Gradient Descent

Hudson Arney and Carson Willms

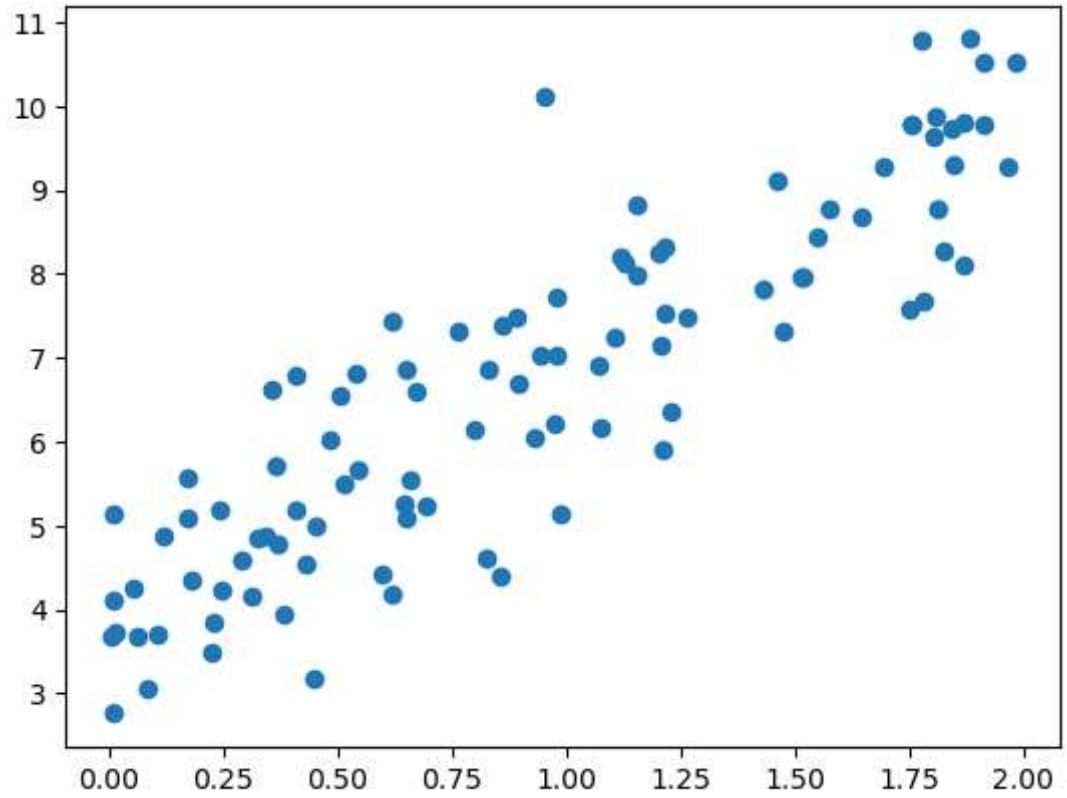
Import Statements

In [1]:
```
import numpy as np
import seaborn as sb
import matplotlib.pyplot as plt
import time
```

Creating a set of random points with a X val random from 1-100 and a y value following the expression y = 4 + 3x with some random offset

In [2]:
```
X = 2 * np.random.rand(100,1)
y = 4 +3 * X+np.random.randn(100,1)
plt.scatter(X, y)
```

Out[2]: <matplotlib.collections.PathCollection at 0x1c7289c2110>



Defining an eval function to determine correctness of prediction (We used MSE as our eval)

In [3]:
```
def cal_cost(theta, X, y):
    '''
    theta - vector of weights for each variable
    X - vector of all independent variables
    y - vector of dependent variable
    '''
    m = len(y)

    predictions = X.dot(theta) #USING DOT PRODUCT TO APPLY WEIGHTS TO EACH INDEPENDENT VARIABLE
    cost = (1/2*m) * np.sum(np.square(predictions-y)) #CALCULATING COST USING MSE
    return cost
```

Gradient Descent Algorithm

Intuitive approach:

1. initialize weights
2. apply weights to independent variables
3. calculate closeness to local optimum
4. tweak weights accordingly
5. repeat

In [4]:
```
def gradient_descent(X,y,theta,learning_rate=.01,iterations=100):
    m = len(y)
    cost_history = np.zeros(iterations) #INITIALIZING ARRAY OF PAST COSTS
    theta_history = np.zeros((iterations, 2)) #INITIALIZING ARRAY OF PAST WEIGHTS
    for it in range(iterations):
        prediction = np.dot(X, theta)
        theta = theta - (1/m)*learning_rate*(X.T.dot((prediction-y))) #TWEAKING WEIGHTS BASED ON MSE OF C
        theta_history[it,:]=theta.T
        cost_history[it] = cal_cost(theta,X,y)
    return theta, cost_history, theta_history
```

Testing Gradient descent (final thetas are very close to the initial y of 4 + 3x)

In [5]:

```
lr =0.01
n_iter = 1000

theta = np.random.randn(2,1)

X_b = np.c_[np.ones((len(X),1)),X]
theta,cost_history,theta_history = gradient_descent(X_b,y,theta,lr,n_iter)


print('Theta0:          {:0.3f},\nTheta1:          {:0.3f}'.format(theta[0][0],theta[1][0]))
print('Final cost/MSE:  {:0.3f}'.format(cost_history[-1]))
```

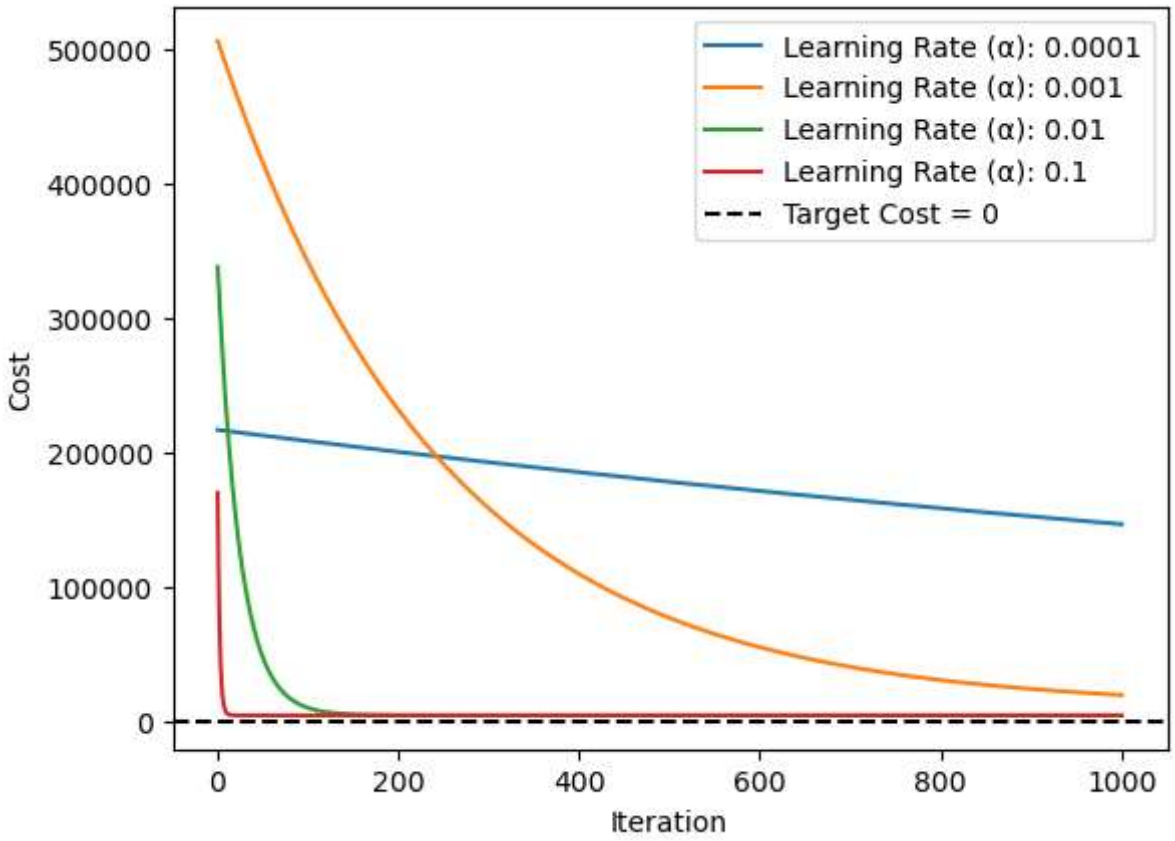
```
Theta0:          3.731,
Theta1:          3.152
Final cost/MSE:  4524.570
```

Benchmarking

The first plot will show the cost function over a set number of iterations, and will compare a group of learning rates. This can be used to determine the "optimal" learning rate as you want to choose one that will converge to 0 in the fewest iterations.

In [6]:

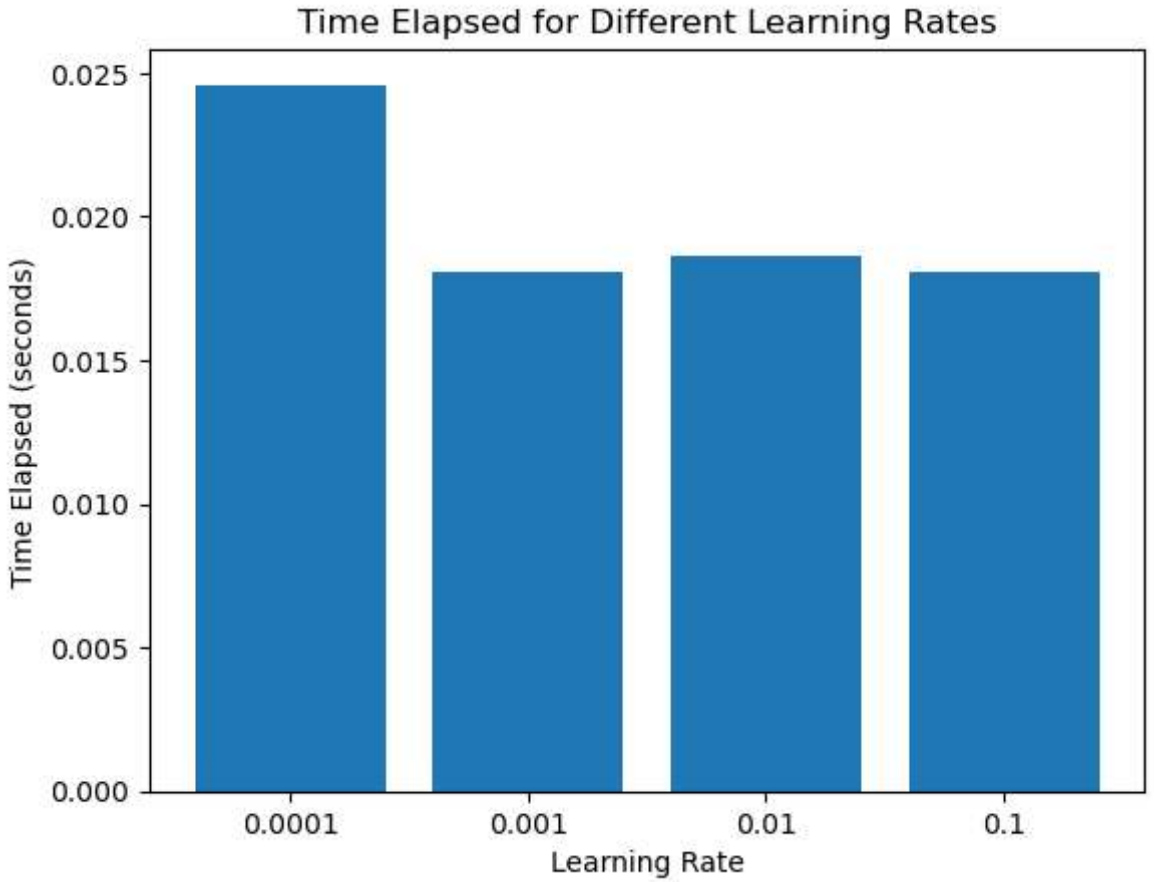
```
learning_rates = [0.0001, 0.001, 0.01, 0.1]
elapsed_times = []

for lr in learning_rates:
    start_time = time.time()

    # Initialize weights randomly
    theta = np.random.randn(2, 1)

    # Run gradient descent
    theta, cost_history, theta_history = gradient_descent(X_b, y, theta, lr, n_iter)

    end_time = time.time()
    elapsed_time = end_time - start_time
    elapsed_times.append(elapsed_time)

    plt.plot(range(1, n_iter + 1), cost_history, label=f'Learning Rate (α): {lr}')

plt.axhline(y=0, color='black', linestyle='--', label='Target Cost = 0')
plt.xlabel('Iteration')
plt.ylabel('Cost')
plt.legend()
plt.show()
```



Here because the alpha value of 0.1 was able to converge to 0 the quickest, it is considered the most accurate learning rate.

In the next plot will be showing the times it takes for specific learning rate values to finish iterating.

In [7]:
```
plt.bar(np.arange(len(learning_rates)), elapsed_times, tick_label=[str(lr) for lr in learning_rates])
plt.xlabel('Learning Rate')
plt.ylabel('Time Elapsed (seconds)')
plt.title('Time Elapsed for Different Learning Rates')
plt.show()
```



This plot shows a trend where the "larger" the learning rate value gets, the faster the algorithm gets. This is because for this case, the convergence of the gradient descent algorithm becomes faster with the larger rate.

- In this case the smaller alpha value are slower because the steps taken during each iteration are tiny. This means it will take a large number of iterations for the algorithm to reach the minimum of the cost function. However this doesn't simply mean we can just increase the alpha value.

- On the other hand, if the learning rate is too large, the algorithm might overshoot the minimum of the cost function. This canlead to failing to converge to the optimal solution.

Entrepreneurial Minded Learning

Gradient Descent can be a great tool/addition to be used for machine learning problems, and is particularly helpful with linear and logistic regression models. One such example could include image recoginition where Gradient Descent helps in optimizing the parameters of the models for recognizing and classifying images. We would like to focus on using gradient descent in medical imaging recogonition.

Business Plan

Value Proposition:

- We want to improve medical image recognition by using gradient descent. Specific applications could come with improving accuracy in medical imaging which could greatly enhance treatment planning.

Customer Segments:

- Our primary target is the healthcare sector, where accurate image recognition is critical. This will require us to collaborate closely with hospitals and medical professionals, while following the healthcare standards requirements.

Revenue Streams:

- Our revenue will come through the sale of the software we use to recoginize the medical imaging, optimized using Gradient Descent. We could also provide training programs about how to use the application for healthcare professionals.

Risk Analysis:

- There will need to be a paramount importance on identifying worst-case scenarios in medical image recognition. The consequences of a false negative in this process are severe, and will have to be addressed using a strategy that will more likely produce a higher false positive rate.