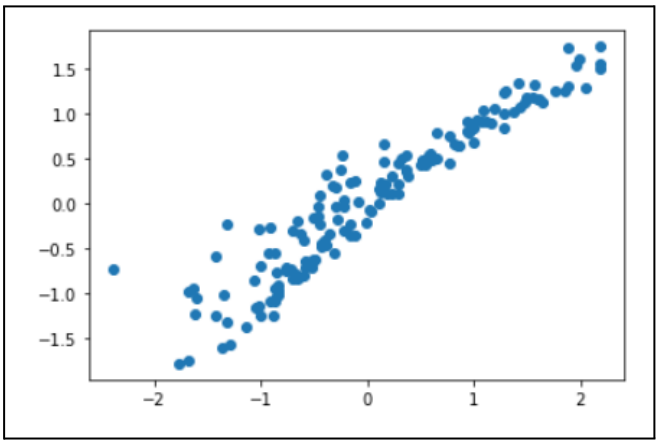
[CMSC 197] - Aren Deza - HW3

1. The optimal weights found by implemented gradient descent using the training dataset are added to the linear regression function here:

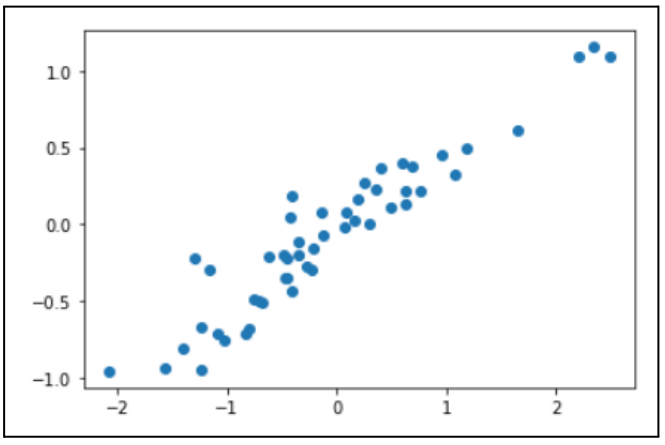
$h_{\theta}(x) =$ (0.009973254791945253)
+ (0.6827554374541572) (**TV**)
+ (0.4357970099774008) (**Radio**)
+ (0.050348952521936406) (**Newspaper**)

The large weight given to TV advertising budget implies that it has a significant effect on the sales rates, with Radio having a similar (if smaller) influence on the sales. By comparison, the very small weight given to Newspaper implies that it only has a minimal effect on sales.
Furthermore, the extremely small bias implies that sales are indeed largely determined by the advertising budgets.

2. Scatterplot for y-hat and y of training dataset



Scatterplot of y-hat and y for testing dataset



As evidenced by the adherence of the above scatterplots to a diagonal linear shape, there is indeed a trend in how the predicted values match the actual sales.

```
from sklearn.metrics import r2_score

# using sklearn to produce the r2 scores of the training and testing datasets
print ('TRAINING r2 SCORE: ', r2_score(trpr['Sales'],trpr['Predicted']))
print ('TESTING r2 SCORE: ', r2_score(tepr['Sales'],tepr['Predicted']))
```

```
TRAINING r2 SCORE:  0.8813043384764145
TESTING r2 SCORE:  0.885845827374728
```

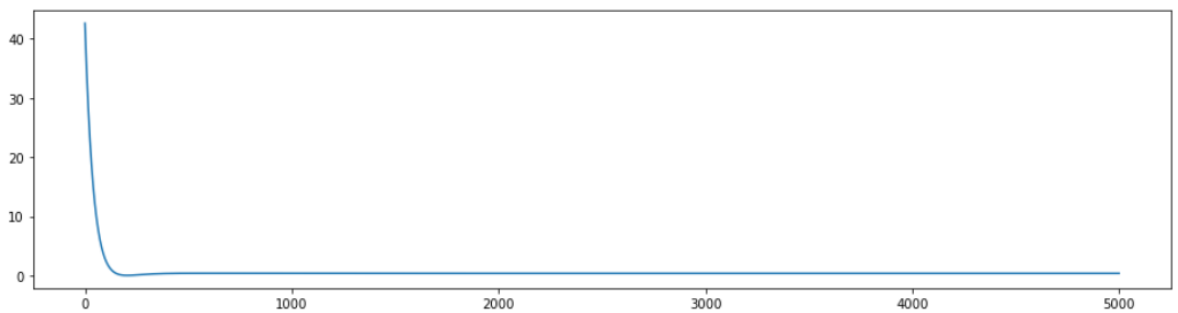
Using the sklearn package, it was identified that the R2 scores of the training and testing datasets are 0.8813043384764145 and 0.885845827374728 respectively

3. Setting the gradient descent function to finish after 5000 iterations instead of waiting for the cost to converge at the first possible point.

```
# doing 5000 iterations

def grad_desc_5k(alpha, x_set, y_set):
    costs = []

    # initial calling of functions to make sure all our variables are in order before looping
    weights = initialize_weights()
    y_hat = predict(weights, x_set)
    cost = compute_cost(y_set.shape[0], y_hat, y_set)
    prev_cost = cost + 1                        # arbitrary value that just needs to be bigger than cost
    w = compute_gradient(y_set.shape[0], alpha, y_hat, y_set, x_set)

    # we're doing this 5000 times instead of waiting for convergence.
    for i in range (0,5000):
        weights = update_weights(weights, w)     # update weights
        costs.append(cost)                       # update costs
        y_hat = predict(weights, x_set)          # predict new value for y-hat
        prev_cost = cost                         # update previous cost
        cost = compute_cost(y_set.shape[0], y_hat, y_set) # calculate new cost
        w = compute_gradient(y_set.shape[0], alpha, y_hat, y_set, x_set)
    return weights, costs
```

```
weights_5k, costs_5k = grad_desc_5k(alpha, x_train, y_train)
plot_costs(costs_5k)
```

```
[<matplotlib.lines.Line2D at 0x1ffd7aedf70>]
```




```
print("5k cost:          ", costs_5k[len(costs_5k)-1], "     number of iterations: 5000")
print("convergence cost: ", costs_train[len(costs_train)-1], "   number of iterations: ", len(costs_train))
```

```
5k cost:           0.35085871197031293      number of iterations: 5000
convergence cost:  3.016247325301351e-06    number of iterations:  205
```

```
3.016247325301351e-06 < 0.35085871197031293
```

```
True
```

As shown in the above pictures,
as the number of iterations increases beyond the initial available point of convergence, the cost initially increases, then steadily decreases over time.

```
other = y_train.copy()
other = pd.DataFrame(other, columns=['Sales'])
other['Predicted'] = (x_train['bias']*weights_5k[0]) + (x_train['TV']*weights_5k
other.head()
```

	Sales	Predicted
0	1.552053	1.254012
1	-0.696046	-0.363758
2	-0.907406	-0.388640
3	0.860330	0.668077
4	-0.215683	-0.121774

```
r2_score(other['Sales'],other['Predicted'])
```

```
0.8957583858078612
```

However, it is also observed that the r2 score increases.
Compared to the r2 scores of the training and testing datasets using the least amount of iterations, this 5000-iteration r2 is slightly larger.

```
other['Error'] = other['Sales']-other['Predicted']
other['Error'].mean()
```

```
-2.4054832200211726e-17
```

```
trpr['Error'] = trpr['Sales']-trpr['Predicted']
trpr['Error'].mean()
```

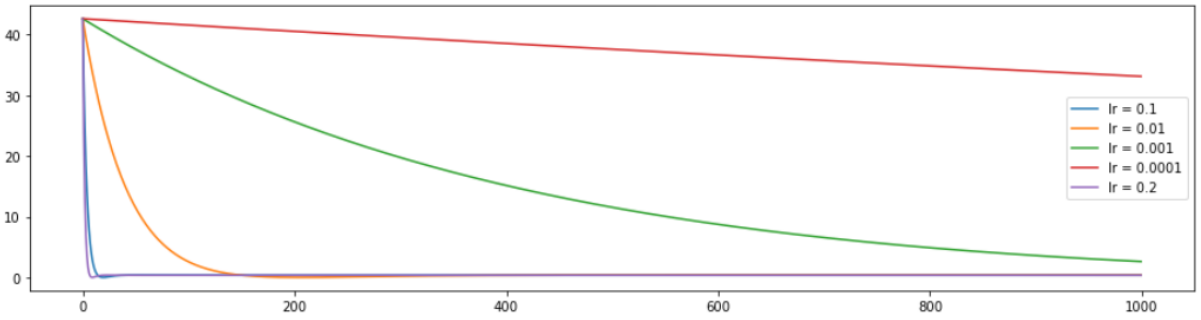
```
0.0005029562702831221
```

```
-2.4054832200211726e-17 < 0.0005029562702831221
```

```
True
```

Finally, when comparing the mean error of the 5000-iteration data, it is much smaller than the error of the predicted data with less iterations.

4.



As shown in the plot here, greater learning rates can reach a state of convergence much more quickly, but are at much greater risk of accidentally increasing the cost by making jumps that are too large for the gradient descent to correct (as seen in the

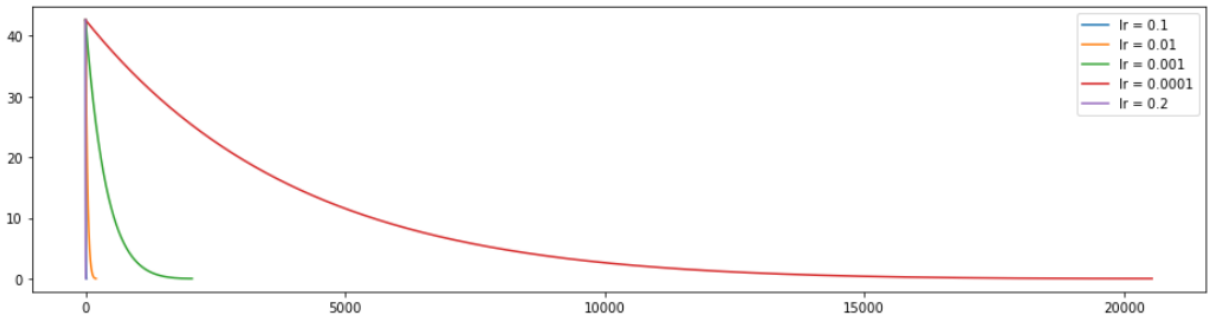
blue and purple lines, were learning rates 0.1 and 0.2 make a sudden increase after reaching a desirably low cost).

On the other hand, smaller learning rates suffer no such issues with accidental overshooting, but they require a much larger amount of iterations to reach a convergent state and are thus more expensive because of it.

Larger learning rates can be more effective if you limit the number of iterations they are allowed to make. Conversely, smaller learning rates are favourable over larger learning rates due to their sensitivity, but they require a large amount of iterations, making them more expensive. In an ideal situation, we would want a learning rate that is just the right size- sensitive enough to avoid overshooting, but large enough to not make the gradient descent take too much time.

5. Learning rate is inversely proportional to the number of iterations. As a smaller learning rate will cause the gradient descent algorithm to progress more incrementally, it will inevitably add more iterations as it slowly moves towards a state of convergence.

   As seen in the graph below, it takes smaller learning rates much more time to reach a state of convergence compared to larger rates.



6. The results are similar to that of the ordinary least squares function as it was demonstrated in the sample codes for our class. It just so happens that gradient descent is much less expensive to run and implement.