FYS-STK4155 - Project 2

Kari Lovise Lodsby

November 15, 2021

Abstract

In this project, I have looked at applications of neural networks in linear regression problems and classification problems, and compared them to other methods. For the linear regression, I tried to fit a model to the Franke function. For the classification case, I looked at the Breast Cancer Wisconsin Data Set and tried to train a network to classify tumors as benign or malignant based on their characteristics. I also tried to compare neural networks to other methods.

For linear regression, I found that the easiest SGD method (the one based on OLS regression) tends to have worse precision and efficiency than methods that require matrix inversion. While basing the SGD on Ridge regression instead considerably improved its accuracy, it also slowed it down further.

For the classification problem, I found logistic regression easier to use and faster than neural networks.

1 Introduction

Approximations for complex problems often require complex models to be accurate, but we also have to make sure they do not become so complex that they end up overfitting. As neural networks provide a non-linear way to approximate data, they could prove useful in accurately modelling complex data sets. In this report, I compare neural networks to linear regression methods for function evaluation and logistic regression for classification.

My tests of linear regression involve fitting the Franke function with various methods. First I compare an Ordinary Least Squares approach to Stochastic Gradient Descent methods based on both Ordinary Least Squares and Ridge regression methods. Then I compare them to a neural network. I include some results from testing and optimizing parameters for these methods.

Then, for classification problems, I compare a logistic regression method to a neural network. The data set will be the Breast Cancer Wisconsin Data Set, which contains information about tumors and whether they were benign or malignant. I will attempt to classify tumors as benign or malignant based on their characteristics.

The metrics I will take a look at are the computational time it takes to train the models, and the accuracy of the models.

2 Theory

2.1 Linear analysis - stochastic gradient descent

Many methods of linear regression lead to problems that require inverting a matrix. Two common examples of this are Ordinary Least Squares (OLS) and Ridge regression.

$$\hat{\beta}_{OLS} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T y$$

$$\hat{\beta}_{Ridge} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T y$$

In both formulas, $\mathbf{X}$ is the design matrix, y is the true value for the training data, and $\hat{\beta}$ are the optimal parameter values. Additionally, in Ridge regression, $l\lambda$ is a chosen hyperparameter. It is easy to see that both of these methods require a matrix to be inverted.

However, the matrix $\mathbf{X}^T\mathbf{X}$ is often close to being singular. As a result, the matrix inversion is likely to lead to large numerical errors, and trying to solve the equivalent linear system of equations has the same problem.

We try to avoid this calculation by going back to the step where these relationships were derived. That was the step where we wanted to minimize the cost functions:

$$C(\beta)_{OLS} = \frac{1}{n}(\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta)$$

$$C(\beta)_{Ridge} = \frac{1}{n}(\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta) + \lambda\beta^T\beta$$

To minimize these costs, we calculated their gradients and set them to 0. This yields the $\hat{\beta}$ that minimizes them. To avoid the matrix inversion, we will instead try to find the gradient numerically.

The first idea is to sum the calculation over all data points. This yields an accurate result, but is computationally expensive if the data set is large. If there are many parameters β, that will also contribute to the calculation being heavy.

$$\nabla_\beta C(\beta)_{GD} = \sum_{i=0}^{n-1} \nabla_\beta c_i(\mathbf{X}_i, \beta)$$

To prevent this problem, we perform the calculation for a randomly selected subset B_k of the data instead:

$$\nabla_\beta C(\beta)_{SGD} = \sum_{i \in B_k} \nabla_\beta c_i(\mathbf{X}_i, \beta)$$

The updated parameters use the formula

$$\beta^{(i+1)} = \beta^{(i)} - \gamma \cdot \nabla_\beta C(\beta)$$

The parameter γ is the learning rate. The method converges if this parameter is small enough. If it is too large, β will diverge. If it is constant, the solution will converge, but only to a point. To make a method that converges, we design it so that γ decreases after each run:

$$\gamma_j(t; t_0, t_1) = \frac{t_0}{t + t_1}$$

t_0 and t_1 are positive, freely chosen parameters. $t = e \cdot m + j$ increases for each run: e is the current epoch, m is the current mini-batch and j is the iteration number when iterating over the mini-batches.

2.2 Neural networks

Another parametrization method is neural networks, a type of system inspired by biology and neuroscience. A neural network consists of nodes connected by links. Usually, there are three types of layers of nodes: one for input, one for output, and hidden layers.

Individual nodes calculate an output based on the inputs received from the nodes of the previous layer (x_i for the i-th of N nodes), the weights (w_i) corresponding to these nodes, and the bias (b_j) for the node performing the calculation. The weights and biases can be chosen freely for each node. The activation function used is on the following form:

$$f(u) = f(\sum_{i=1}^{N} x_i w_i + b_j)$$

Some choices of f include:

- $f(u) = \sigma(u)$ (the sigmoid function)
- $f(u) = \tanh(u)$ (the hyperbolic tangent function)
- $f(u) = \max(0, u)$ (the ReLU function)

- $f(u) = \max(0.01u, u)$ (the leaky ReLU function)

The node then sends its output to the nodes in the next layer. As a result, the network as a whole creates a non-linear relation between the set of input values and output values.

We wish to train this network using the back-propagation algorithm, which adjusts the weights and biases to adapt them to a desired model. In order to do that, we need choose a cost function C (often the mean-squared error), which will be used to find the input error. We calculate it based on the function value of the derivative of the activation function ($f'(z_j^N)$) and the gradient of the cost function ($\frac{\partial C}{\partial(a_j^N)}$). N reflects that this is the output layer - the N-th layer - and j is the j-th node of the layer.

$$\delta_j^N = f'(z_j^N)\frac{\partial C}{\partial(a_j^N)}$$

For a hidden layer ($n = N-1, N-2, \ldots, 2$), the back-propagation error is

$$\delta_j^n = \sum_k \delta_k^{n+1} w_{kj}^{n+1} f'(z_j^n)$$

For the j-th node, we sum the errors from the nodes from the $l+1$-th layer multiplied by the corresponding weights and the derivative of f. This leads to an iterative scheme where every δ_j^n can be found using only the output layer.

These values are then used to update the weights and biases: we set the new w_{jk}^n equal to $w_{jk}^n - \eta\delta_j^n a_k^{n-1}$, and the new b_j^n equal to $b_j^n - \eta\delta_j^n$. η is the learning rate, here a constant. It is chosen to fit the problem.

For a linear regression, the output layer will only have a single node, corresponding to the function value. The input layer consists of the input to the function,

2.3 Classification problems - logistic regression

In logistic regression, we wish to adapt a function to the following sigmoid function:

$$\sigma(t) = \frac{e^t}{e^t + 1}$$

We define t as a linear combination of independent variables, where the x_i are defined from the problem and the σ_i are free parameters that we wish to adjust.

$$t = \sigma_0 + \sigma_1 x_1 + \cdots + \sigma_n x_n$$

As the sigmoid function has outputs in the range $[0, 1]$, it is useful for binary problems. However, if we have a problem with more than two possible outcomes, we instead use the softmax function Σ, a generalization of the sigmoid function that uses a sum over all the classes K, $\Sigma(t_i)$ is the probability that class i is the correct one, and $t_i = \theta_{i,0} + \theta_{i,1}x_1 + \theta_{i,2}x_2 + \cdots + \theta_{i,n}x_n$:

$$\Sigma(t_i) = \frac{e^{t_i}}{\sum_{j=1}^k e^{t_j}}$$

However, since we will look at a binary classification problem, we will not discuss the softmax function further.

For a binary problem, the cost function is

$$C(\theta) = -\ln P(D|\theta) = -\sum_{i=1}^n y_i \ln[P(y_i = 0)] + (1 - y_i)\ln[1 - P(y_i = 0)] = \sum_{i=1}^n L_i(\theta)$$

$$C(\theta) = -\sum_{i=1}^n y_i \ln[\sigma(t)] + (1 - y_i)\ln[1 - \sigma(t)]$$

The elements of the gradient are

$$\frac{\partial}{\partial \theta_j} C(\theta) = \sum_{i=1}^{n} (\sigma(t) - y_i) x_j$$

This will be used as the gradient in the standard linear regression.

The data set we will look at is the UC Irvine Machine Learning Repository Breast Cancer Wisconsin (Diagnostic) Data Set, available for download via the SKLearn. It contains 569 rows, each corresponding to a patient and their tumor. The first two of the 32 columns correspond to the patient ID and the label of their tumor. The 30 remaining rows come from continuous features from the raw input data. They correspond to 10 measurable features and 3 categories of them: mean value, standard deviation and worst case / largest value. In the design matrix, we make a column for each category, starting from the mean value.

We associate 1 with a malignant tumor and 0 with a benign tumor.

2.4 Classification problems - neural network

For the classification problem, we need to make some adjustments to the neural network.

The input layer will have $n \cdot m$ nodes, one for each pixel in the picture. The output layer has two nodes, one for each class.

The hidden nodes and output function will use the sigmoid function σ, which outputs values between 0 and 1. The sum over both classes should normalize to 1.

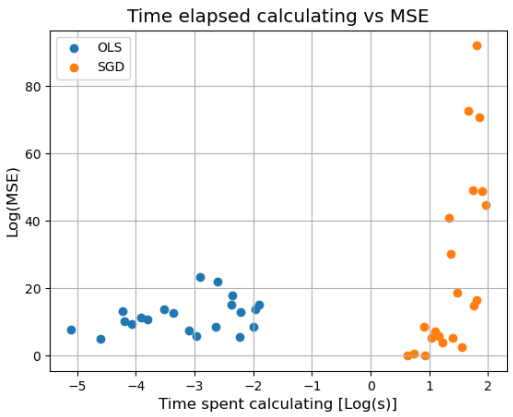
We choose the accuracy score $\frac{1}{n} \sum_{i=0}^{n-1} I(t_i = y_i)$, where I is the indicator function, t_i are the predicted outcomes and y_i are the actual outcomes.

3 Results and discussion

3.1 Linear regression

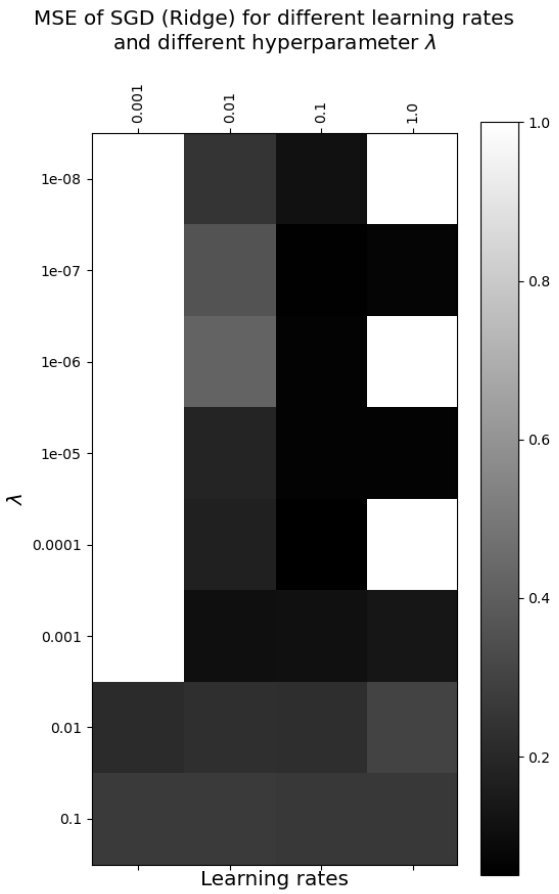
First I performed an experiment with the Franke function and just my stochastic gradient descent and the SKLearn implementation to confirm that mine works well.

Then I tested the OLS variation of the SGD method with OLS itself.

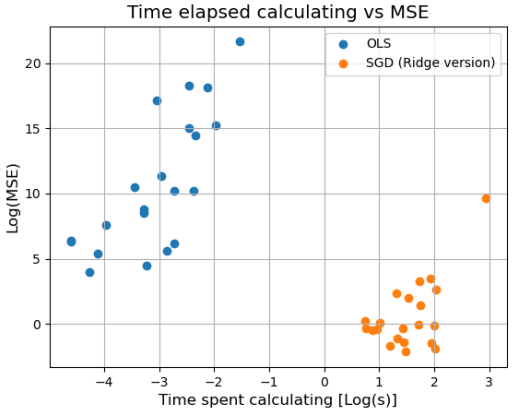


We can see that SGD is considerably worse in both ways.

Then I tested the Ridge variation of the SGD method. First I looked at how the choice of hyperparameter and learning rate affected the method:

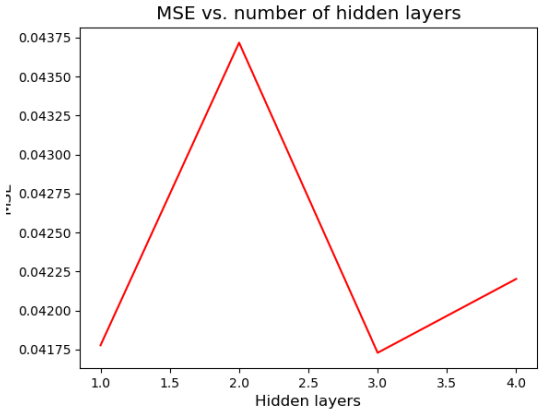


We can see that the choice of learning rate matters more than the hyperparameter. Based on this, I choose $\lambda = 0.0001$ and a learning rate of 0.001. This yielded the following result:

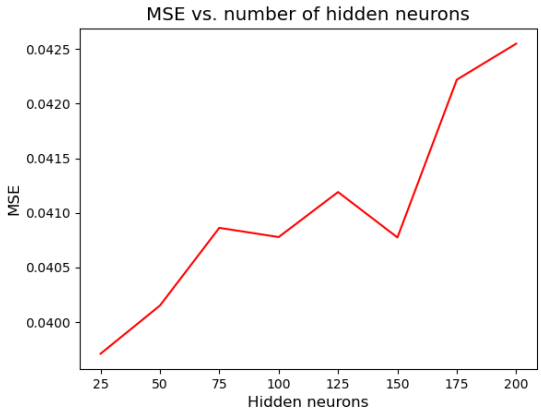


This time, the accuracy of the SGD is better than for OLS. This calculation itself is not much slower than the previous SGD method, but the optimization of the parameters slows down the SGD method considerably, as it essentially has to redo the SGD calculation many times. As a result, OLS still outperforms it.

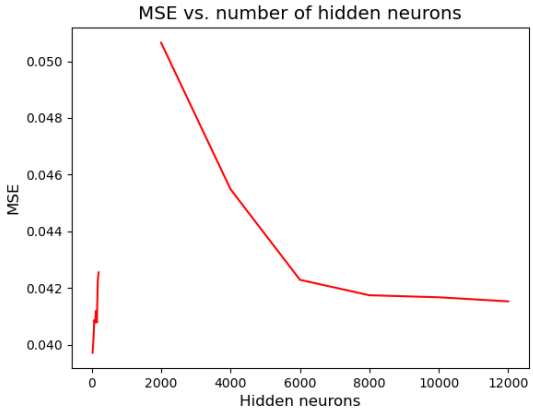
Then I used the neural network and looked at how the number of layers, neurons and epochs:



It seems like three hidden layers is the best number, as four can lead to overfitting.
Then I tried experimenting with the number of hidden neurons:



The MSE seems to increase when the number of hidden neurons increases, which suggests overfitting. I chose to stick to a relatively high number of neurons, as I want to avoid underfitting. 100 seemed like a good compromise.

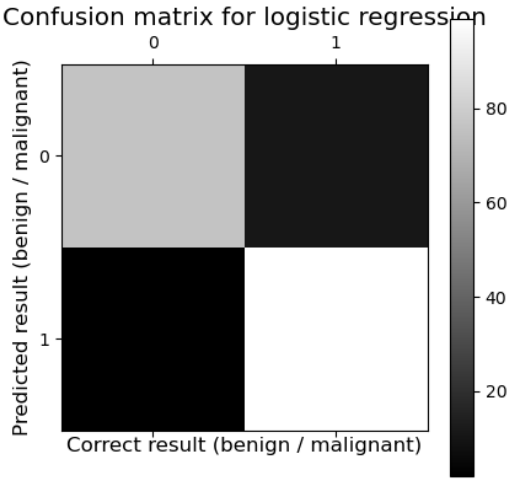


We see that iterating over more epochs makes the model more accurate, but at some point we hit dimishing returns. I chose to go with 10,000 epochs here.

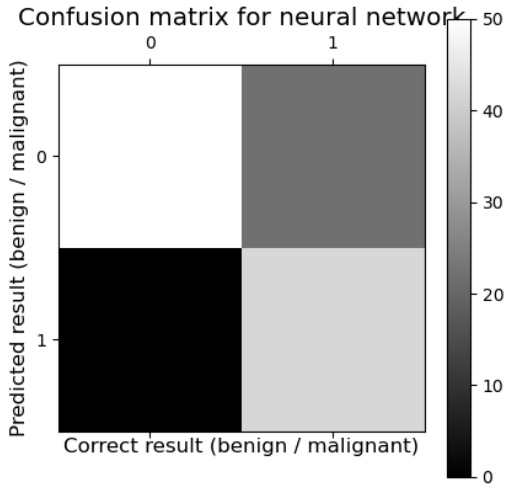
I found that neural networks outperform SGD methods, as the OLS-SGD suffers from low accuracy (especially for its runtime), and the Ridge-SGD requires a lot of parameter optimization to be good. However, the plain OLS still appears to be largely better than neural networks.

3.2 Logistic regression

First I used logistic regression on the breast cancer data set. I was able to get a program that worked pretty quickly and was fairly accurate:



Then I used a neural network. It ended up being slower because it needed a lot of time to find a good learning rate and hyperparameter, but the best result was more accurate than the one from my logistic regression function.



4 Conclusion

In this report, I have examined linear regression methods in the form of a matrix inversion method, gradient approaches and neural networks. I also looked at classification problems, where I compared logistic regression to neural networks.

It looks like the easiest form of SGD is both slower and less accurate than its matrix inversion counterpart. While basing the SGD on Ridge regression makes it considerably more accurate, it is

further slowed down by the search for good learning rates and hyperparameters. In total, matrix inversion methods seem to be preferable despite their potential for numerical problems.

Neural networks appear to be about as accurate as matrix inversion methods, but they require a lot of time to train because they have so many parameters to optimize for. As a result, they may not be able to outperform OLS. However, in this case, they did come close.

For the tumor classification problems, I found that logistic regression came close to the accuracy to neural networks, while being faster to train as there were fewer design choices to optimize.

One important takeaway from this project was that the choices of parameters play a big role for the success of a method. The most striking example from my experiments was that poor parameter choices for the classification problems could lead to useless algorithms that put every input in the same category.