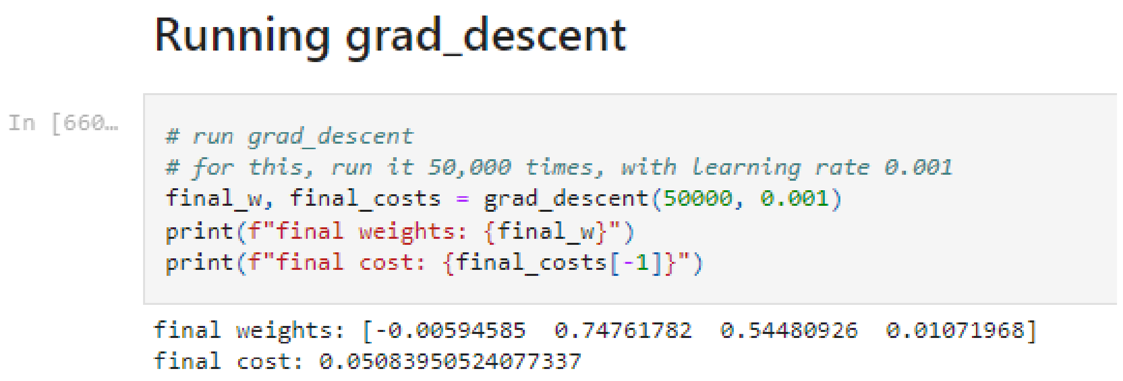
Guides, Emmanuel Tarek Shayne
CMSC 197 (Machine Learning) – HW3
Student ID: 2019-51020
GitHub Link: https://github.com/emmGuides/Machine-Learning

1. What are the optimal weights found by your implemented gradient descent? Plug it into the linear model. What are your interpretations regarding the formed linear model?



## Running grad_descent

```
In [660…

# run grad_descent
# for this, run it 50,000 times, with learning rate 0.001
final_w, final_costs = grad_descent(50000, 0.001)
print(f"final weights: {final_w}")
print(f"final cost: {final_costs[-1]}")
```

```
final weights: [-0.00594585  0.74761782  0.54480926  0.01071968]
final cost: 0.05083950524077337
```

After running grad_descent() for 50,000 iterations on 0.001 learning rate, the final and most optimal weights it has returned have been -0.00594585, 0.74761782, 0.54480926, and 0.01071968. Plugging it in the linear model, we would have:

$$h_0(x) = (-0.00594585) + (0.74761782 * \text{TV}) + (0.54480926 * \text{Radio}) + (0.01071968 * \text{Newspaper})$$

With these weights, we can easily spot that the TV variable contributes the most to the total amount of the predicted Sales value, followed by Radio, then lastly Newspaper.

2. Provide a scatter plot of the $\hat{y}^{(i)}$ and $y^{(i)}$ for both the train and test set. Is there a trend? Provide an r2 score (also available in sklearn).

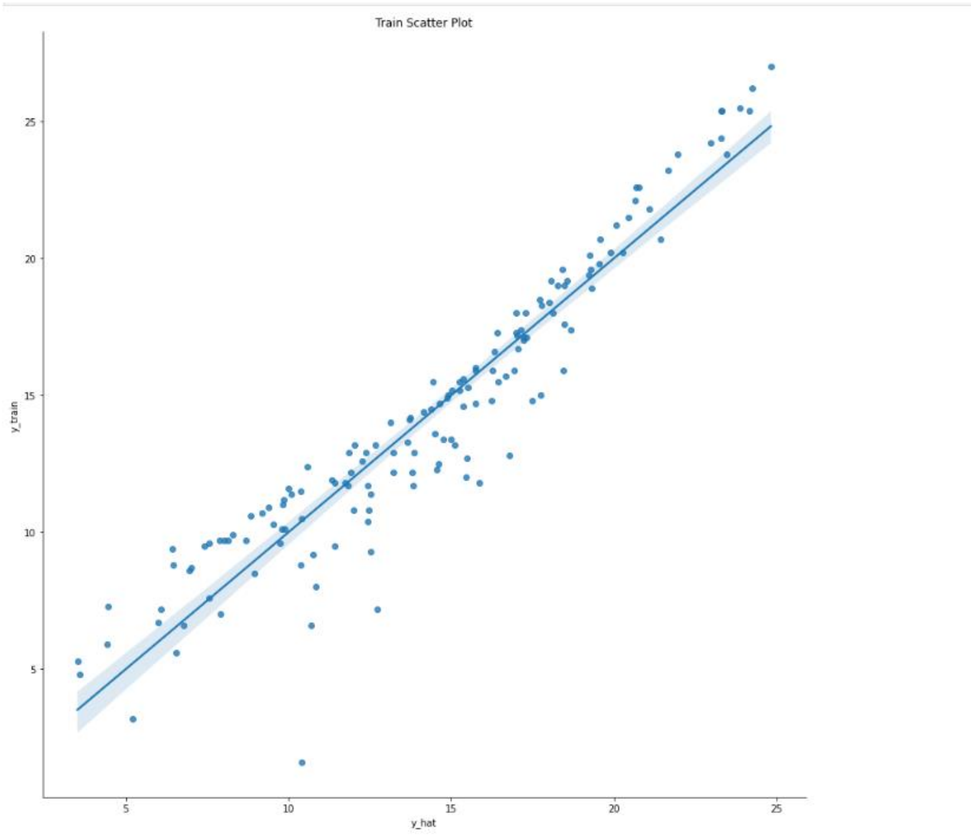


*Figure 1 y-y_hat scatterplot (train)*

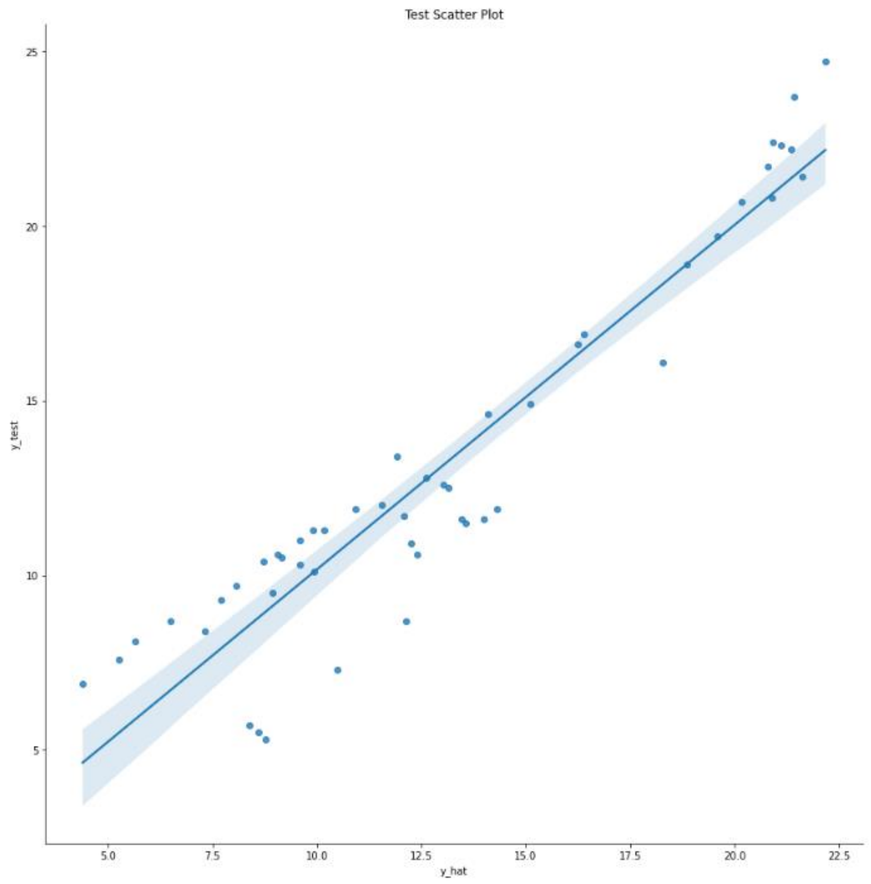
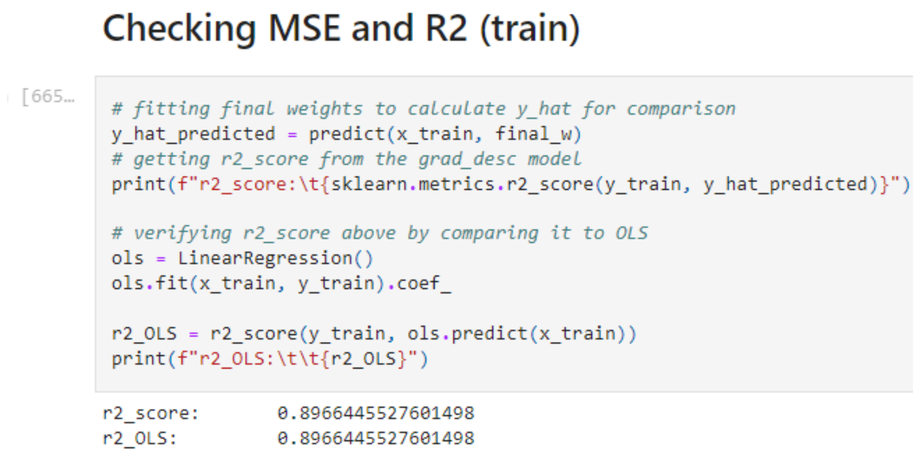


*Figure 2 y-y_hat scatter plot (test)*



## Checking MSE and R2 (train)

[665…

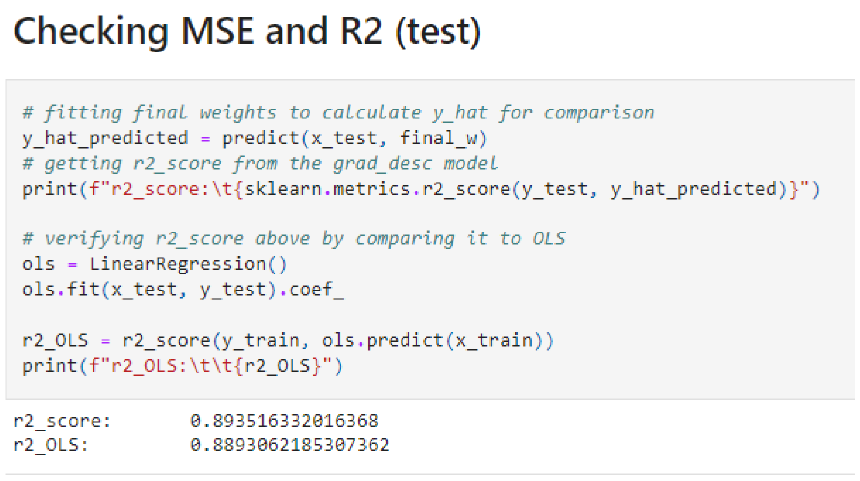
```
# fitting final weights to calculate y_hat for comparison
y_hat_predicted = predict(x_train, final_w)
# getting r2_score from the grad_desc model
print(f"r2_score:\t{sklearn.metrics.r2_score(y_train, y_hat_predicted)}")

# verifying r2_score above by comparing it to OLS
ols = LinearRegression()
ols.fit(x_train, y_train).coef_

r2_OLS = r2_score(y_train, ols.predict(x_train))
print(f"r2_OLS:\t\t{r2_OLS}")
```

```
r2_score:       0.8966445527601498
r2_OLS:         0.8966445527601498
```

*Figure 3 R2 Scores (train)*



## Checking MSE and R2 (test)

```python
# fitting final weights to calculate y_hat for comparison
y_hat_predicted = predict(x_test, final_w)
# getting r2_score from the grad_desc model
print(f"r2_score:\t{sklearn.metrics.r2_score(y_test, y_hat_predicted)}")

# verifying r2_score above by comparing it to OLS
ols = LinearRegression()
ols.fit(x_test, y_test).coef_

r2_OLS = r2_score(y_train, ols.predict(x_train))
print(f"r2_OLS:\t\t{r2_OLS}")
```

```
r2_score:       0.893516332016368
r2_OLS:         0.8893062185307362
```

*Figure 4 R2 Scores (test)*

As seen from the charts above, we can relatively see a linear relationship (and trend) between the y value and the y predicted value. R2 scores on both train and test sets also gave us a score of roughly around 0.89, which means that the model that we have has around 89% accuracy in determining the variability of all response data around the mean.

3. What happens to the error, R2, and cost as the number of iterations increase? Show your data and proof. You can alternatively plot your result data for visualization and check until 50000 iterations or more (actually).

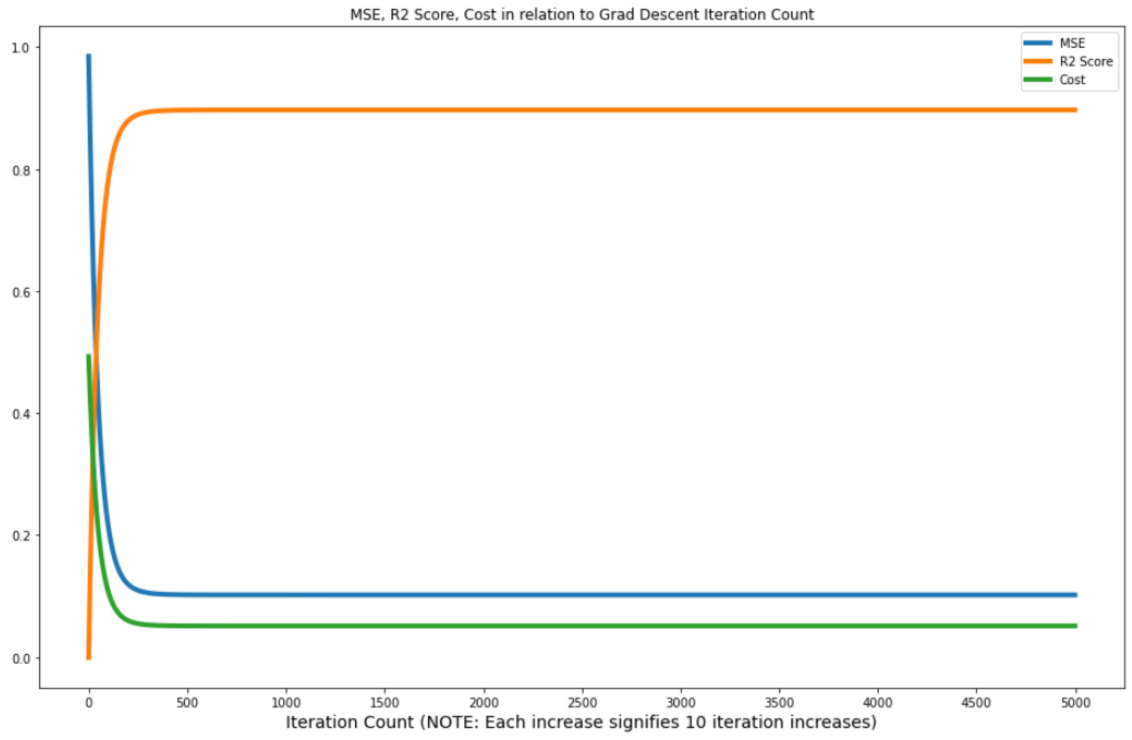


*Figure 5 MSE, R2 Score, Cost*

Over 50,000 iterations, 5,000 length weight caches were retrieved. All 50,000 weight caches were not retrieved since the data rate limit have been reached, as such, weights have only been recorded every after 10 iterations. This handicap, however, did not hinder the appearance of the graph as shown above. In Figure 5, we can see that the MSE and Cost decreases while the R2 Score increases over the total number of iterations committed. This is because the accuracy of the model in predicting the true y values increases in proportion to the learning rate that has been used, which in this case is 0.001, and the number of iterations the algorithm has done, for recalibrating of the weights.

4. Once you determine the optimal number of iterations, check the effect on the cost and error as you change the learning rate. The common learning rates in machine learning include 0.1, 0.01, 0.001, 0.0001, 0.2 but you have the option to include others. Visualize the cost function (vs the optimal number of iterations) of each learning rate in ONLY ONE PLOT. Provide your analysis.

For the following plots, Alpha values 0.001, 0.1, 0.01, 0.5, 0.2, 1e-06, and 1.0 were used. 10,000 iterations were done (1,000 weight caches).

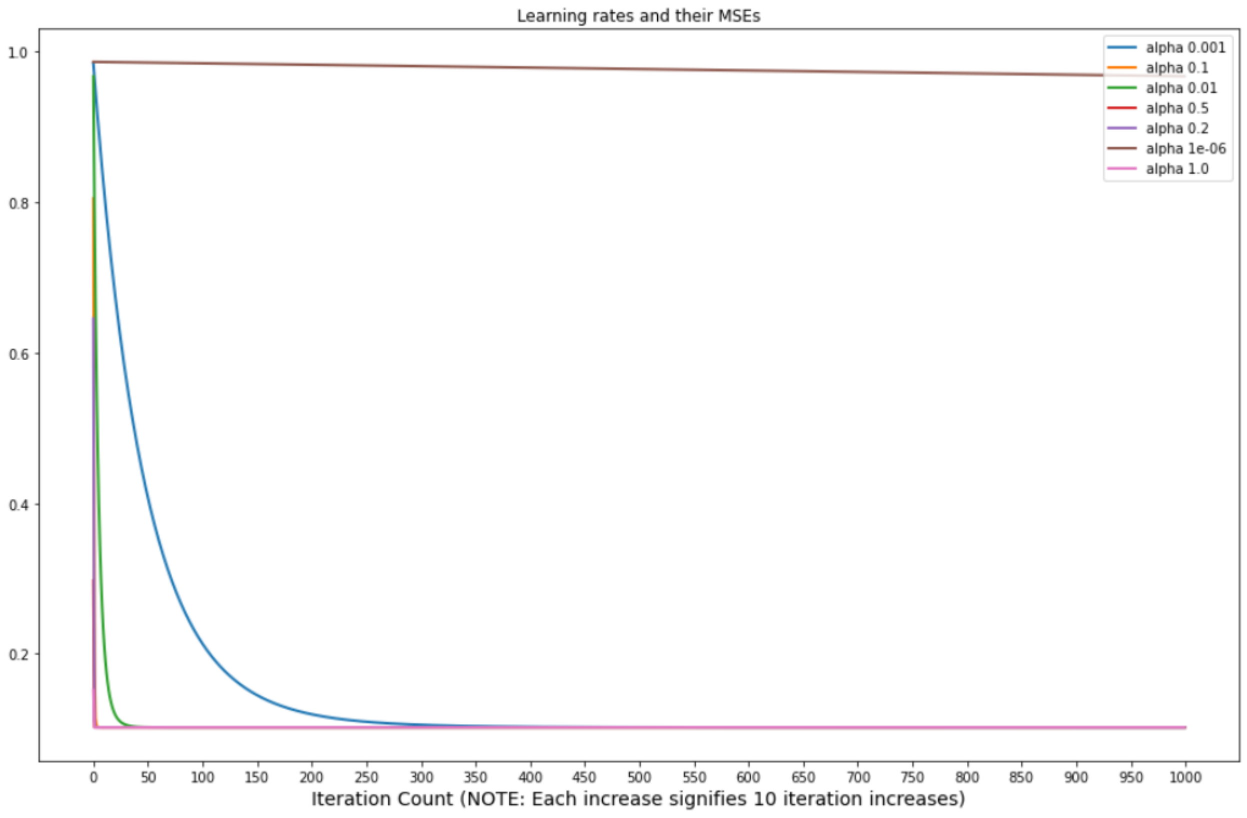


*Figure 6 Learning rates and MSEs (over 10,000 iterations)*

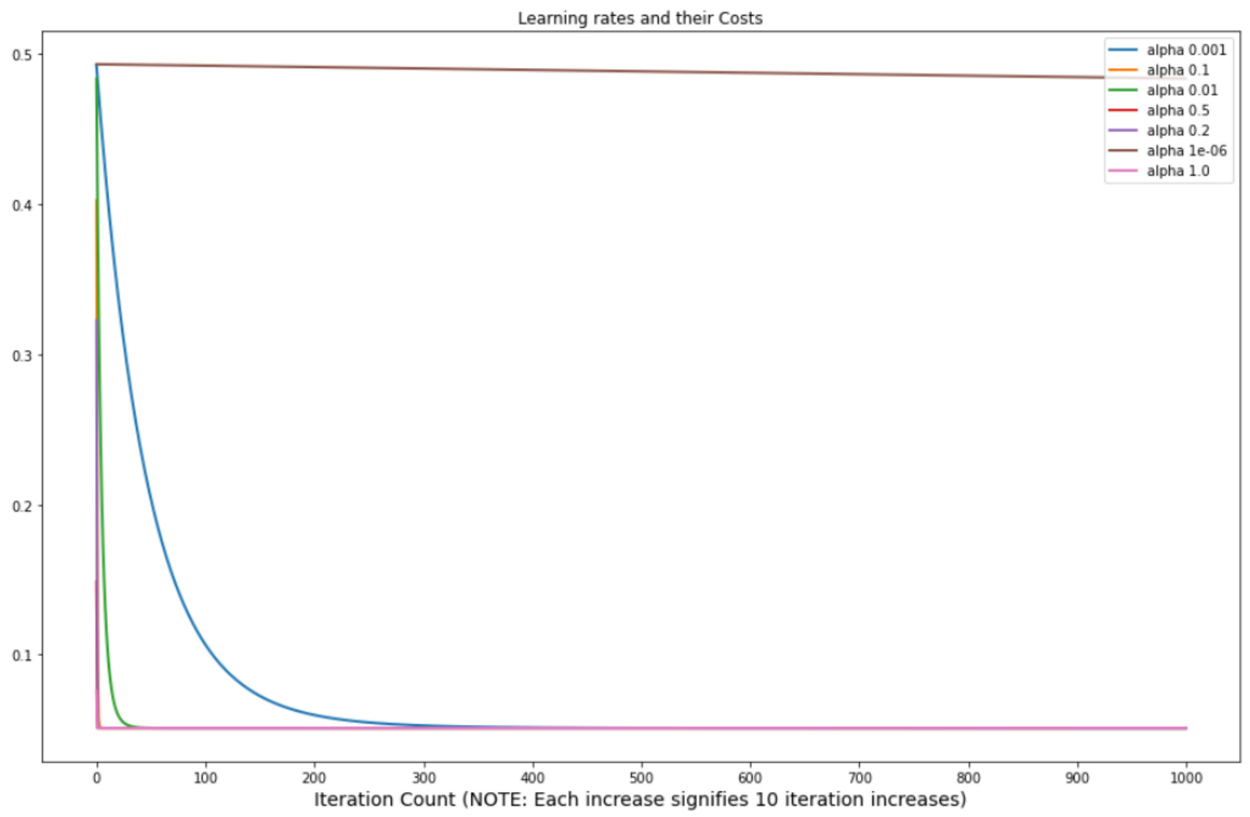


*Figure 7 Learning rate and their Costs (over 10,000 iterations)*

Based on the graphs above, we can see that the learning rate (alpha) drastically influences the rate at which the costs decrease over a number of iterations. Alpha 1e$^{-06}$ (0.000001) showed miniscule changes in the progress of its Costs, rather only showing a slightly tilted horizontal line. Alpha 1.0 on the other hand, quickly dropped towards the least values of both Costs and MSEs. This is because Alpha 1.0 has a larger “step” compared to the other learning rates. This learning rate, however, can be risky as it can be more prone to overshooting as it could’ve easily skipped past the most optimal weight over one of its iterations.

Figures 6 and 7 show the Cost and MSE behavior of all the Alphas over 10,000 iterations. Figures 8 and 9 below show the same graph, but only over 1,000 iterations for a closer look.

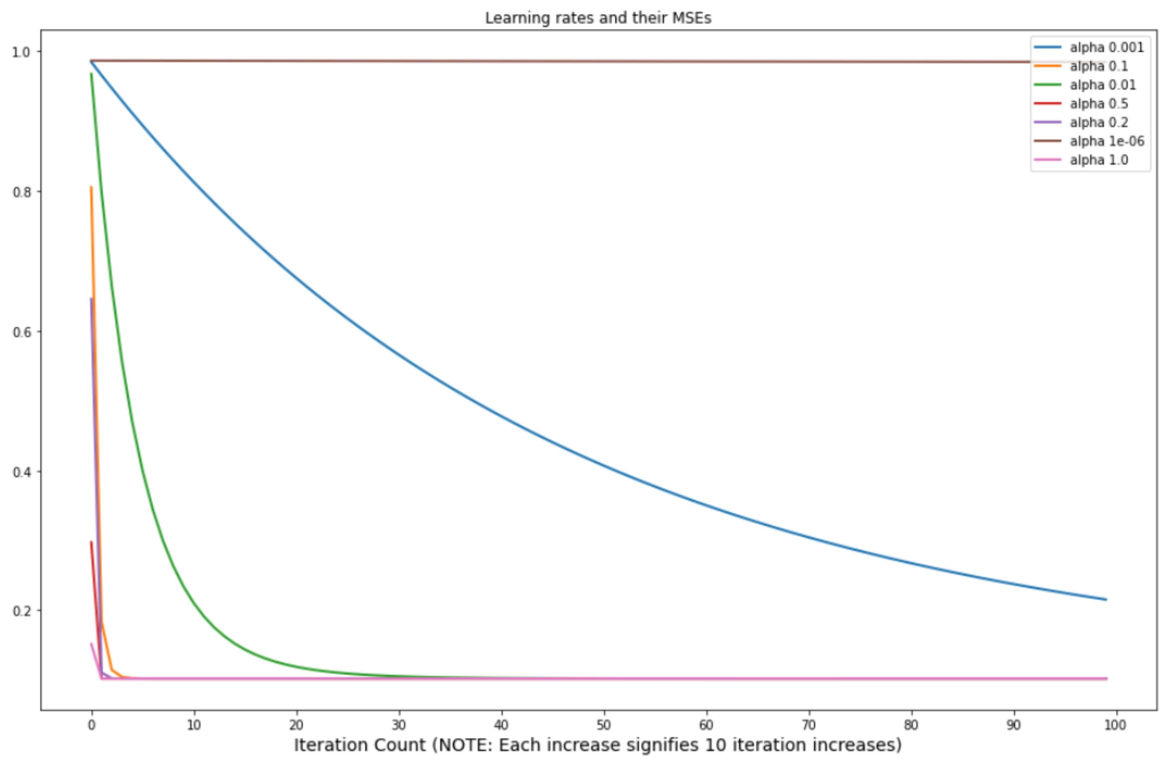


*Figure 8 Learning rate and their MSEs (over 1,000 iterations)*

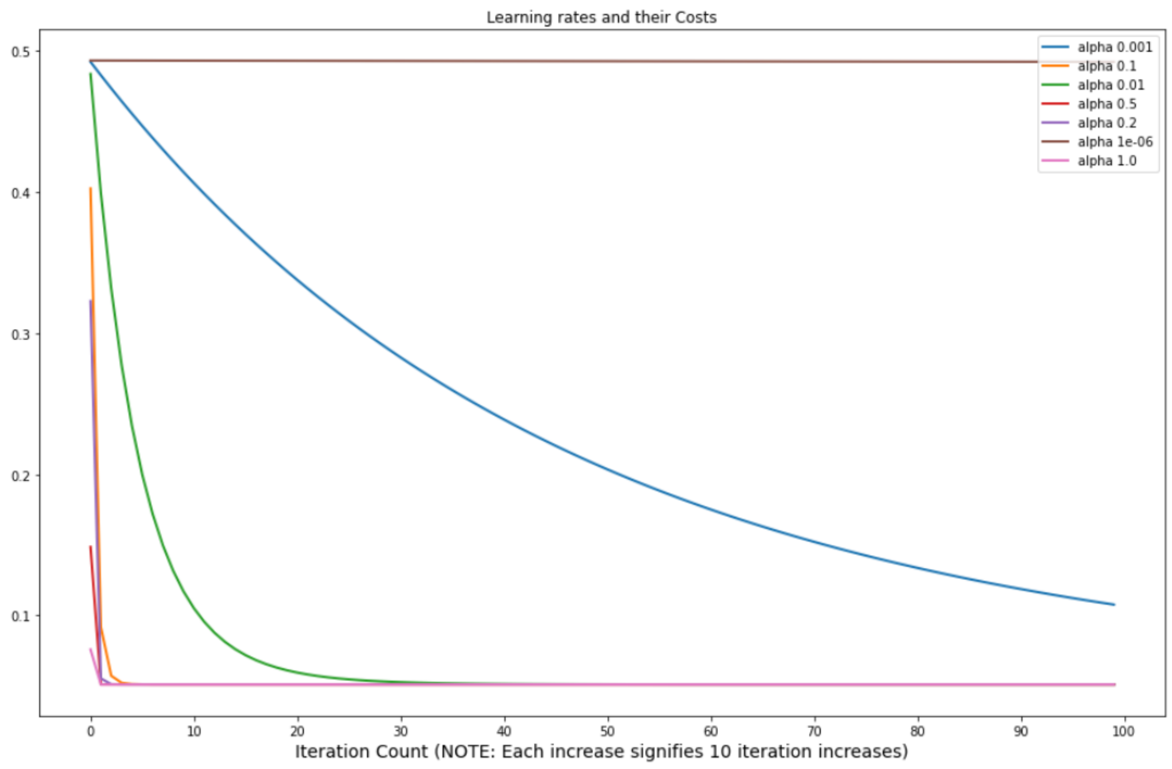


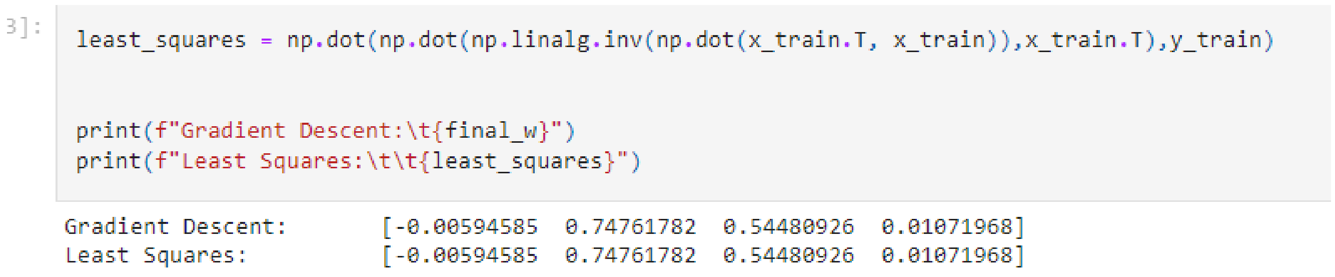
*Figure 9 Learning rate and their Costs (over 1,000 iterations)*

5. Is there a relationship on the learning rate and the number of iterations?

There is a strong relationship between the learning rate and the number of iterations. The lower the learning rate is, for example in the plots above like Alpha 0.000001, the slower the rate at which the cost and MSEs decrease compared to that of other greater learning rates. In instances like these, even more iterations are then needed for it to slowly iterate and finally settle down on an optimized value for the weights to be used in the model.

Greater learning rates like 1.0 take larger leaps on their estimation of the weights, and hence, require lesser iterations. This, however, comes with some caveats; the optimal weights produced by a greater learning rate might not be the most optimal there is, and it can be prone to overshooting. With that said, we can say that with a lower learning rate, the greater the number of iterations and time needed for it to produce a more precise estimate, whereas the higher the learning rate is, the lesser the number of iterations and time that it would need to produce (most probably not optimal) weights.

6. Compare the results with the results of ordinary least squares function.



```
3]: least_squares = np.dot(np.dot(np.linalg.inv(np.dot(x_train.T, x_train)),x_train.T),y_train)


    print(f"Gradient Descent:\t{final_w}")
    print(f"Least Squares:\t\t{least_squares}")
```

```
Gradient Descent:       [-0.00594585  0.74761782  0.54480926  0.01071968]
Least Squares:          [-0.00594585  0.74761782  0.54480926  0.01071968]
```

*Figure 10 OLS and Gradient Descent weights*

As seen in the screenshot above, the optimal weights of the least squares function compared to that of the multilinear regression with gradient descent as the optimizing algorithm produced equal weights of all three features, including the bias.