Building Linear Regression and Artificial Neural Network Models In Python

Tagliatela college of Engineering

Name	: Muvva Mukesh
Guidance	: Prof. Minkyu Kim, PhD.
Course	:Math for Data Scientists

Introduction

This report consists of how to implement our own Linear Regression Model and Neural Networks Model from scratch in python. This helps many of us to construct our own models according to our requirement. These models would not use any powerful machine learning algorithms like scikit-learn. Building these models from scratch would give you an in depth idea of how these models work. These models were constructed in python3 using only NumPy and Pandas libraries. NumPy is used to leverage the use of its advanced mathematical calculations and to reduce the complexity of code and time. Pandas is used to load the inputs and outputs into our models. Pandas is also used for data wrangling and data manipulation easily and quicker.

This report will walk through you the concepts of Linear Regression and Artificial Neural Networks and how to build them. For each model the implementations will be explained. A brief Data Analysis and model results will be explained.

Linear Regression

Theory

Linear Regression tries to establish a relationship between two variables by fitting a linear equation to observed data. One variable is considered to be an explanatory variable, and the other is considered to be a dependent variable. For example, a modeler might want to relate the weights of individuals to their heights using a linear regression model.

Before attempting to fit a linear model to observed data, a modeler should first determine whether there is a relationship between the variables of interest. This does not necessarily imply that one variable *causes* the other but that there is some significant association between the two variables. A scatterplot can be a helpful tool in determining the strength of the relationship between two variables. If there appears to be no association between the proposed explanatory and dependent variables (i.e., the scatterplot does not indicate any increasing or decreasing trends), then fitting a linear regression model to the data probably will not provide a useful model. A valuable numerical measure of association between two variables is the correlation coefficient, which is a value between -1 and 1 indicating the strength of the association of the observed data for the two variables.

A linear regression line has an equation of the form **Y = a + bX**, where **X** is the explanatory variable and **Y** is the dependent variable. The slope of the line is **b**, and **a** is the intercept (the value of ***y*** when ***x*** = 0).

The model I have chosen is Multivariate Linear Regression i.e input has more than two features

Y=a0*X1+a1*X2+a2*X3+a3*X4+a4*X5+a5*X6+b.

Implementation

1) Input file has 5 features and output file has 3 different outputs
2) Building 3 Linear Regression Models for each of these output
3) Load the input using Pandas (easy tool for data manipulation and preprocessing)
4) Create scatterplots between input features and output to check whether we can apply linear regression or not
5) After finding a correlation between input and output, split the data set into two parts
    - Train Data
    - Test Data
6) Build a Linear Regression Class with initializing all the variables which are necessary for regression. Making input coefficients and intercept global has got advantages and later can be used for prediction of new input.
7) Implement cost function, optimization function (gradient descent), fit method and predict method.
8) After initializing the linear regression class try to fit the trained data
    - The fit method will call the gradient descent method
    - The gradient descent calculates the estimated output using the initialization parameters.
    - The gradient descent calculates the loss and derivates to reduce the cost.
    - If the cost is less than the threshold value what we have set for while initializing, it updates all the newly calculated coefficients.
    - This process goes on until we gets the threshold value or for fixed range of iterations. This process gives us the best fit model.
9) After model fitting, we can predict for new input values (test data)

MODEL RESULTS

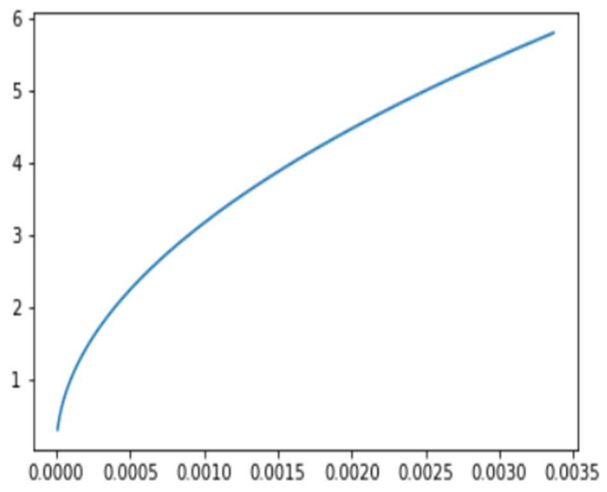
As there are three outputs, three different linear regression models were created. For each model the least cost value was taken to determine the coefficients and intercepts for the model.

	A0	A1	A2	A3	A4	b
Model-1	0.6111	-0.2248	-0.180	0.2498	0.3475	3.4620
Model-2	0.7937	-0.366	4.19	0.1108	1.595	2.623
Model-3	2.824	-7.019	1.5701	2.1862	-1.0499	27.1535

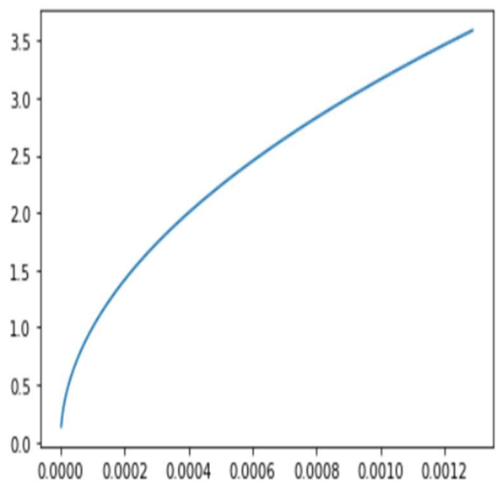
The above table represents the three best fit models according to the three outputs.

	Initial Cost	Final Cost
Model-1	6.536	0.23947
Model-2	2.9567	0.215795
Model-3	752.004	69.9168

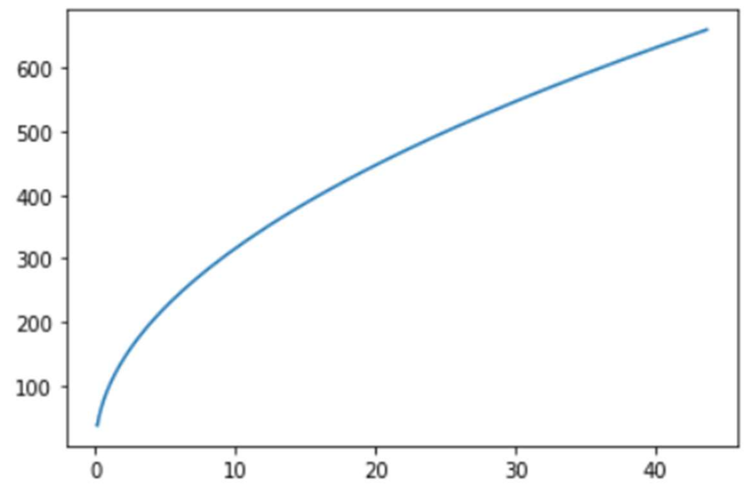
The above table represents the Initial costs and final costs for the models



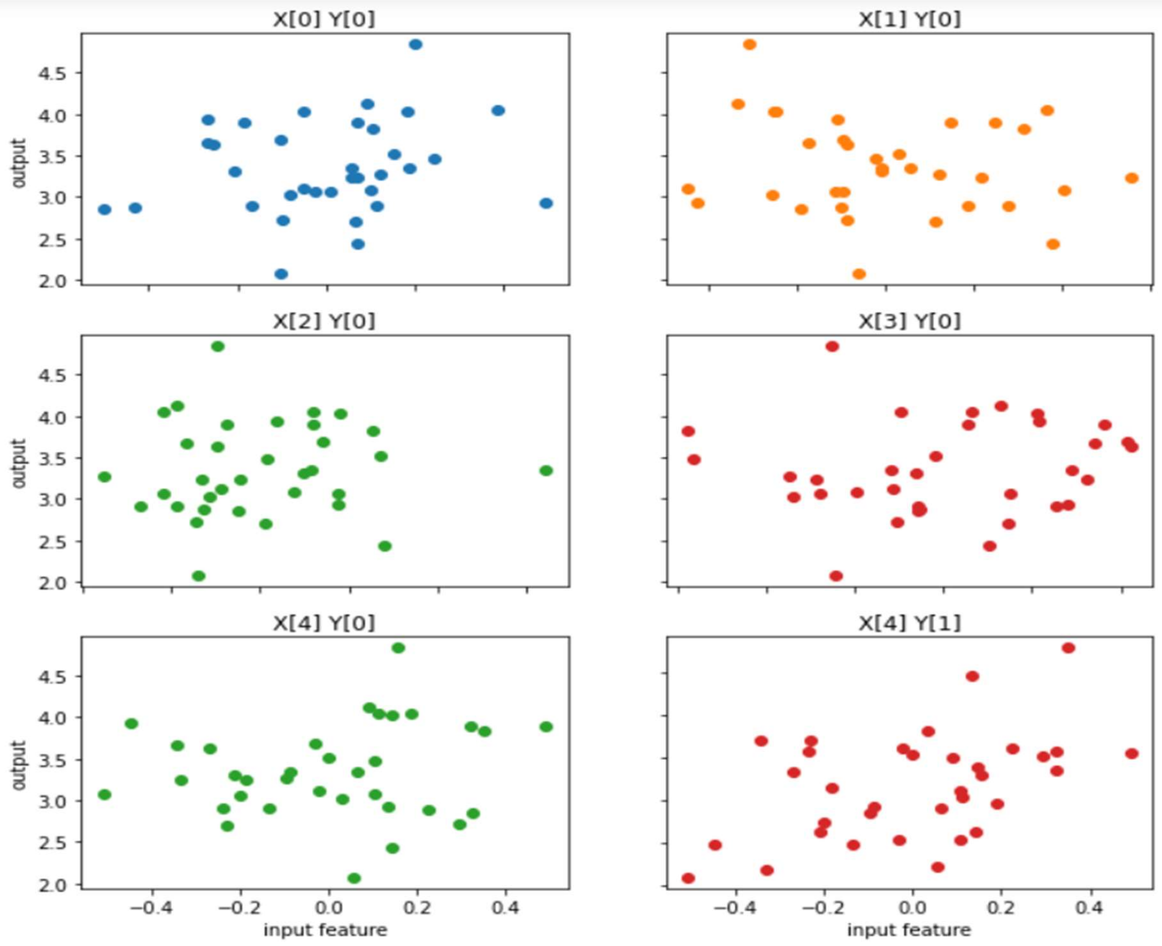
Bias vs Variance Model-1



Bias vs Variance Model-2



Bias vs Variance Model-3



Scatter Plots of the input features and outputs

Analysis

The input data has 6 features and they all were positively and negatively correlated with the outputs. As there is correlation between input and output, So I am able to apply Linear Regression for the data.

The input data has at least 200 Null columns without having any data. In order to use NumPy underlying powerful mathematical methods we have to filter out all the Null columns.

Using NumPy methods to calculate coefficients is the best way, because NumPy internally vectorizes all the inputs and process the data faster.

All the three regression models used all the 10,000 iterations to reach the best fit model. The reason behind is my threshold value is much lesser than the cost. My cost would never able to approach the threshold value despite after trying different iteration rate and No of epochs.

All the three models final cost is very much lesser than the initial cost they have started.

Using gradient descent method helped the models to determine the best fit line.

For all the three models there is very low bias and low variance which made these models can generalize new inputs.

No overfitting found

No underfitting found

Model-1 and Model-II have nearly equally coefficient values and near cost values, the applications may change according to the models used

Model-III has the highest cost when compared to other two models.

The model can be used to other data by changing coefficients according to the features available in the new data set.

Artificial Neural Network

Theory

Artificial neural networks are inspired by the organic brain, translated to the computer. It is not a perfect comparison, but there are neurons, activations, and lots of interconnectivity, even if the underlying processes are quite different. A single neuron by itself is relatively useless, but, when combined with hundreds or thousands (or many more) of other neurons, the interconnectivity produces relationships and results that frequently outperform any other machine learning methods.

Neural networks are black boxes in that we often have no idea why they reach the conclusions they do. Dense layers, the most common layers, consist of interconnected neurons. In a dense layer, each neuron of a given layer is connected to every neuron of the next layer, which means that its output value becomes an input for the next neurons. Each connection between neurons has a weight associated with it, which is a trainable factor of how much of this input to use, and this weight gets multiplied by the input value. Once all the inputs weights flow into our neuron they are summed, and a bias, another trainable parameter, is added. The purpose of the bias is to offset the output positively or negatively which can further help us map more real-world types of dynamic data.

In between Neural Network layers each layer uses Activation functions to decide when a neuron should fire. Activation functions control output and improves the accuracy of the network. There are many Activation functions, one has to choose a particular activation function based on the application they are using. A neural network consumes lot of memory for computation.

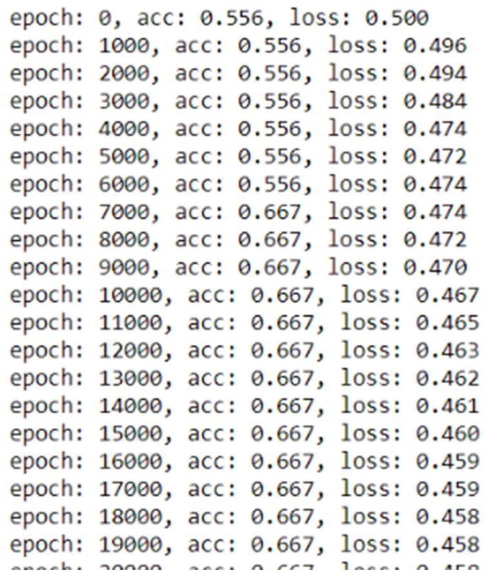
Implementation

1) Create a dense layer
    - Takes the size of input neurons and size of output neurons
    - Initializes weights according to difference between input and output neurons
    - Define forward Propagation.
    - In forward propagation it combines all the inputs, weights and biases and gives output of the layer
    - Define backward Propagation
    - This Dense Layer backward propagation takes the derivate outputs of activation function as feedback.
2) Create activation function
    - Define forward Propagation for the activation function
    - The forward propagation processes the input according to the activation function characteristics
    - Define backward Propagation to take back derivative feedback from the next dense layer
3) Create Loss Function
    - Create a loss function which gives the difference between actual and predicted values
4) Create an Optimizer (Gradient Descent)
    - Create an optimizer which updates all the weights across all the layers according to the derivative feedback inputs
5) Create an Predict function
    - Creating an predict function which takes all the dense layer objects, input and predicts the output
6) Create input and output neurons w.r.t other dense layers
7) Preprocess the data and initialize functions according to implementation

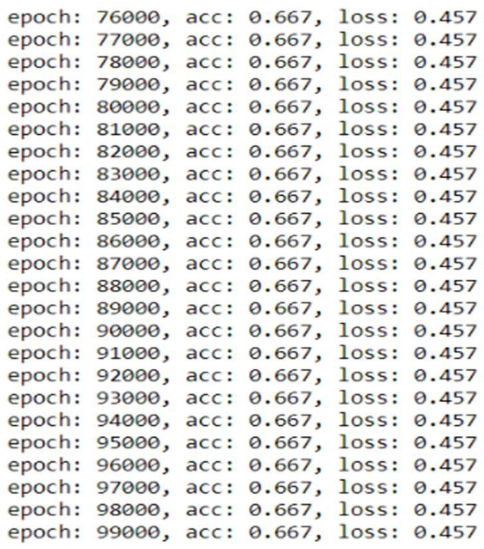
Network Description

#	Layer Type	Input neuron	Output neuron	Activation
1	Input	2	5	Sigmoid
2	Hidden	5	4	Sigmoid
3	Hidden	4	3	Sigmoid
4	Hidden	3	2	Sigmoid
5	Output	2	1	Sigmoid

Model Results

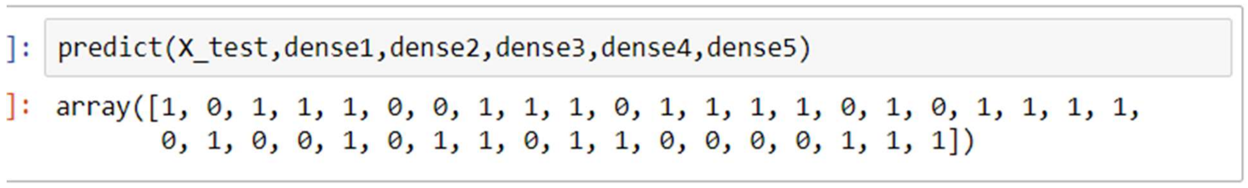


```
epoch: 0, acc: 0.556, loss: 0.500
epoch: 1000, acc: 0.556, loss: 0.496
epoch: 2000, acc: 0.556, loss: 0.494
epoch: 3000, acc: 0.556, loss: 0.484
epoch: 4000, acc: 0.556, loss: 0.474
epoch: 5000, acc: 0.556, loss: 0.472
epoch: 6000, acc: 0.556, loss: 0.474
epoch: 7000, acc: 0.667, loss: 0.474
epoch: 8000, acc: 0.667, loss: 0.472
epoch: 9000, acc: 0.667, loss: 0.470
epoch: 10000, acc: 0.667, loss: 0.467
epoch: 11000, acc: 0.667, loss: 0.465
epoch: 12000, acc: 0.667, loss: 0.463
epoch: 13000, acc: 0.667, loss: 0.462
epoch: 14000, acc: 0.667, loss: 0.461
epoch: 15000, acc: 0.667, loss: 0.460
epoch: 16000, acc: 0.667, loss: 0.459
epoch: 17000, acc: 0.667, loss: 0.459
epoch: 18000, acc: 0.667, loss: 0.458
epoch: 19000, acc: 0.667, loss: 0.458
```



```
epoch: 76000, acc: 0.667, loss: 0.457
epoch: 77000, acc: 0.667, loss: 0.457
epoch: 78000, acc: 0.667, loss: 0.457
epoch: 79000, acc: 0.667, loss: 0.457
epoch: 80000, acc: 0.667, loss: 0.457
epoch: 81000, acc: 0.667, loss: 0.457
epoch: 82000, acc: 0.667, loss: 0.457
epoch: 83000, acc: 0.667, loss: 0.457
epoch: 84000, acc: 0.667, loss: 0.457
epoch: 85000, acc: 0.667, loss: 0.457
epoch: 86000, acc: 0.667, loss: 0.457
epoch: 87000, acc: 0.667, loss: 0.457
epoch: 88000, acc: 0.667, loss: 0.457
epoch: 89000, acc: 0.667, loss: 0.457
epoch: 90000, acc: 0.667, loss: 0.457
epoch: 91000, acc: 0.667, loss: 0.457
epoch: 92000, acc: 0.667, loss: 0.457
epoch: 93000, acc: 0.667, loss: 0.457
epoch: 94000, acc: 0.667, loss: 0.457
epoch: 95000, acc: 0.667, loss: 0.457
epoch: 96000, acc: 0.667, loss: 0.457
epoch: 97000, acc: 0.667, loss: 0.457
epoch: 98000, acc: 0.667, loss: 0.457
epoch: 99000, acc: 0.667, loss: 0.457
```

Predictions:



```
]: predict(X_test,dense1,dense2,dense3,dense4,dense5)

]: array([1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1,
          0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1])
```

Analysis

Neural Networks consumes lot of time for computations

The same problem could be also solved with linear regression

These neural networks are best to use with an cloud computing machine like ec2

There should not be any difference in terms of neurons from the output of one dense layer to the other layer

Activation function Sigmoid is chosen because output of my application is binary, Sigmoid comes of good use for binary predictions

Learning rate should be chosen carefully, Less learning rate takes more iterations and high Learning rate takes less iterations to reach the threshold cost or loss.

Usage of NumPy is recommended, writing your own mathematical calculations takes toll on runtime

Good to use these neural networks when trying to model complex relationships

Even without proper input the network tends to give some information

The duration of the network to compute changes from time to time. We can’t predict the time of the neural network takes to complete

Improper choosing of activation derivation function may make the neural network weights unchanged resulting in constant loss all the time.

CONCLUSION

Building a Linear Regression and Neural Network Model from scratch through python has got lot of advantages

One could build their own model or custom model according to the problem

Linear Regression and Neural Network both depends on weights to predict output or to the train the data, but whereas a neural network model is quite complex model involved with lot of feedback.

Neural Network can be used for complex models like health care where the domain itself is as complex as neural network.

Solving Neural Network problems on normal PC is not feasible because it takes lot of computational resources. It’s very good to run Linear Regression on a normal PC

Using of inbuilt libraries like Pandas, NumPy, SciPy is highly recommended.

Data Preprocessing before training to either neural network or Linear Regression is necessary.

Data Preprocessing and Cleaning yields very good results improves predictive power.

Neural Network activation functions and their derivates should be chosen very carefully according to the application.

Improper choosing of activation function makes the neural network bad and predict very poor results

Neural networks should be run in parallel computing machines for better processing and faster results

References

1)Neural Networks from Scratch in Python by Harrison Kinsley, Daniel Kukieła

2) http://www.stat.yale.edu/Courses/1997-98/101/linreg.htm

3)medium

4)stack overflow