Traffic Volume Prediction with Linear Regression

Carlos Avelar
168605
c168605@dac.unicamp.br

Tiago Chaves
187690
t187690@dac.unicamp.br

***Abstract*—In this project we investigate whether Linear Regression is capable of handling the "Metro Interstate Traffic Volume Data Set". We tested a few variations of the Gradient Descent (GD) algorithm to progressively improve our prediction model, as well as the Normal Equation for an analytical approach, achieving reasonable and comparable results.**

Considering the models used and the amount of data, the best results on the validation set were achieved by our implementation of Stochastic GD, closely followed by our Mini Batch GD trained with a batch size of 173, and scikit-learn's SGDRegressor.

However, the analytical solution obtained with the Normal Equation differed by only 1%, while being 10 times quicker to compute, making it the best solution for the given problem when time is a valued factor.

I. Introduction

Linear Regression (LR) is a basic supervised learning technique that is perfect for simple data modeling, as it is generally simpler than other Machine Learning methods like neural networks, and is able to achieve similar results in these cases. However, Linear Regression has its limitations, such as the need to explicitly define the model, select features and its impossibility of handling very complex data.

Here we use LR to find a linear relationship between a target variable y (output), and one or more features x (inputs), i.e.: $y \approx h_\theta(x)$, such that:

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1 + ... + \theta_n x_n \tag{1}$$

II. Problem

Many techniques for adjusting models to data are available, from Least Squares [1] to Neural Networks, with each one being best suited to handle data of some complexity and size.

The aim of this report is to evaluate the applicability of Linear Regression to a given problem: our challenge is to create a model to the "Metro Interstate Traffic Volume Data Set" [2], that has 48204 examples with 8 features and one value to be predicted. The features include date, time, holidays and weather, and the value we want to predict is the traffic volume.

To quantify and compare the quality of our models we will be using the Mean Squared Error (MSE):

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \tag{2}$$

$$\implies \frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} \left(\left(h_\theta(x^{(i)}) - y^{(i)}\right) x_j^{(i)} \right) \tag{3}$$

where:

- m is the number of training examples
- n is the number of features in the dataset
- $x = (1, x_1, ..., x_n)$ are the dataset features (plus a constant $x_0 = 1$, relative to the intercept term)
- $y = (y^{(1)}, ..., y^{(m)})$ are the target values
- $h_\theta(x)$ is the predicted value of y (i.e. the hypothesis)
- $\theta = (\theta_0, \theta_1, ..., \theta_n)$ are the LR model parameters
- $y^{(i)}$ is the target value for the i^{th} training example
- $x^{(i)}$ are the features of the i^{th} training example, with $x_j^{(i)}$ being its feature j value

III. Proposed solution

A. The Gradient Descent Algorithm

To obtain the best possible model for our problem we must minimize the difference between the real traffic volumes y and the predicted value $h_\theta(x)$, thus our goal is:

$$\underset{\theta}{\text{minimize}}\ J(\theta) \tag{4}$$

which can be approximated with the Gradient Descent (GD) search algorithm [3].

We first start with an initial vector θ, and then iteratively update it by changing its parameters to reach a minimum:

repeat $\{\theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}\}$ (simultaneously update for all j)

for a given learning rate value α, which controls the size of our "steps" in the model parameter space.

If we consider a matrix $\mathbb{X}$, with each row i equal to $x^{(i)}$ transposed ($1 \leq i \leq m$), we have:

$$\theta := \theta - \alpha \left(\frac{1}{m} \mathbb{X}^T (\mathbb{X}\theta - y) \right) \tag{5}$$

B. Gradient Descent Variants

There are three ways to implement the GD algorithm [4]:

1) Vanilla Gradient Descent (VGD): The whole training set is used for each step of gradient descent, so every update is an epoch.

2) Stochastic Gradient Descent (SGD): Each step uses exactly one example, so it takes m updates of θ for an epoch.

3) Mini-batch Gradient Descent (MBGD): Each step of gradient descent uses b training examples (where $1 < b < m$ is the batch size).

While VGD will converge straight towards the global minimum of a convex function, it's slow to do so, since we have to run through every example before updating θ. SGD, on the other hand, is generally way faster to get close to the minimum for larger datasets (i.e. big m), since it updates the parameter

values more frequently, but it may never actually reach it, as θ may keep indefinitely oscillating.

Therefore, MBGD tries to be the best of both worlds, and is specially useful when dealing with a number of examples so big that we can't keep them all in memory at the same time.

C. Normal Equation

There is, however, an analytical method to solve 4, since the loss (cost) function $J(\theta)$ we will use is the MSE, namely, the normal equations:

$$\mathbb{X}^T\mathbb{X}\theta = \mathbb{X}^T y \tag{6}$$

$$\implies \theta = (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T y \tag{7}$$

Alas, the matrix $\mathbb{X}^T\mathbb{X}$ can be noninvertible (specially if we have redundant or too many features), and besides that, calculating it's inverse is computationally expensive.

IV. Data Preprocessing

The dataset file has 48204 rows and 8 columns: holiday, temp, rain_1h, snow_1h, clouds_all, weather_main, weather_description, and date_time [2].

We first removed 11 invalid entries, since either their temperature was 0 Kelvin or they had a value of over 1000mm of rain in an hour, so we could confidently discard entries like these in test – never seen – data.

Then, after shuffling the data to avoid ordering related biases, we split it into:

- Training set (70%) – 33735 examples
- Validation set (15%) – 7229 examples
- Test set (15%) – 7229 examples

Since 99.87% of samples showed 0mm of snow in an hour, we removed the snow_1h column, together with weather_description, as they seemed to be redundant when we also have the weather_main values.

Besides that, we also did some feature extraction:

- Holiday names were only set on the first hour of the day, so we expanded them to the whole day and gave them a binary value (1 when there was a holiday and 0 otherwise), to make the column numerical.
- We divided the date_time column into three – hour, weekday and weekofyear – as we believed the split values would be more meaningful for prediction.
- The categorical feature weather_main was encoded as a one-hot vector and one of the resulting columns was removed to avoid the Dummy Variable Trap [5].

Finally, the three sets were Z-Score [6] normalized, with the care of also using the mean and standard deviation values of the training set on the validation and test sets.

This resulted in a (*v1*) model with 18 features. We later created a more complex model, turning the hour, weekday and weekofyear into their sine and cosine transformations, so that we could capture their cyclic nature [7], resulting in a new (*v2*) model, with 21 features.

V. Results

We have implemented and tested the following methods for the optimization of the parameters that predict our target value:

- Vanilla Gradient Descent (VGD)
- Mini-batch Gradient Descent (MBGD)
- Stochastic Gradient Descent (SGD)
- Normal Equation
- scikit-learn's SGDRegressor [8]

We have achieved very similar results with all of them when using the same fixed learning rate of 0.0005, however the number of epochs differed with the method, in order to have each run take around 10 seconds (this is due to VGD and MBGD taking more advantage of vectorized operations as they run more training examples at once, compared to SGD's evaluating one sample per iteration in a for-loop).

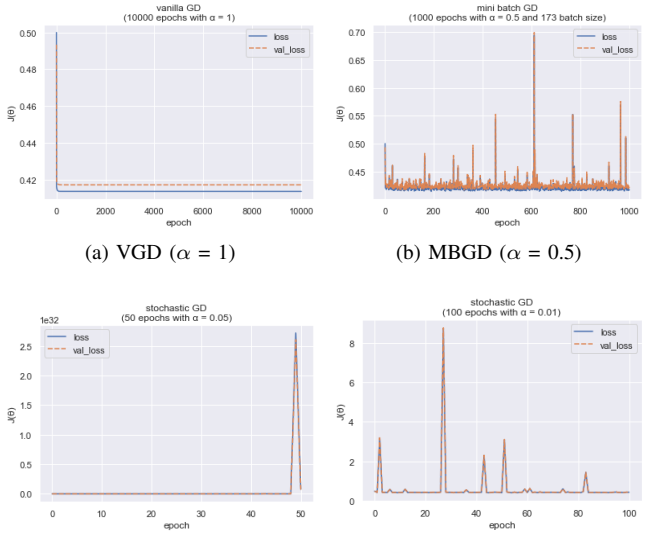
After some initial tests with the simpler undefitting model, that got an R^2 score of 0.15, we transformed time related features to better represent their cyclic nature. The model with this small changes was able to get to an R^2 score of 0.65, although it continued to undefit the training data.

A. Higher learning rates

Values larger than 0.0005 were also tested for α, so we could see when each variant of GD would become unstable and, eventually, unusable.

As expected, VGD could get solid results even with $\alpha = 1$, while MBGD would start becoming unstable with $\alpha = 0.5$ and SGD would explode even when the learning rate was set to 0.05, as shown in Figure 3.

Therefore, we decided to maintain a small learning rate to assure convergence.



(a) VGD (α = 1)

(b) MBGD (α = 0.5)

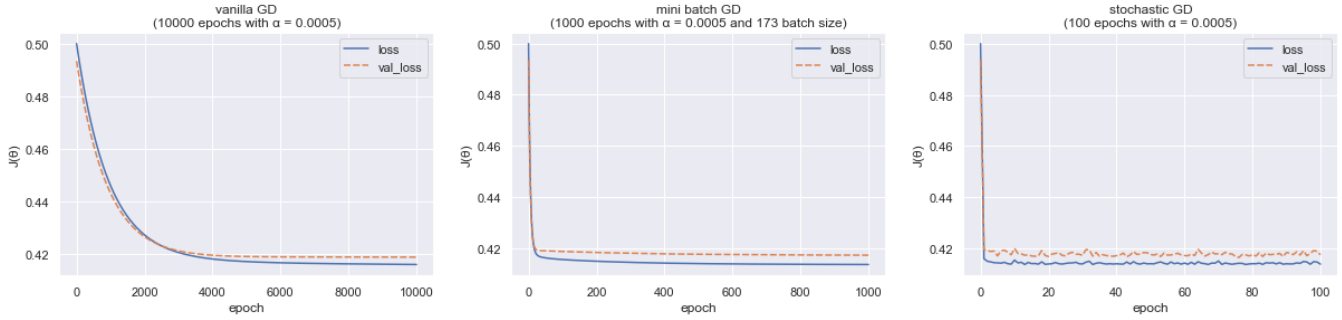
(c) SGD (α = 0.05)

(d) SGD (α = 0.01)

Figure 3. Tests on the *v1* Model with learning rates higher than 0.0005

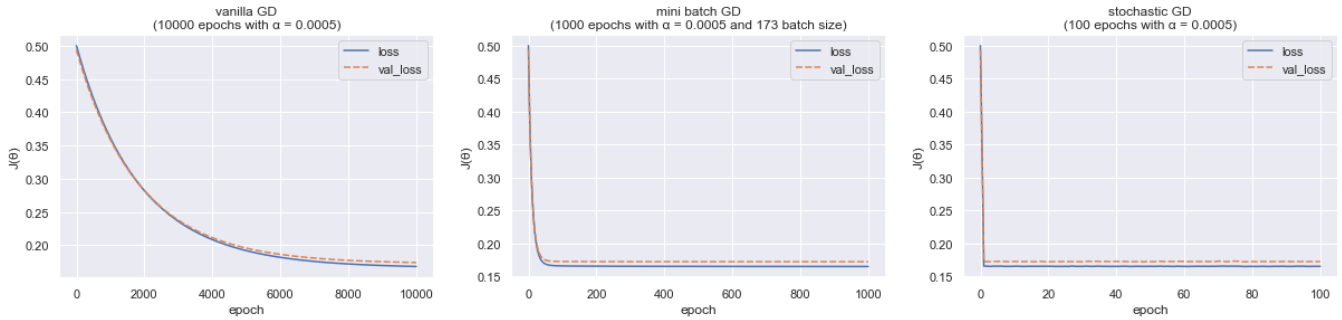
B. v1 Model (18 features)

The results for the GD variations are shown in Figure 1. When solving for θ using the Normal Equation (7) with the



(a) VGD (best validation loss = 0.4190) (b) MBGD (best validation loss = 0.4173) (c) SGD (best validation loss = 0.4165)

Figure 1. Gradient Descent variations used for training on the *v1* Model, together with the lowest values of $J(\theta)$ on the training set



(a) VGD (best validation loss = 0.1741) (b) MBGD (best validation loss = 0.1718) (c) SGD (best validation loss = 0.1717)

Figure 2. Gradient Descent variations used for training on the *v2* Model, together with the lowest values of $J(\theta)$ on the training set

training set values, the loss $J(\theta)$ on the validation set was 0.4171, smaller than VGD and MBGD, and higher than SGD, but computed over 10 times quicker (since we were dealing with a small amount of features).

C. v2 Model (21 features)

The only changes from *v1* to *v2* were that the hour, weekday and weekofyear columns were converted into: hour_cos, hour_sin, weekday_cos, weekday_sin, weekofyear_cos and weekofyear_sin. Nonetheless, this more than halved the value of $J(\theta)$ on the validation set, as seen on Figure 2.

As the model was still relatively small, the Normal Equation remained faster than GD, reaching 0.1718 on the validation set, the same as MBGD and a little worse than SGD.

D. scikit-learn's SGDRegressor

To mimic our implementation as best as we could, for better comparison, we set the following parameters when using scikit-learn's implementation:

```
sklearn.linear_model.SGDRegressor(
    loss="squared_loss",
    penalty="none", # regularization term
    max_iter=1000,
    tol=None, # stopping criterion
    learning_rate="constant", # uses eta0
    eta0=0.0005,
    early_stopping=False)
```

After fitting it to our training set, the predicted values on the validation set had a loss value of 0.4173 on *v1* and 0.1717 on *v2*, both close to our implementations, but a little worse than what our SGD achieved.

VI. Discussion

After starting with a small 0.0005 learning rate, we tested a few other values to see how the different methods responded, with each one behaving differently.

The Vanilla GD is not very sensitive to high learning rates, this happens because it uses the entire training set, so its gradient is not violently altered by outliers and more deviant data points. It handled up to $\alpha = 1$, which sped up convergence considerably, however we couldn't get higher learning rates tested because of numerical explosion.

The Mini-batch method is a bit more sensitive, because it uses smaller groups of data to train, so its higher limit is lower, being 0.5 already too high (see Figure 3), making the gradient too unstable., and usable values begin at $\alpha = 0.1$.

Finally, Stochastic Gradient Descent is very sensitive to changes in the learning rate, as it updates the parameters for every sample in the training data. At 0.01 learning rate it would diverge and at 0.5 it simply exploded. Well behaved graphs of loss function *vs.* number of epochs were only seen when α was lower than 0.001.

We have also run the scikit-learn implementation SGDRegressor, obtaining slightly different results compared to

our implementations, although it took less computation time, which was already expected of a popular and highly optimized library.

The Vanilla GD is by far the slowest method to converge, needing around 10.000 epochs to get values close to MBGD and SGD. It is followed by Mini-batch GD and finally by Stochastic GD, which is the fastest. This makes sense as the larger the batch size, the closest to VGD Mini-batch becomes, and the lower its size, the closest to SGD.

It is also interesting to notice how much slower a Stochastic epoch is compared to the other methods. Basically, the bigger the batch size, the quicker the epoch, but the less progress is made with one epoch. This is an effect of vectorized computations made with NumPy [9] evaluating more training examples at each iteration of MBGD and VGD, compared to SGD, where for each epoch we must go through a for-loop inspecting one sample per iteration.

To analyze the implemented models we computed the R^2 score [10] to compare the true traffic volume values with the predicted ones. The results are shown in Table I.

Method	Best Loss on Validation Set	R^2 score
VGD* (*v1*)	0.418970	0.150064
MBGD* (*v1*)	0.417261	0.153531
SGD* (*v1*)	0.416476	0.155123
Normal Equation (*v1*)	0.417103	0.153851
sklearn SGDRegressor* (*v1*)	0.417342	0.153366
VGD* (*v2*)	0.174052	0.646913
MBGD* (*v2*)	0.171827	0.651426
SGD* (*v2*)	0.171668	0.651749
Normal Equation (*v2*)	0.171829	0.651420
sklearn SGDRegressor* (*v2*)	0.171713	0.651657

Table I
R^2 SCORE (*$\alpha = 0.0005$)

As both our models had relatively few features, no matrix inversion problems were experience when using the Normal Equation analytical solution to minimize the parameters θ for the training set. When using them to predict the validation values the results were better than Vanilla GD ones.

One interesting thing is that the computation time for the Normal Equation was less than a second, compared to around 15s for the GD variations, which makes it the best option for the amount of data that we are working with.

Finally, since our best model was *v2* with Stochastic Gradient Descent, we used its parameters to predict the traffic volumes of the test set to conclude the project. The value of the loss (cost) function $J(\theta)$ for it resulted in 0.17085, with an R^2 score of 0.6576, both being slightly better than the results on the validation set.

VII. Conclusions and Future Work

Our best results were reached with Stochastic Gradient Descent when considering 21 features for prediction. The outcome of the test set matched the validation results, showing resilience and applicability to never seen data.

We concluded that for simple models with relatively small datasets, the Normal Equation is the probably the best method, as it computes θ a lot faster than the iterative Gradient Descent algorithms, and also does not request you to define as many hyperparameters.

However, we can see how the Gradient Descent is more robust to bigger and more complex datasets, as well as being able to deal with cases that would be impossible for the Normal Equation, such as when $(\mathbb{X}^T\mathbb{X})$ is noninvertible. We believe that the Mini-batch GD is one of the best options in that case, as it's not as sensitive to high learning rates as the Stochastic GD, and also not as slow to converge as the Vanilla GD.

Although the achieved results are not bad, they aren't great either, seeing that Figures 1 and 2 show none of our models overfit the data, suggesting that a more complex model could be elaborated, leading to better predictions of the traffic volume. Thus, working more on feature extraction could have led to better results, specially based on the improvements we had from it on *v2*.

Lastly, we note two ways of improving the current models: *i)* polynomial regression, which would lead to a more complex hypothesis function $h_\theta(x)$, and *ii)* locally weighted linear regression, where the prediction error of each sample is differently weighted.

The notebook with the source code and the datasets used can be found on github.com/laurelkeys/machine-learning (inside the `assignment-1` folder).

References

[1] "Least squares — Wikipedia, The Free Encyclopedia," accessed September 2019. [Online]. Available: https://en.wikipedia.org/wiki/Least_squares 1

[2] UC Irvine Machine Learning Repository, "Metro Interstate Traffic Volume Data Set," accessed September 2019. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Metro+Interstate+Traffic+Volume 1, 2

[3] Andrew Ng, "CS229 Lecture notes," accessed September 2019. [Online]. Available: http://cs229.stanford.edu/notes/cs229-notes1.pdf 1

[4] Sandra Avila, "Regressão Linear — Aula 3," accessed September 2019. [Online]. Available: http://www.ic.unicamp.br/~sandra/pdf/class/2019-2/mc886/2019-08-19-MC886-Linear-Regression.pdf 1

[5] "Dummy variable trap in regression models," accessed September 2019. [Online]. Available: https://www.algosome.com/articles/dummy-variable-trap-regression.html 2

[6] "Standard score — Wikipedia, The Free Encyclopedia," accessed September 2019. [Online]. Available: https://en.wikipedia.org/wiki/Standard_score 2

[7] Andrich van Wyk, "Encoding Cyclical Features for Deep Learning," accessed September 2019. [Online]. Available: https://www.kaggle.com/avanwyk/encoding-cyclical-features-for-deep-learning 2

[8] "sklearn.linear_model.SGDRegressor," accessed September 2019. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html 2

[9] "NumPy," accessed September 2019. [Online]. Available: http://www.numpy.org/ 4

[10] "Coefficient of determination — Wikipedia, The Free Encyclopedia," accessed September 2019. [Online]. Available: https://en.wikipedia.org/wiki/Coefficient_of_determination 4