ]:

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

np.random.seed(100)

plt.rcParams["figure.figsize"] = (10,8)

df = pd.read_csv('/Users/danielnguyen/Repo/AI/Training_set/likes.csv')

df.tail()
```

Out[ ]:

	Day	Likes
102	102	2495
103	103	2495
104	104	2499
105	105	2502
106	106	2511

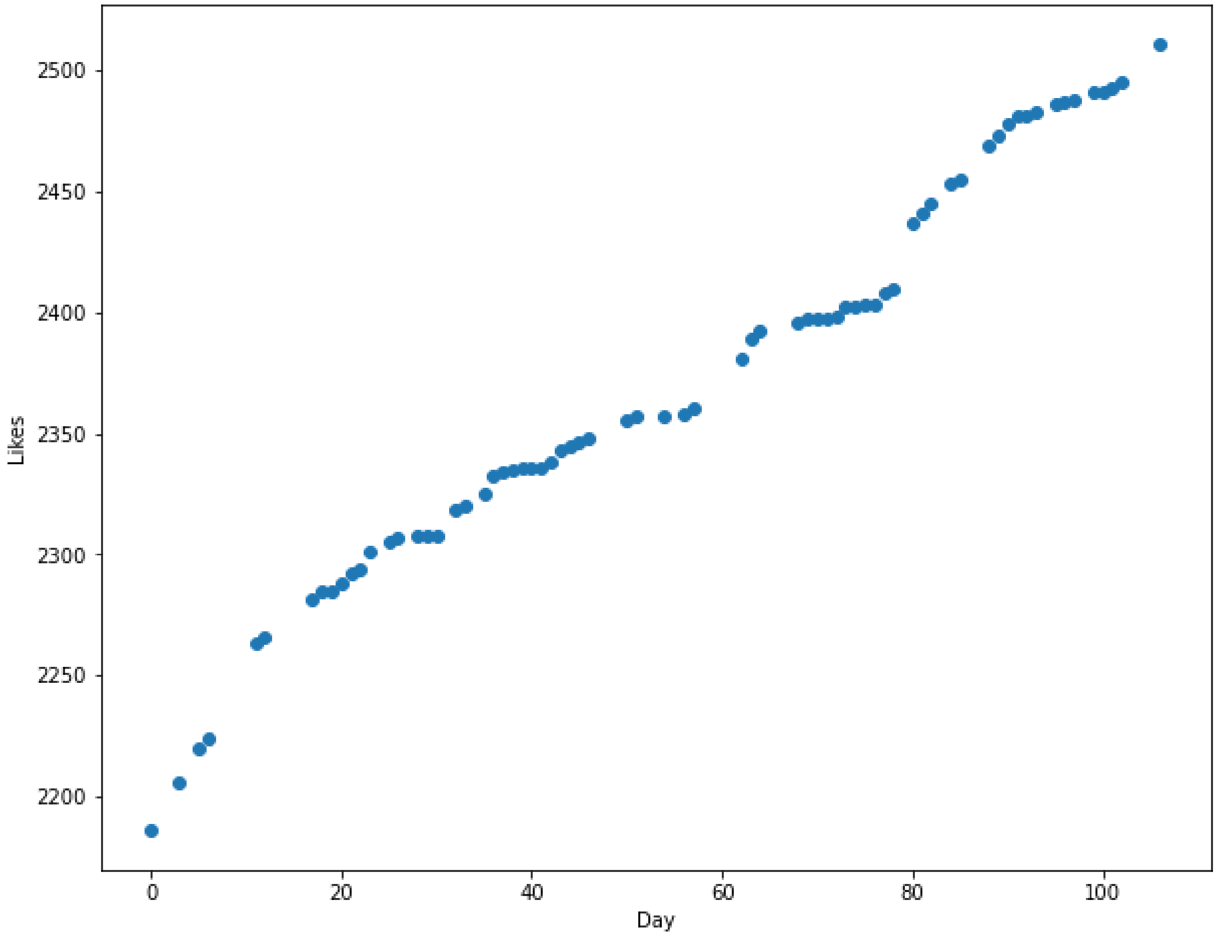
In [ ]:

```
data0 = np.array(df.loc[:, 'Day':'Likes'])

#Shuffle the data
data1 = np.random.permutation(data0)

#Sort the first 70 pairs (x, y) and use it as our training set
training_set = np.sort(data1[:70].T).T

#Shows the first 10 data
print(training_set[:10].T)

#Arrays of x and y
days = np.sort(training_set[:70, 0])
likes = np.sort(training_set[:70, 1])

#Training set size
m = len(likes)

plt.xlabel('Day')
plt.ylabel('Likes')
plt.scatter(days, likes)
```

```
[[   0    3    5    6   11   12   17   18   19   20]
 [2186 2206 2220 2224 2263 2266 2281 2285 2285 2288]]
```

Out[ ]:

```
<matplotlib.collections.PathCollection at 0x7f9cd9615b90>
```



Choosing the hypothesis function

Here is the plot for the total number of likes of the **GDSC**'s Facebook page for 107 days from **31/12/2020**. First, we choose the hypothesis fucntion $f_\theta(x) = \theta_0 + \theta_1 x$ (linear one-variable function). We will use $\theta_0 = 2186$ and $\theta_1 = 0$.

In [ ]:
```
theta0 = 2186
theta1 = 0

t_vect = np.array([[theta0], [theta1]])
print(t_vect)

#Change the x matrix to a (m x 2) matrix

x_0 = np.ones((m, 1))
x_1 = days.reshape(m, 1)

x = np.hstack((x_0, x_1))
y = likes.reshape(m, 1)

mean = np.mean(y)

print(x[:10])
```

```
[[2186]
 [   0]]
[[ 1.  0.]
 [ 1.  3.]
 [ 1.  5.]
 [ 1.  6.]
 [ 1. 11.]
```

```
 [ 1. 12.]
 [ 1. 17.]
 [ 1. 18.]
 [ 1. 19.]
 [ 1. 20.]]
```

Find the Cost Function

Now we write the cost function for this hypothesis.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (f_\theta(x_i) - y_i)^2$$

With m is the number of training example.

In [ ]:

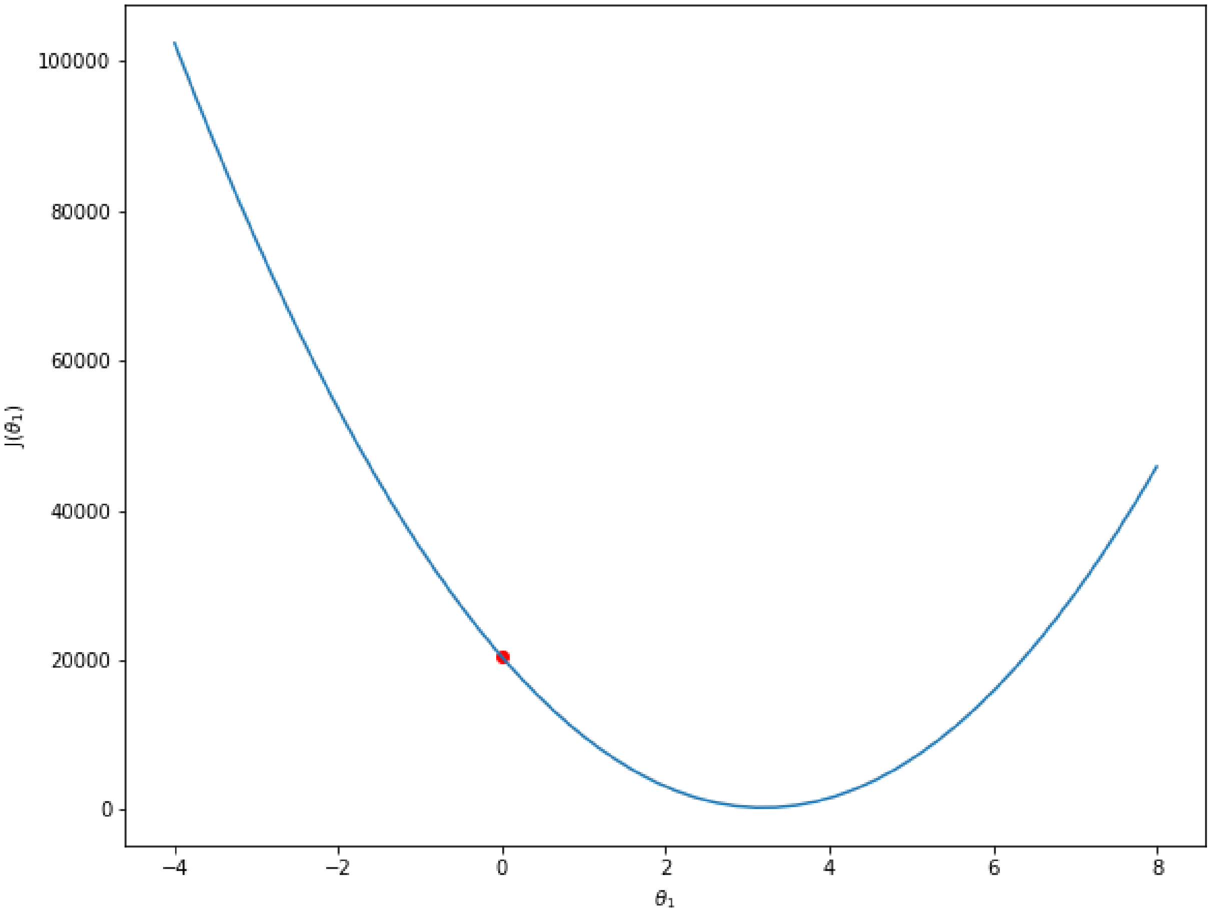
```python
def calculate_cost(t_vect, x_matrix, y_matrix) :
    """Calculate the cost function for linear regression model"""

    cost_matrix = np.dot(x_matrix, t_vect) - y_matrix

    return np.sum(np.square(cost_matrix)) / (2 * m)
```

In [ ]:

```python
theta1_lst = np.linspace(-4, 8)
cost_lst = []

#Calculate the cost function for every theta1 from -4 to 8
for theta in theta1_lst:
    t_vect_temp = np.array([[theta0], [theta]])
    cost_lst.append(calculate_cost(t_vect_temp, x, y))
cost_lst = np.array(cost_lst)

plt.xlabel(r'$\theta_1$')
plt.ylabel('J(' + r'$\theta_1)$')

plt.scatter(theta1, calculate_cost(t_vect, x, y), c='red')
plt.plot(theta1_lst, cost_lst)
```

Out[ ]:

```
[<matplotlib.lines.Line2D at 0x7f9cd9b0f490>]
```



Use Gradient Descent to optimize the model

From the plot of data above, we can see that the line has not fit the data, which means the hypothesis h_θ is not sufficient for predicting new values. This is called UNDERFITTING. If we want to be more precisely, we have to minimize the result of cost function $J(\theta_0, \theta_1)$, meaning find the local minima of the function. To achieve this, we use the Gradient Descent. The GD formula for this will be:

$$\theta_0 = \theta_1 - \alpha \frac{\partial}{\partial\theta_0} J(\theta_0)$$

$$\theta_1 = \theta_1 - \alpha \frac{\partial}{\partial\theta_1} J(\theta_1)$$

This is equivalent to:

$$\theta_0 = \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} ((f_\theta(x_i) - y_i))$$

$$\theta_1 = \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} ((f_\theta(x_i) - y_i)x_i)$$

In [ ]:

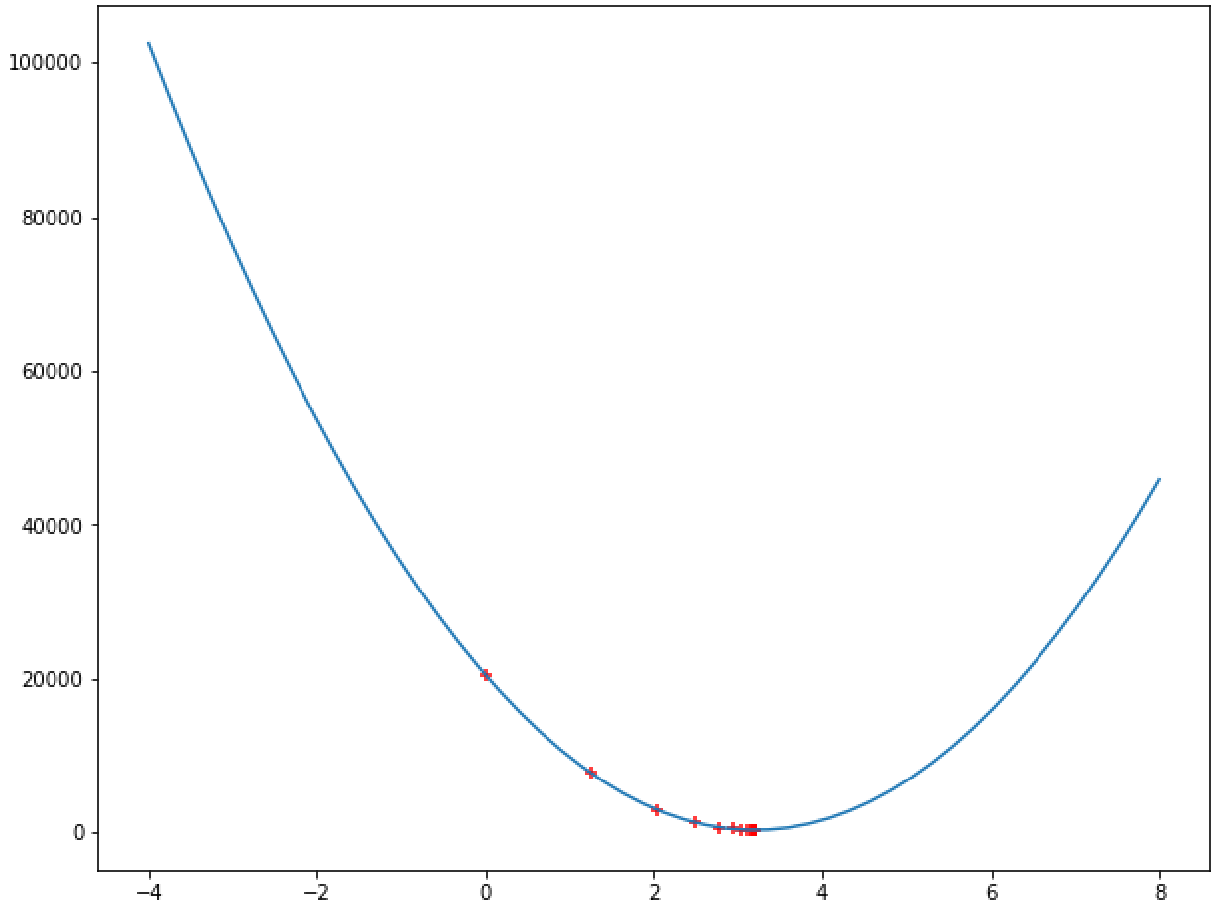
```
def gradient_func(theta, x_matrix, y_matrix):
    """Find the gradient of the given model"""

    gradient_matrix = np.dot(x_matrix.T, (np.subtract(np.dot(x_matrix, thet

    return gradient_matrix / m
```

In [ ]:
```
def gradient_descent(theta, x_matrix, y_matrix, threshhold = 1.0e-6, max_ite
    """Optimize θ1 to minimize the cost function and return 2 arrays contai
    on the cost function plot for which the gradient descent visited each i

    theta1_history = theta[1][0]
    j_history = calculate_cost(theta, x_matrix, y_matrix)
    i = 0
    diff = 1e10

    while i < max_iterations and diff > threshhold:
        theta = theta - learning_rate * gradient_func(theta, x_matrix, y_ma
        theta1_history = np.vstack((theta1_history, theta[1][0]))
        j_history = np.vstack((j_history, calculate_cost(theta, x_matrix, y_

        i += 1
        diff = np.absolute(j_history[-1] - j_history[-2])

    return theta, theta1_history, j_history
```

In [ ]:
```
t_vect, theta1_arr, j_arr = gradient_descent(t_vect, x, y)

print(t_vect)

plt.scatter(theta1_arr.T, j_arr.T, c='red', marker='+')
plt.plot(theta1_lst, cost_lst)
```

```
[[2186.90589965]
 [   3.182753  ]]
```

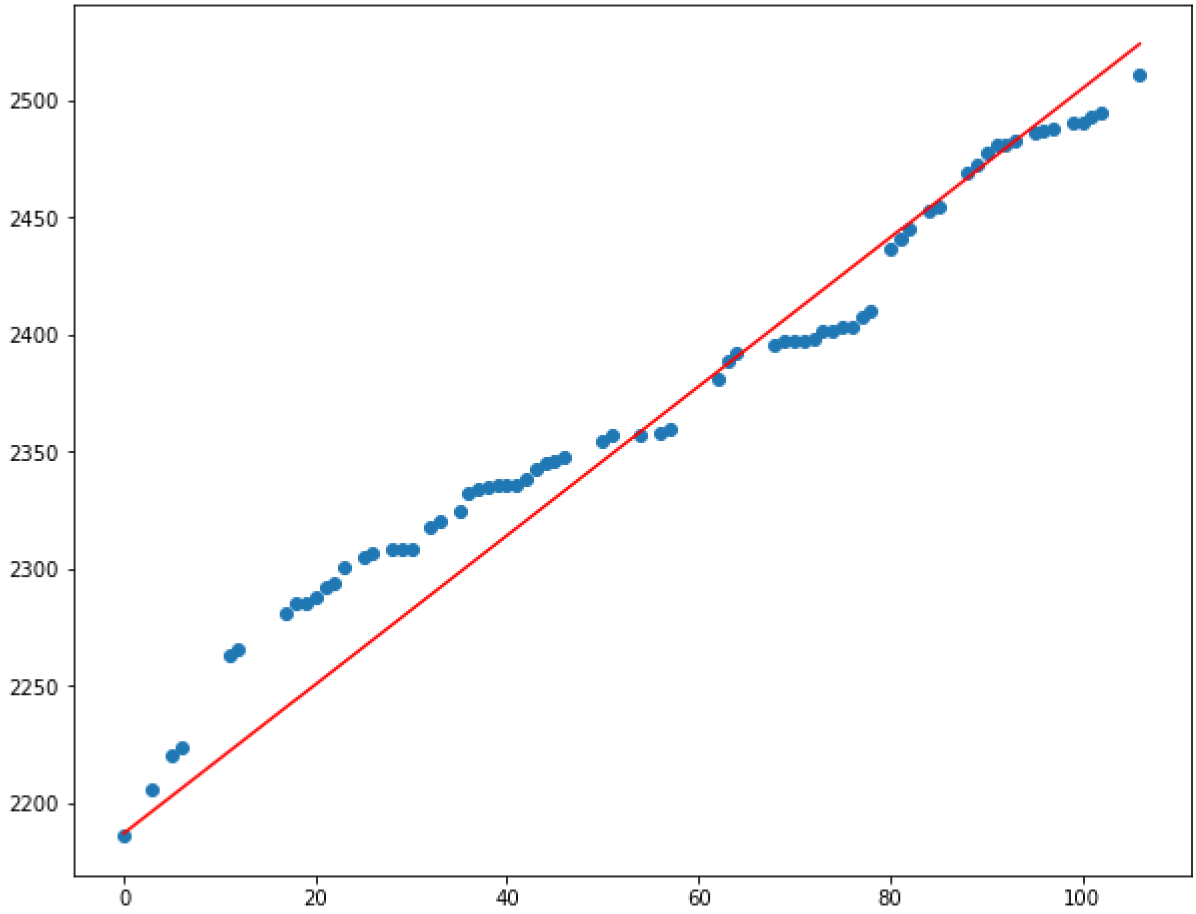
Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7f9cd9bdbed0>]
```



In [ ]:
```
plt.scatter(days, likes)
```

```
plt.plot(days, np.dot(x, t_vect).reshape(1, m)[0], color='red')
```

Out[ ]: [<matplotlib.lines.Line2D at 0x7f9cd9f78890>]

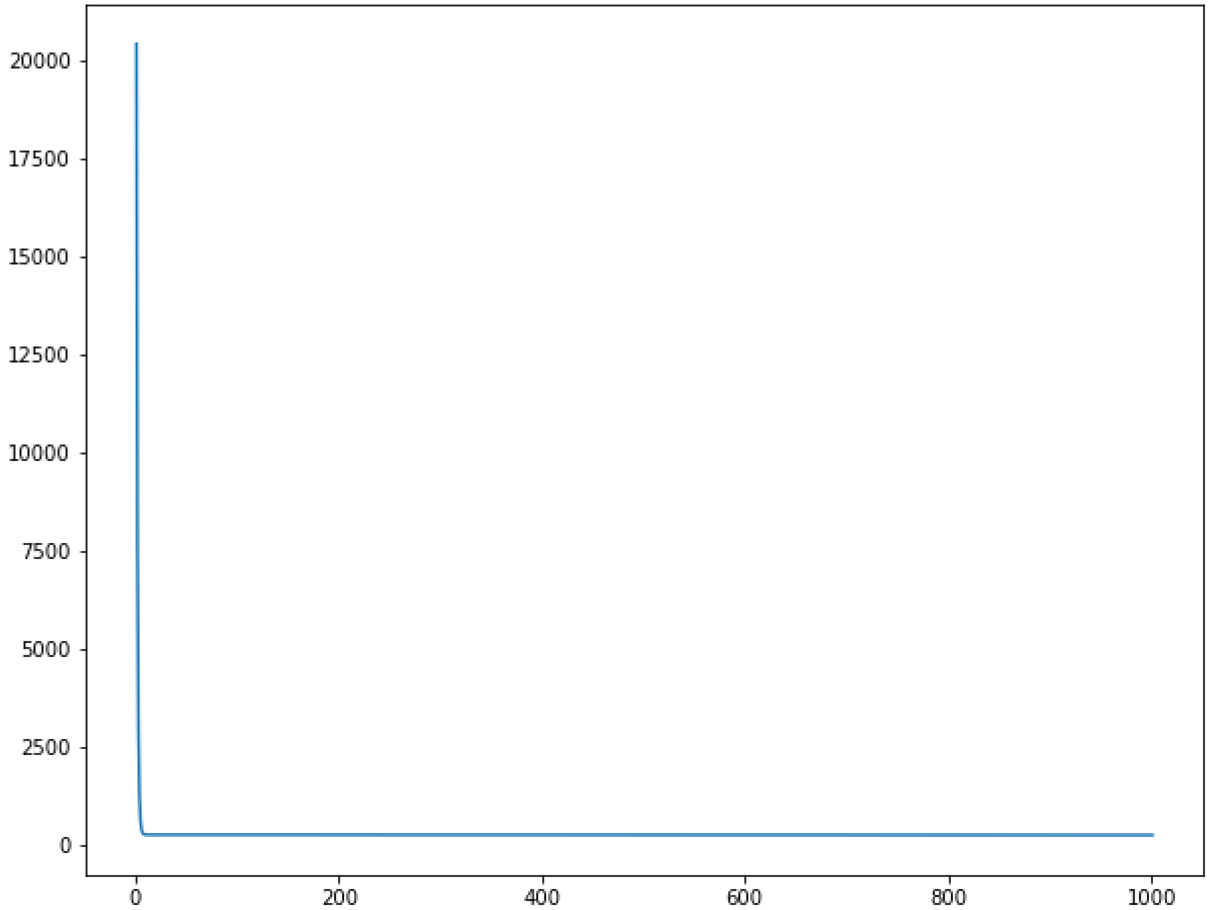


Gradient Descent Performance

This step is to check the performance of GD. For example, by looking at this plot, we can know if we had chosen the correct learning rate α or not. The plot gets steeper if we chose the correct learning rate. If it just a horizontal line, that means your learning rate is too large, making the algorithm diverge.

In [ ]:
```
plt.plot(range(1, len(j_arr) + 1), j_arr.reshape(1, len(j_arr))[0])
```

Out[ ]: [<matplotlib.lines.Line2D at 0x7f9cda117e90>]



Testing and evaluating the model

After finish optimizing the model, we use the unused data to test our model.

In [ ]:
```
#test
def get_num_likes(day=0):
    global t_vect
    return t_vect[0][0] + t_vect[1][0] * day

col_names = ['Day','Predicted','Actual','Error']
test_set = np.sort(data1[70:].T).T
predicted_values = np.around(get_num_likes(test_set[0:, 0]))
real_values = test_set[0:, 1]

diff = predicted_values - real_values

with open('/Users/danielnguyen/Repo/AI/Testing_set/test.csv', 'w') as f:
    f.write(','.join(col_names) + '\n')
    for i in range(len(test_set)):
        f.write(','.join([str(test_set[i][0]), str(predicted_values[i]),
                          str(real_values[i]), str(diff[i] / real_values[i]
```

To how fit is the model, we calculate R^2 (R-Squared value) of the model.
R^2 = 1 - $\frac{J}{SSTO}$

SSTO (Total Sum of Square) is calculated according to this formula:
$SSTO$ = $\frac{1}{2m}\sum_{i=1}^{m}(y_{mean} - y_i)^2$

In [ ]:
```
SSTO = np.sum(np.square(mean - y)) / (2 * m)
```

```
r_square = 1 - (calculate_cost(t_vect, x, y) / SSTO)

print('R-Squared Value = ' + str(r_square))
```

```
R-Squared Value = 0.9247201340616541
```

The R^2 is 0.92, which is a good fit model, can make good predictions without being overfitted

Changing the hypothesis

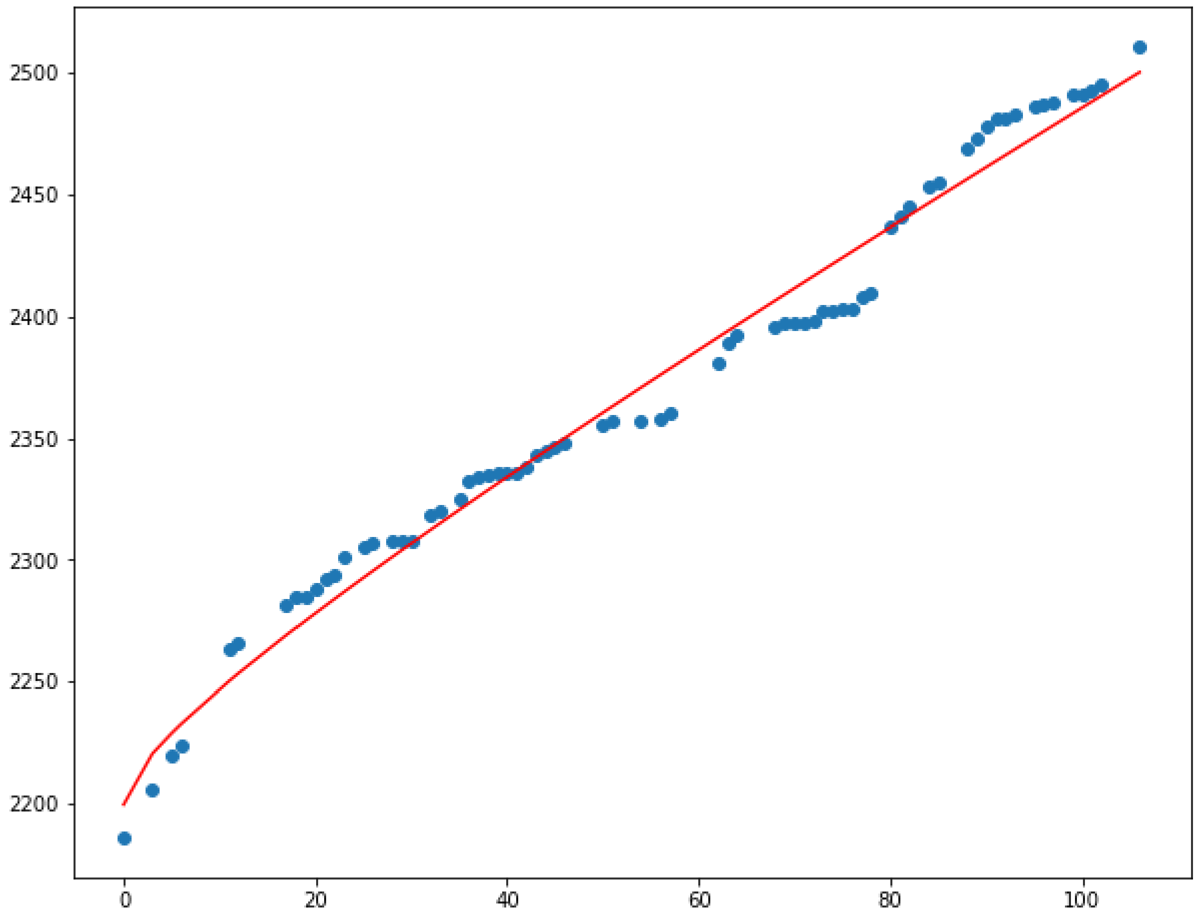
In the previous model, the R^2 = 0.92, but we can do better. Now I am going to change my hypothesis by using a different basis function $h_\theta(x) = \theta_0 + \theta_1 \sqrt{x} + \theta_2 x$

In [ ]:

```
t0 = 2199.55779
t1 = 8.622317
t2 = 1.998494

plt.plot(x, t0 + t1 * x ** 0.5 + t2 * x, c='red')
plt.scatter(days, likes)
```

Out[ ]:

```
<matplotlib.collections.PathCollection at 0x7f9cd9f6b450>
```



Now we also try a polynomial function with a degree equals 3.

$$g_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$$

In [ ]:

```
#First step: process all thetas into a column vector for easy computation
theta0 = 0 #2202.2975324712297
```

```
theta1 = 0 #5.01399360
theta2 = 0 #-5.49625136e-2
theta3 = 0 #3.43653139e-4

theta = np.array([[theta0], [theta1], [theta2], [theta3]])

print(theta)

x_1 = (x_1 - x_1[0][0]) / (x_1[-1][0] - x_1[0][0])
x_2 = np.square(days).reshape((m, 1))
x_2 = (x_2 - x_2[0][0]) / (x_2[-1][0] - x_2[0][0])
x_3 = np.power(days, 3).reshape((m,1))
x_3 = (x_3 - x_3[0][0]) / (x_3[-1][0] - x_3[0][0])

# Second step: process x into a m x 4 matrix containing (from left to right
x = np.hstack((x_0,x_1,x_2,x_3))

print(x[-5:])
```

```
[[0]
 [0]
 [0]
 [0]]
[[1.         0.93396226 0.87228551 0.81468175]
 [1.         0.94339623 0.88999644 0.83961928]
 [1.         0.95283019 0.90788537 0.86506059]
 [1.         0.96226415 0.9259523  0.8910107 ]
 [1.         1.         1.         1.        ]]
```

In [ ]:

```
def cost_func2(theta, x_matrix, y_matrix) :
    return np.sum(np.square(np.dot(x_matrix, theta) - y_matrix)) / (2 * len
```

In [ ]:

```
def grad_desc(theta, x_matrix, y_matrix, threshhold=1e-10, iterations=7749,

    diff = -1e5
    cost_history = cost_func2(theta, x_matrix, y_matrix)
    i = 0
    while i < iterations and -diff > threshhold and diff <= 0:
        hypo = np.dot(x_matrix, theta)
        errors = np.subtract(hypo, y_matrix)
        delta_theta = np.dot(x_matrix.T, errors) / len(y_matrix)

        theta = theta - learning_rate * delta_theta

        i += 1

        cost_history = np.vstack((cost_history, cost_func2(theta, x_matrix,

        diff = cost_history[-1] - cost_history[-2]

    return theta, cost_history
```

In [ ]:

```
def norm_func(x_matrix, y_matrix) :
    return np.linalg.multi_dot((np.linalg.inv(np.dot(x_matrix.T, x_matrix))
```

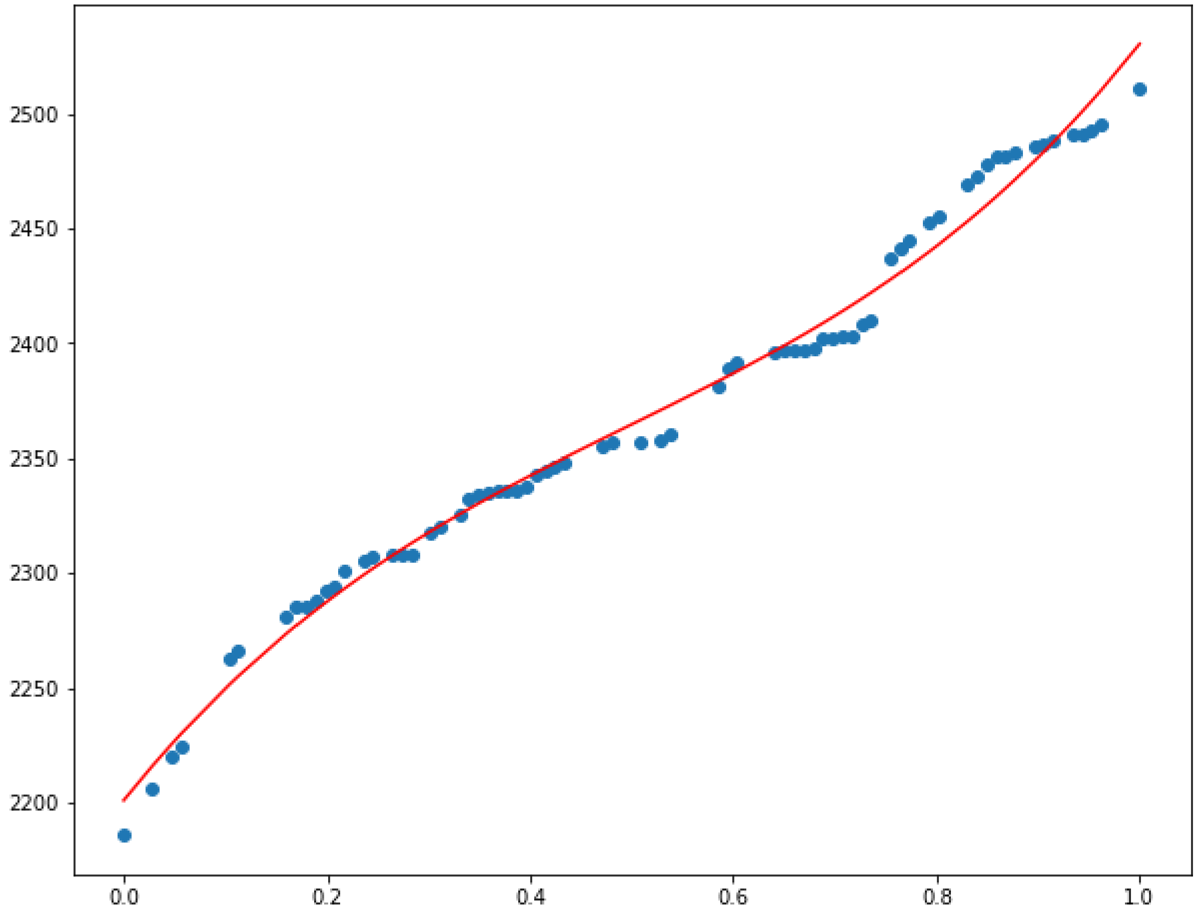
In [ ]:

```
theta, j_arr = grad_desc(theta, x, y)

theta = norm_func(x, y)
```

```
print(theta)

plt.scatter(x[:,1].T, likes)
plt.plot(x[:,1].T, np.dot(x, theta).reshape(1, m)[0], c='red')
```

```
[[2200.93237779]
 [ 549.18692639]
 [-667.79279393]
 [ 448.12902657]]
```

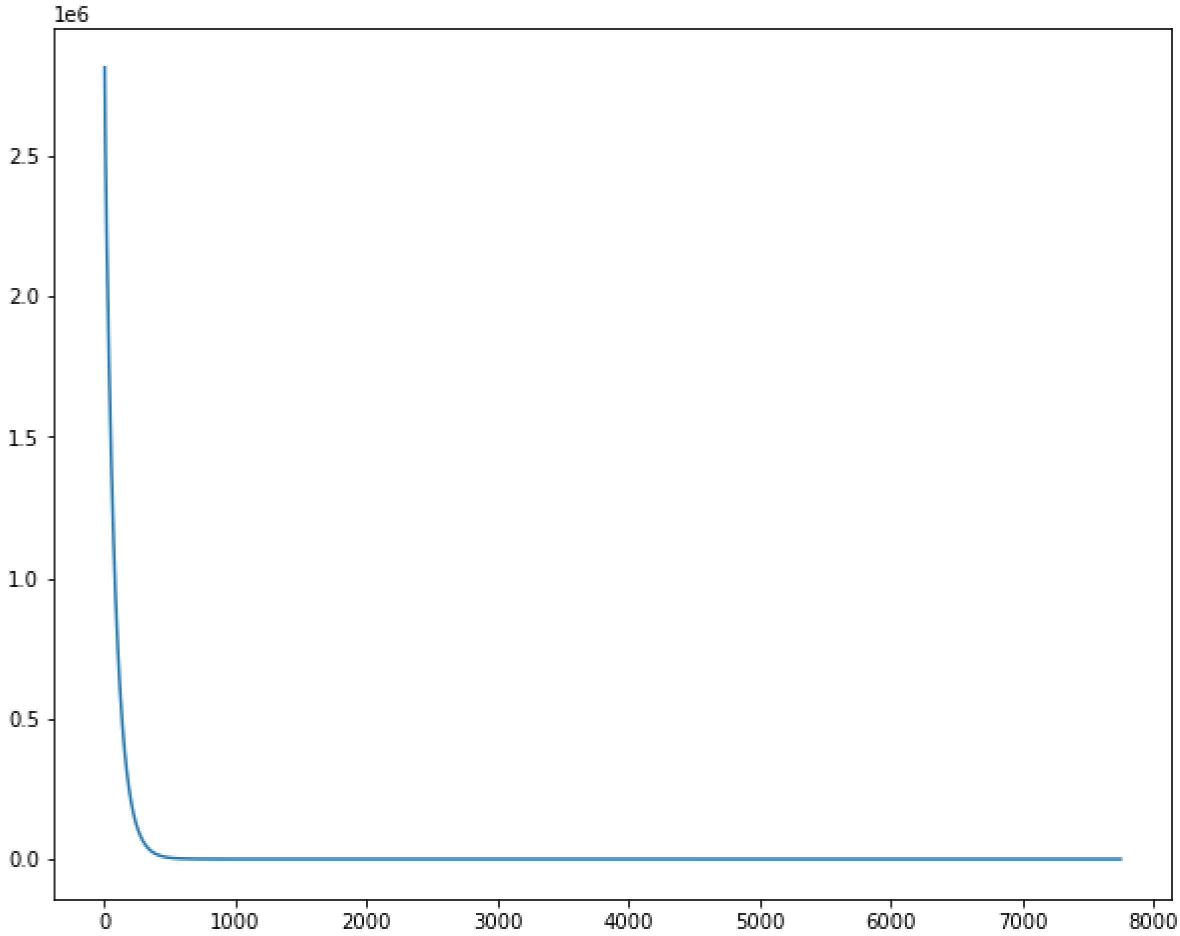
Out[ ]: [<matplotlib.lines.Line2D at 0x7f9cdbd69b90>]



In [ ]:

```
plt.plot(range(1, len(j_arr) + 1), j_arr.reshape(1, len(j_arr))[0])
```

Out[ ]: [<matplotlib.lines.Line2D at 0x7f9cdc0109d0>]



In [ ]:

```
#test
def get_num_likes(day=0):
    global t0, t1, t2
    return t0 + t1 * day ** 0.5 + t2 * day

def cost_func1(theta0, theta1, theta2, x_matrix, y_matrix) :
    return np.sum((theta0 + theta1 * x_matrix ** 0.5 + theta2 * x_matrix - y

r_square = 1 - cost_func1(t0, t1, t2, days, likes) / SSTO
print("R-Squared value for h(x) " + str(r_square))

r_square = 1 - cost_func2(theta, x, y) / SSTO
print("R-Squared value for g(x) = " + str(r_square))
```

```
R-Squared value for h(x) 0.9789513092581201
R-Squared value for g(x) = 0.9876896649613331
```

The R^2 = 0.97895 and 0.98745 which is higher than before. Hence, base on the demand, we can choose a simple or a complicated hypothesis (unless it is underfitted or overfitted). Although higher R^2 can sometimes lead to overfitting.