

TiML

Hello reader,

The world of artificial intelligence is growing faster each day and its associated algorithms more complex. I sometimes struggle with the abundance of techniques available, and the options to choose from when starting a new project. For any project, one can opt for not only different kinds of models, but additional choices need to be made along with the careful consideration of each model. Choosing the correct optimizer, learning rate, loss function, train/test split size are just a few of the general considerations. Then choices need to be made about model-specific optimizations. To me and some of my fellow peers, these choices are just too many, and it is impossible to understand each one of them. However, to effectively use these powerful methods, it is important to – at the least – have a vague notion of what they mean.

With this in mind, I have attempted to document the implementation of various machine learning algorithms, along with explanations, information, and some illustrations. Most Python implementations require the use of the NumPy library. It is highly recommended that you install this library as well, if you want to replicate any of my work. The actual code on GitHub might differ slightly from some of the code snippets included in this document. The code does not work any differently. It has purely been made easier to read for the layman here.

Since I am writing this in the early stages of this project (August 2022), with only a few finished algorithms, this document and its accompanying code are most likely flawed in more ways than one, and some information you read might be incorrect. If you stumble upon such mistakes or have any other questions, please, do not hesitate to send an email to tim.koornstra@gmail.com or create an “issue” on the GitHub page for this project (<https://github.com/TimKoornstra/TiML/>).

Table of contents

Table of contents	2
Part I: Loss functions.....	3
R-Squared	3
Mean Squared Error (MSE).....	4
Mean Absolute Error (MAE).....	4
Part II: Simple Models	6

Part I: Loss functions

You might be disappointed that I do not start this document off with the most exciting things that the magical world of artificial intelligence has to offer. However, **error** or **loss functions** are some of the most important things to consider when training and testing your model. It is the way we tell our model how well it is performing. If the model is performing poorly, we must let it know that it is doing so, so it knows that it has some area to improve upon and it needs to rearrange some *weights* (i.e., *reweigh* information differently to get to a better answer). Different modeling problems require different loss functions. This section aims to describe the implementation and use of some of these functions.

R-Squared

The R-Squared or R^2 loss function is mostly used in Linear Regression and is a measure of describing how well the regression predicts the approximate real data points. It describes how much the variance in the dependent variable is explained by the independent variables collectively. In other words: how well your model can explain the difference between the ground truth and its prediction. If you have a higher R-Squared, your model better fits your observations. If you have a lower R-Squared, your model cannot explain all the variance in the response variable. This measure is scaled from 0-100%. There is no need to panic if your calculated R^2 is lower than 100%, since there might not exist a perfect fit for this data using the model type you use. It might not be possible to create a perfect fit for your model because of noise in your data, or maybe you should think about using another type of model.

Because we want to know the variability of data around the mean, we can calculate the R-Squared using two sum of squares formulas. First, we calculate the sum of squares of residuals (residual sum of squares). The residual is the difference between the observed value and the estimated value of the quantity of interest (e.g., sample mean). Then, we also want to calculate the total sum of squares. Once we have those, we can calculate our loss.

Sum of squares of residuals:

$$SS_{res} = \sum_i (y_{true}^i - y_{pred}^i)^2 \quad (1)$$

Sum of total variance:

$$SS_{tot} = \sum_i (y_{true}^i - \text{mean}(y_{true}))^2 \quad (2)$$

R-squared:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} \quad (3)$$

Using these formulas, the implementation in Python is very straightforward.

```
def r_squared(y_true, y_pred):  
    SS_res = np.sum(np.square(y_pred - y_true))  
    SS_tot = np.sum(np.square(y_true - np.mean(y_true)))  
    # Return the R squared score  
    return 1 - SS_res / SS_tot
```

Snippet 1 - R-Squared implementation in Python

Mean Squared Error (MSE)

The mean squared error (MSE) is a very straightforward loss function: for each of the data points, we calculate the difference between the prediction and the ground truth. To punish big errors more than small errors, we square it. Then, we want to average these squared errors to get the intuitive mean squared error. This error is scaled from 0 to infinity. It is used for all sorts of problems but is most often employed when we want to penalize big differences. When predicting crowdedness for the metro, for example, we do not really mind if we predict 3 passengers, and there are truly 4. However, we do not want to predict 300 passengers, when there are 400. This could also be achieved with the [mean absolute error](#), but because that loss is linear, there is not as much “urgency” to the model to correct these predictions.

The implementation in Python is also very simple, and can be described in one line of code:

```
def mean_squared_error(y_true, y_pred):  
    return np.mean(np.square(y_pred - y_true))
```

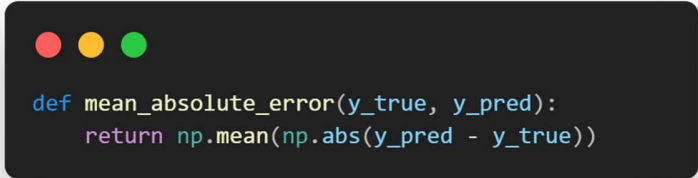
Snippet 2 - MSE implementation in Python

Mean Absolute Error (MAE)

The mean absolute error (MAE) is even more straightforward and easy to understand than the MSE. In essence: we sum the absolute value of the prediction, subtracted by the true value for each of our data points, and take its mean (i.e. divide it by N data points).

This is a very general-purpose loss function and is used when big errors are not that important. Another benefit of this error is that it is very intuitive and can therefore be explained to the layman very easily. It is simply the average “mistake”.

This implementation in Python is similar to the MSE, and can also be described in one line of code:

A dark-themed code editor window with three colored window control buttons (red, yellow, green) in the top-left corner. It contains a single Python function definition for calculating the Mean Absolute Error (MAE).

```
def mean_absolute_error(y_true, y_pred):  
    return np.mean(np.abs(y_pred - y_true))
```

Snippet 3 - MAE implementation in Python

Part II: Simple Models

We have finally arrived at the *good* part: actual machine learning! Well, not exactly... but we are close. To understand the more complex algorithms, it is important to first understand the basics. Do not be discouraged, because “simple” models oftentimes outperform very complex models. Not only are they more interpretable, but also easier to train. For most of the models described in this part, we do not need fancy GPUs and big supercomputers to train everything, but we can do it within seconds (depending on the size of the dataset, of course). Still, some of the stuff is quite difficult to understand – especially if you have trouble with math. I will try, however, to explain these methods as simple as possible, without losing too much information.

Linear Regression

Linear regression is not a method that is unique to the field of Artificial Intelligence. It is a well-known statistical approach for modeling the relationship between a scalar response and one or multiple explanatory variables. We can, for example, ask ourselves the question “How much will the value of my house increase if I were to repaint it?”. In this case, we want to know the price difference (the scalar response) when we repaint the house (an explanatory variable). *Simple* linear regression (i.e. linear regression with one response, and one variable) is best described by the following function:

$$y = mx + b \quad (4)$$

You might recognize this formula from high school math. In the previous example: y is our house price, m is the added value of one x amount of paint added to the house, and b is the original house price. We can imagine that if we do not paint our house ($x=0$), the price will remain the same as before, whereas the price should increase the more we paint our house (until it is fully painted).

Then there is also something called *multiple* linear regression. The amount of responses stays the same, but the amount of *variables* is now bigger than 1. We must rewrite our original formula to include not only multiple variables, but also multiple coefficients that belong to those variables. We get something that looks like the following:

$$y = \sum_i m_i x_i + b \quad (5)$$

Or, when using matrix notation:

$$y = \mathbf{w}^T \mathbf{x} \quad (6)$$

Notice that the y -intercept (the bias term b) has disappeared. This is because it is usually included within the weight and variable vectors.

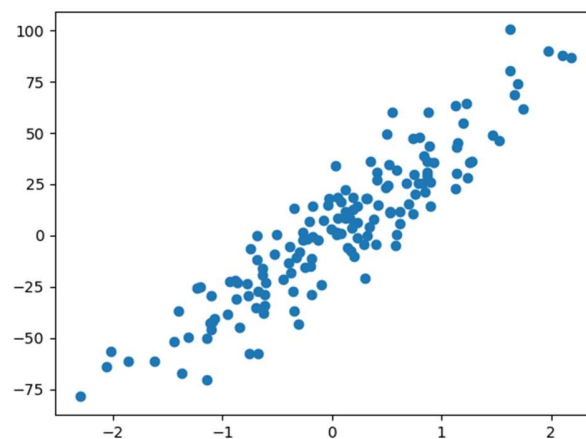
Lastly, there is also *multivariate* linear regression, where we predict multiple correlated variables, rather than a single scalar variable. In matrix format, we usually get a function like the following:

$$y = w^T x \quad (7)$$

This looks very similar to the previous formula but has a y vector as output, rather than a y scalar.

Implementation

To describe the implementation as best as possible, it is useful to add an illustrated example. We have the following data points in a scatter plot:



It is obvious that we cannot come up with a linear function that describes all the data points exactly. Therefore, we want to approximate a linear function that describes the relationship between the x and the y axis as accurately as possible.

To do so, we use the least squares method. Without boring you with all of the details, it is most important to know the following: it is an approximation method to minimize the sum of squared residuals. As explained in the losses section, a residual is a difference between the observed value and the estimated value of the quantity of interest (e.g. sample mean). Using a lot of math, it can be rewritten as the following formula:

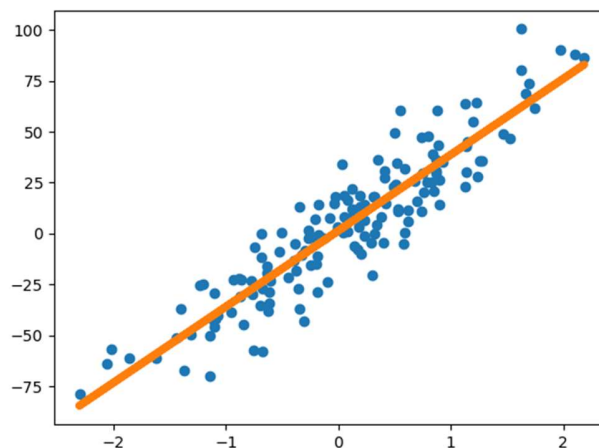
$$\beta' = (X^T X)^{-1} X^T Y \quad (8)$$

Note that β' is the estimated coefficient matrix. The implementation of this in Python looks complicated but is actually very straightforward. Take a look at the snippet below:

```
def linear_regression(X , y):  
    # First, add a bias term  
    X = np.c_[X, np.ones(X.shape[0])]  
  
    # Calculate the weights and bias using Least Squares  
  
    #  $(X^T X)^{-1}$   
    XTX_inv = np.linalg.inv(np.dot(X.T, X))  
    #  $XTX_{inv} X^T$   
    inv_XT = np.dot(XTX_inv, X.T)  
    #  $\beta = (X^T X)^{-1} X^T y$   
    beta = np.dot(np.dot(np.linalg.inv(np.dot(X.T, X)), X.T), y)  
  
    # Store the weights and bias  
    weights = beta[:-1]  
    bias = beta[-1]
```

Snippet 4 - Linear Regression implemented in Python

And we are done! Yes, this is everything that is needed to implement linear regression from scratch. If we run our linear regression model on the datapoints from before, we get the following linear estimation:



Which looks very good. However, to estimate the goodness of a fit, the R-squared measure is most often used for linear regression. If we use our R-squared loss function, we find a score of 0.842. Remember, the closer to 1, the more variance in the data is explained by the model. Given that our model has quite a bit of noise, I would say that an 84.2% explanation of variance is a very good result.