Recursive Least Squares

Carl Richardson

19th February 2021

Contents

1 Introduction

This report compares the recursive least square's implementation of the linear regression estimation against closed form and gradient descent implementations. The linear model, shown by equation 1, is referenced throughout the report; y represents the target vector, X represents the data matrix containing the input vectors, w represents the weights to be estimated and ϵ represents the corrupting noise vector.

$$y = Xw + \epsilon \tag{1}$$

2 Synthetic Problem

The synthetic data was generated using the default random number generator from *numpy*. To allow easy comparison, the random number generator was seeded to ensure the same data was used by each algorithm. The synthetic problem was to estimate the weights of a linear model, where the target vector was corrupted by random noise. The input to the linear system was a 10,000 by 8 data matrix. This was comprised of 10,000 input vectors, each made up of 8 features. The elements of this matrix were sampled from a uniform distribution on the interval [-10, +10]. The true 8-element weight vector was formed by sampling from a uniform distribution on the interval [-5, +5]. Finally, the target vector was generated using the linear model; the noise vector was sampled from a zero-mean, unity variance, normal distribution.

3 Linear Regression Estimation

In this section, given the input data matrix and the corrupted target vector, the weights of the model were estimated using linear regression. Four different implementations were considered, each have their own benefits in terms of computation time and accuracy.

The closed form solution, shown by equation 2, was found by minimising the sum of square errors, shown by equation 3. This implementation finds the optimal solution, but it is computationally expensive for models with many weights, due to the costly matrix inversion.

$$w_{CF} = (X^T X)^{-1} X^T y \tag{2}$$

$$E = (y - Xw)^T (y - Xw) \tag{3}$$

Gradient descent (GD) aims to iteratively compute an estimate of the weights using equation 4. During each iteration, the gradient was computed according to equation 5, using all the available data. Then the weights were updated. Stochastic gradient descent (SGD) and mini batch gradient descent (MBGD) were also investigated. Both use equations 4 and 5 for computing the gradient and updating the weights; however, the input data and targets used to compute the gradient varied throughout the recursion. SGD randomly selects one data point for computing the gradient on each iteration, whilst MBGD cycles through the data set in subsets of a chosen size. In this experiment, the 10,000 data points were split into 500 batches, each contained 20 data points. The convergence of these implementations was dependent on the choice of the learning rate. For this problem it was set to 0.001.

$$w^{k+1} = w^k - \eta \nabla_w E \tag{4}$$

$$\nabla_w E = -2X^T (y - Xw) \tag{5}$$

Figure 1 shows an approximate linear relationship between the true weights and the corresponding estimations from each implementation. Each implementation appears to result in very similar estimates. Figure 1 also shows that the relationship between the true target and the predicted target has a linear trend, but has more error than that of the weights. This was because the estimated true targets were corrupted by noise whereas the estimated model had captured the underlying behaviour well. Table 1 shows, for each implementation, the final uncorrupted mean square error (MSE) was much lower than that of its corrupted counterpart, which confirms the underlying model was well estimated. It also shows that SGD performed significantly worse than the other implementations, but this was to be expected.

Figure 2 shows how the MSE evolved over the first 200 iterations for each GD implementation. Interestingly, SGD appears to converge slightly quicker; however, this was due to the scale of the y-axis. In fact, the MSE for the SGD implementation never converged because the gradient did not stabilise. The randomly selected data point used to compute it, at each iteration, was the reason for this. In the final 50 iterations, the MSE varied between 1.5 and 2.3. This was the trade-off for a more computationally efficient scheme compared to the closed form and GD implementations. MBGD provides a parameterised medium. As the batch size increases (number of batches decreases), the performance tends to that of GD. This means improved convergence, but greater computational requirements. Conversely, when the batch size decreases, the performance tends to that of SGD. When the mini batch size of 20 was used, the MSE varied between 1 and 1.1 during the final 50 iterations. The range reduced by a factor of 8 highlighting the improved consistency.

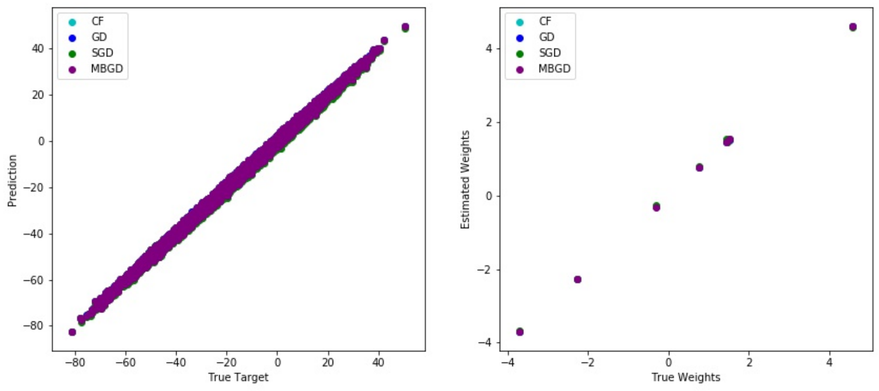

Figure 1: True targets and true weights versus their linear regression estimations for four different implementations

Implementation	MSE between corrupted output and predicted output	MSE between uncorrupted output and predicted output
Closed Form	1.003	0.0007
Gradient Descent	1.004	0.0009
Stochastic Gradient Descent	2.085	0.3361
Mini Batch Gradient Descent	1.005	0.0082

Table 1: MSE using corrupted and uncorrupted targets

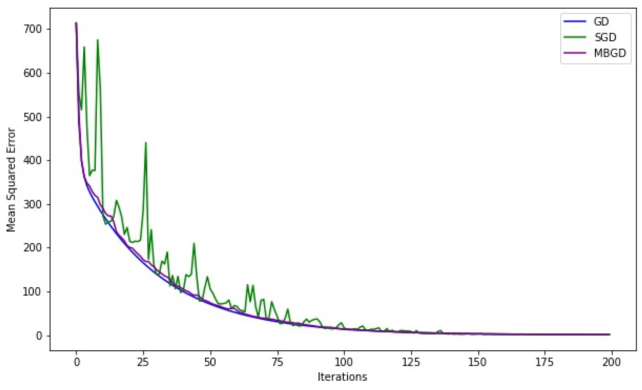

Figure 2: Evolution of the MSE for GD, SGD and MBGD implementations

4 Recursive Least Squares (RLS)

Like SGD, RLS is a method suitable for online learning. Both can update the weights and other variables quickly enough for real time applications. The update equations for the RLS implementation are outlined in the assignment introduction. RLS appends data to the data matrix and target vector as it arrives. It updates the weights using the full dataset available; however, it uses the matrix inversion lemma for more efficient computation. Another benefit over SGD is that it does not require the tuning of a learning rate.

Figure 3 shows how the MSE evolved over the first 200 iterations of the SGD and RLS implementations. The curve of the RLS implementation is much smoother since the update rule builds on all the available data rather than using a random data point. The final MSE of the RLS implementation was 1.571. This was a 25% reduction of the MSE when SGD was used. This indicates the RLS implementation predicts the corrupted targets more accurately. When the MSE was computed using the uncorrupted target, it was 0.6602. This was an increase of 96% with respect to that of SGD. This suggests SGD learnt the true model quicker than RLS and RLS was more influenced by the noise of the targets.

Figure 4 shows the number of iterations required for the RLS implementation as a function of the number of weights of the linear model. The number of weights varied from 10 to 20. A convergence threshold of 0.1 was set. To implement this procedure, a random data set was generated for each iteration. This was repeated 100 times for each number of weights to obtain an average. Each additional weight to be learnt increased the dimensionality of the problem; therefore, the figure indicates a positive linear trend between the increase in dimensionality and the speed of convergence. This relationship seems sensible since the regression now needs to fit the data in an additional dimension.

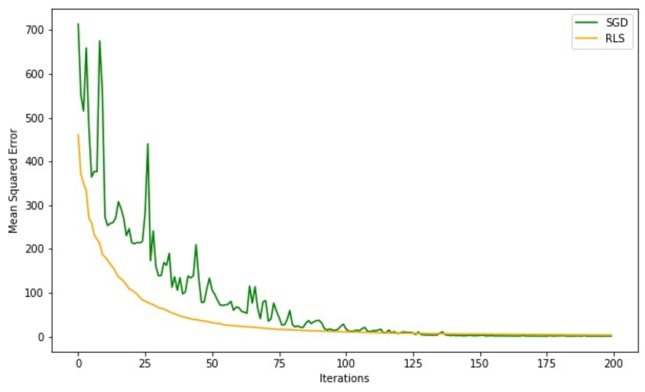

Figure 3: Evolution of the MSE for SGD and RLS implementations

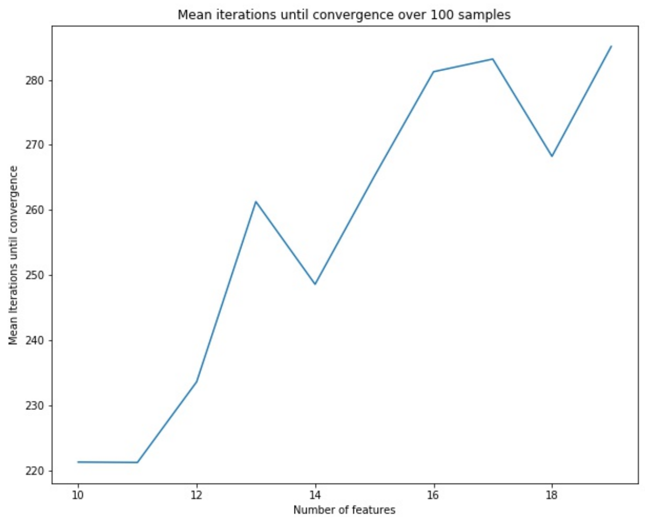

Figure 4: Mean number of iterations until convergence as a function of the number of features

5 Parkinson's Dataset

A dataset containing voice measurements from 42 people (5,875 voice recordings) with early-stage Parkinson's disease was taken from [**?**]. Each voice recording contained several measurements about the jitter (variation in fundamental frequency), shimmer (variation in amplitude), NHR (ratio of noise to tonal components), RPDE (nonlinear dynamical complexity measure), DFA (Signal fractal scaling exponent) and PPE (nonlinear measure of fundamental frequency variation). In total 16 input features were used. The objective of the regression task was to find a model to predict the motor UPDRS (Unified Parkinson's Disease Rating Scale) scores.

SGD and RLS implementations were used to estimate the weights of the linear model. In both cases, the weights were initialised at zero and in the case of RLS, the inverse of the data matrix was computed for 10 data points before being updated following the matrix inversion lemma. The initial 500 iterations are shown in figure 5. In this case, the RLS implementation decreased rapidly in the first 50 iterations, then plateaued to a similar rate as the SGD implementation. The final MSE after the full 3000 iterations was 93.74 and 54.34 for the SGD and RLS implementations respectively. The curves demonstrate that the implementations were able to iteratively reduce the MSE; however, the final values suggest the linear model was unable to capture the relationship between the inputs and outputs. To improve on this, two non-linear basis functions were considered. The first a polynomial basis and the second a sinusoidal basis. Lower order polynomials performed worse and higher order polynomials led to overflow issues, suggesting the values of the transformed matrix were too large. Hence, a sinusoidal basis was chosen to avoid the overflow issue. However, this then led to issues with trying to perform matrix inversion and matrix multiplication. Further investigation would be required to resolve these issues but finding a suitable basis function seems like a good approach for improving performance. It was no surprise a simple linear function of the data was not successful in mapping the input to the target in this complex problem.

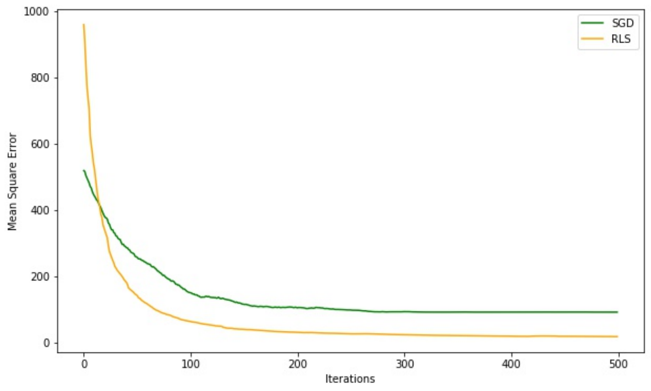

Figure 5: Evolution of the MSE for SGD and RLS implementations

6 Statsmodels

To verify my implementation of the RLS algorithm, the *statsmodels* implementation was used on the synthetic problem and the results were compared. Figure 6 shows the evolution of the weights as the weights were iteratively updated using both implementations. The figure shows that although the weights were initialised at different states, they both moved towards the same stable values. As expected, the final MSE for the *statsmodels* implementation was very similar to that of my implementation, it was 1.140. Furthermore, figure 7 compares the true weights with the estimated weights and the true targets with their predictions using both RLS implementations. As can be seen, they are both accurate and almost identical. These results indicate the prior results on the Parkinson's data set can be trusted.

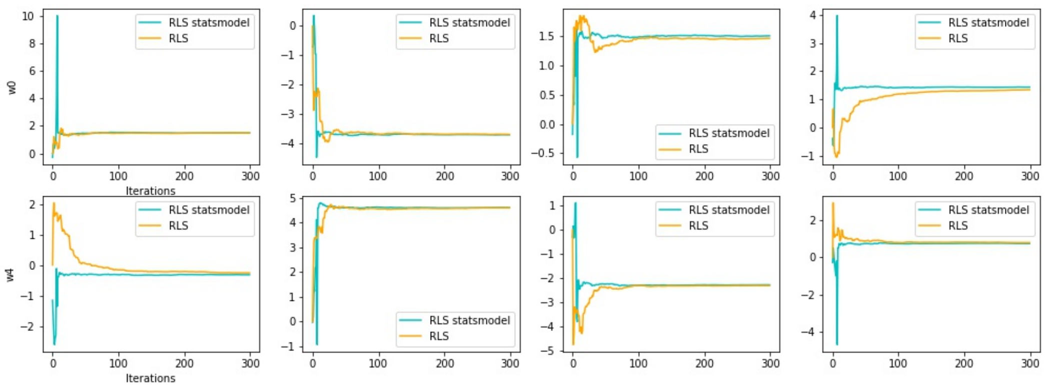

Figure 6: Evolution of the estimated weights using my own and *statsmodels* implementation of RLS

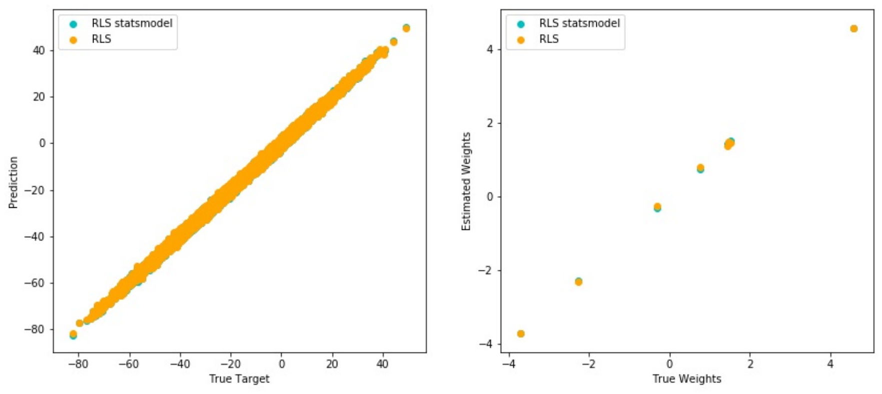

Figure 7: True targets and true weights versus their linear regression estimations for my own and *statsmodels* implementation of RLS