Exploring the ”Diamond Price Prediction” Problem with Linear Regression

Akari Ishikawa
163282
a163282@dac.unicamp.br

José Carlos Vasques Moreira
176596
j176596@dac.unicamp.br

I. Introduction

In the present project, we explore the problem of Diamond Price Prediction using the Linear Regression method. Our objective was to do as many experiments as possible and observe how the techniques performed. This report is organized as follows: In the Data Preprocessing section, we briefly describe the dataset and how we modeled the data before starting work on the problem. In the experiments section, we describe how we tested our hypotheses regarding learning rates, data normalization and gradient descent methods and what were our choices for training.

In the training section, we applied the hyperparameters to predict the diamonds' prices and analyzed the results.

And finally, in the Testing section, we predicted the prices of our test set using the model we built, the normal equation and scikit learn.

II. Data Preprocessing

Using the pandas library, we imported the given dataset to a python script.

The original dataset contained 9 features:

- **Carat**: diamond weight in grams.
- **Cut**: Quality of diamond cut.
- **Color**: Diamond color.
- **Clarity**: Clarity of the diamond.
- **Depth percentage**: Diamond physical depth divided by its width.
- **Table percentage**: The width of the largest surface of the diamond divided by the overall width of the diamond.
- **X, Y and Z**: The dimensions of the diamonds in axis x, y and z.
- **Price**: Price of the diamond and our target in this linear regression problem.

We shuffled the dataset to remove any possible bias in its ordering.

The dataset was then split into three smaller sets:

- Training set: 36679 examples
- Validation set: 9170 examples
- Test set: 8091 examples

III. Building the training platform

We wrote a program in C++ in order to take advantage of the increased speed offered by the language. The program is highly versatile, offering several options for increased flexibility in our experiments, such as Batch, Mini Batch and SGD, limiting the run time by time or number of iterations, among others.

IV. Experiments

A. Data Normalization

This experiment can be executed with the following command:

```
python3 exp1_normalization.py
```

It is well known that normalization might increase the training speed, so we conducted an experiment to determine if it would be applicable or not in our dataset, by testing with different alphas in the original and in the normalized dataset.

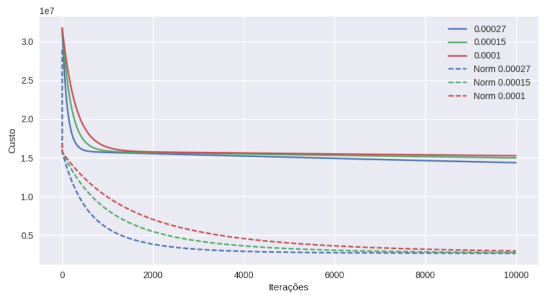

Figure 1. Comparison between normalized and non-normalized data with three small alphas.

We found that, although both datasets seem to have a fast drop in price initially, the non-normalized dataset starts to stagnate at a higher cost, converging much more slowly than the normalized one. Thus, we decided that it would be worthwhile to keep using the normalized dataset.

B. Gradient Descent

This experiment can be executed with the following command:

```
python3 exp2_gradientdescents.py
gradientdescent time
```

where:

- `gradientdescent` can be `batch`, `minibatch` or `sgd`
- `time` is the duration of training in seconds.

In this section, we assess how different gradient descent methods affect our model. We used a simple implementation of the gradient descent algorithm, updating our thetas with $\Theta_j = \Theta_j - \alpha * \sum_{i=0}^{m} (h(x_i) - y_i) * x_i^j$ in each iteration, where the function h(x) is a simple linear function of the shape $h(x) = \sum_{j=0}^{m} x_j * \theta_j$ and α is the learning rate, explained in more details in a further section. In this experiment, we used all nine features and bias, with learning rates 0.2, 0.02, 0.002 and 0.0002.

In the following figures, we can see the cost function during 120 seconds of training for the three methods: Batch gradient descent (Fig. 2), Stochastic gradient descent (Fig. 3) and Minibatch gradient descent (Fig. 4).

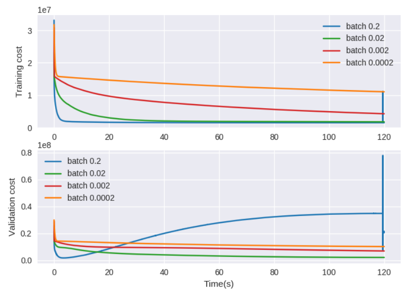

Figure 2. Overview of batch gradient descent with different alphas in 120 seconds of training.

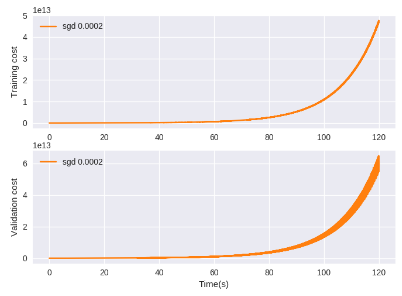

Figure 3. Overview of stochastic gradient descent with different alphas in 120 seconds of training.

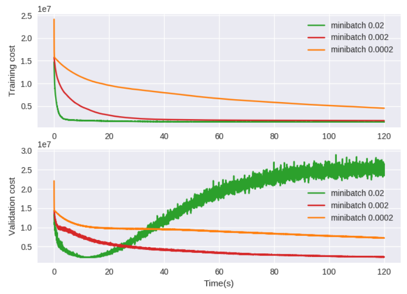

Figure 4. Overview of mini-batch gradient descent with different alphas in 120 seconds of training.

We observed that Stochastic Gradient Descent showed the worst performance of the three methods. We also decided to not consider the 0.0002 learning rate, since it was stabilizing while still at a very high cost. When comparing Minibatch with Batch, Minibatch got lower costs at the first iterations, however, this cost stabilizes while the Batch method continues decreasing. Thus, although Batch gradient descent requires more computational cost, we decided it was the best method for our problem.

	Cost	0.2	0.02	0.002
Batch	Training	1512152.37	1772587.625	4271472.5
	Validation	21147442.0	2204720.5	6994207.5
SGD	Training	Diverged	Diverged	Diverged
	Validation	Diverged	Diverged	Diverged
Mini Batch	Training	Diverged	1520765.75	1795085.875
	Validation	Diverged	26540460.0	2269600.0

Based on these conclusions, we see that batch gradient descent, while having problems with diverging after a point, is still the quickest to converge to an optimal cost.

C. *Learning Rates*

This experiment can be executed with the following command:

```
python3 exp3_learningrates.py
```

Another important parameter in Linear Regression is the **learning rate**. It determines how fast the model parameters (thetas) converge or diverge. A learning rate too small makes our model converge slowly and a large learning rate may lead to divergence. The point in this experiment is to find the largest α that makes our model converge.

In the following results (figure 5, 6), we show the comparison between different learning rates using batch gradient descent.

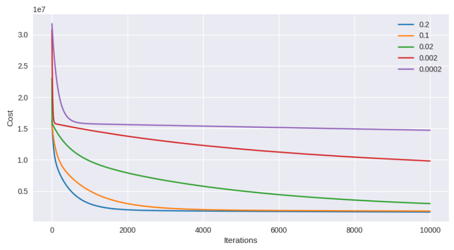

Figure 5. Comparison between different learning rates in the training set. In this experiment, α = 0.2 is likely to be the best choice.

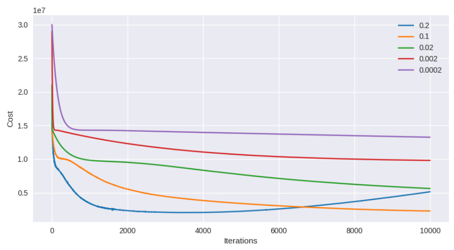

Figure 6. Comparison between different learning rates in the validation set. We can note that α = 0.2 which was, in the training set, the best learning rate, seems to have caused an overfitting in the model.

Based on the behavior of the learning rates in the validation set, we decided to use α = 0.02.

V. Training the model

A. *Training without feature engineering*

This experiment can be executed with the following command:

```
python3 exp4_training.py iterations
alpha gradient
```

where:

- `iterations` is the number of iterations of the training
- `alpha` is the learning rate
- `gradient` can be `batch`, `minibatch` or `sgd`

Based on the previously cited experiments, we used the following settings on the training step:

- Learning rate α = 0.02
- Batch Gradient Descent
- Normalized dataset

However, we did not know how many iterations were necessary to train the model. Thus, we started training with 10000 iterations. As you can see in figure 7, both training and validation cost decreased, leading to an error of 5606750.50.

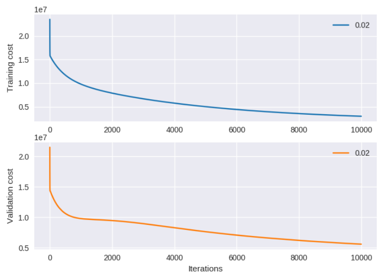

Figure 7. Training for 10000 iterations using batch gradient descent and α = 0.02

In order to make our model more complex, we trained for 50000 iterations. The results improved, the error decreased to 2383383.00 and the predictions got closer to the expected values (Fig. 8).

This last training, with 50000 iterations, led to our final model, with the following thetas:

Table I
PARAMETERS OF THE BUILT MODEL.

Theta 0	-13482.58
Theta 1	30360.47
Theta 2	-252.90
Theta 3	1995.66
Theta 4	4070.75
Theta 5	-15241.46
Theta 6	-13689.23
Theta 7	14877.20
Theta 8	-3987.87
Theta 9	-7404.62

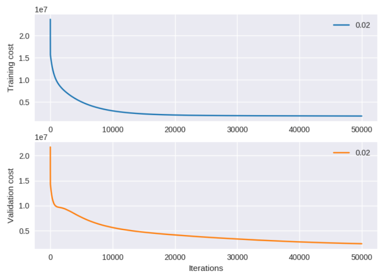

Figure 8. Training for 50000 iterations using batch gradient descent and α = 0.02

B. *Feature Engineering*

This experiment can be executed with the following command:

```
python3 exp5_training.py iterations
alpha gradient
```

With the model evaluated, we tried to improve it by dropping some features and changing the polynomial degree of others. Firstly, we tried to drop **depth** and **table**, but to no avail. Then, we changed the degree of the x corresponding to the weight (**carat**) to x^2, however, it caused no significant changes to the model.

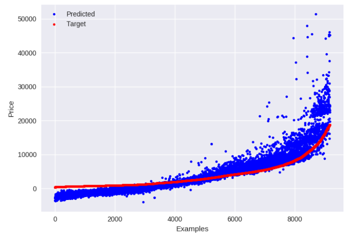

Figure 9. Training for 50000 iterations using batch gradient descent and α = 0.02 while adding new degrees and new features.

None of our trials had any success. They all made our validation set worse. For example, in figure 9, where we tried to square some features (Cut, color, clarity and carat) and also add their harmonic mean as a separate feature, in an attempt to better model a diamond and the different relations between these four main qualities. This didn't work out, leaving us with a much worse cost of 9188263.0

VI. Testing

This experiment can be executed with the following command:

```
python3 exp6_testing.py
```

Our best result until now was using batch gradient descent, with a learning rate of 0.02, with 50000 iterations and all 10 original features. We generated a model with these settings, the theta found can be seen in Table I, and the prediction is shown in figure 10.

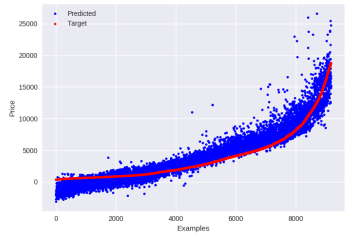

Figure 10. Final prediction in the test set.

The final prediction has an error of 2371867.08, similar to the one obtained with the validation set (2383383.0).

A. Normal Equation

This experiment can be executed with the following command:

```
python3 exp7_normalequation.py
```

For one of our last comparisons, we obtained the parameters calculating the normal equation from our training set. The thetas obtained (Table II) are very close to the ones generated by our experiments and produced an error of 1037569.26. The prediction can be seen in figure 11.

Table II
PARAMETERS OBTAINED FROM NORMAL EQUATION.

Theta 0	-6882.08
Theta 1	38719.65
Theta 2	-79.264527
Theta 3	2111.513635
Theta 4	4011.154696
Theta 5	-19022.168443
Theta 6	-10559.75
Theta 7	7600.49
Theta 8	-16750.40
Theta 9	2514.25

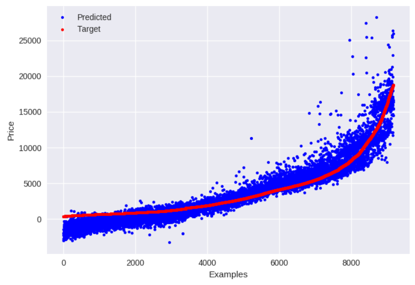

Figure 11. Prediction using the model obtained from normal equation.

B. Training with scikit-learn

This experiment can be executed with the following command:

```
python3 exp8_sklearn.py
```

As suggested by the assignment, we trained a model using `scikit-learn`, a python framework and compared its results with ours. It is important to emphasize that the suggestion was to use Stochastic Gradient Descent in scikit-learn. Since we were using Batch Gradient Descent, we couldn't use the same learning rate in both experiments. When using scikit-learn, we set $\alpha = 0.0001$.

The model built by scikit-learn has an error of 1928168.16, very close to ours. The prediction is shown in figure 12.

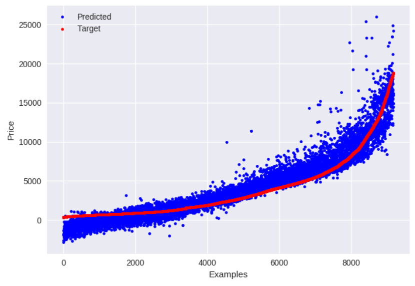

Figure 12. Prediction using the model obtained with scikit-learn.

VII. Conclusions

In the end, some of our results don't line up with what we expected when we first started the experiments. While there seems to be some correlation between weight and price, there is too much variation in the prices, leading us to believe that either there are very complex interactions between all the features of a diamond, or diamonds have very artificial prices, subject to a personal bias from whoever is pricing them at the moment. It would be worthwhile to do more in-depth research on how diamonds are actually price.