Mathematical Modelling I: Some Theory on Machine Learning & Logistic Regression

Kristoffer Vinell

2019-03-05

Short Introduction to Machine Learning

Machine Learning has gained a lot of traction the last few years and has successfully been implemented across various fields, such as e-commerce, finance and medicin, to name a few. With roots in mathematical statistics and mathematics, Machine Learning techniques are today able to solve a diverse range of problems, such as fraud detection, image recognition and optimised product recommendations. In later years, with the massive increase of data volumes, its performance has sky-rocketed and can in many applications perform better than most humans.

Machine Learning can most amply be described as a set of algorithms that can draw conclusions from previously unseen data. For instance, an accurate image recognition algorithm can classify an image of a special dog breed although it hasn't been acquainted with that particular breed before. Broadly speaking, Machine Learning models can be divided into two categories, namely

supervised learning, where an already labeled dataset is used to teach the model how to generalise to other examples

unsupervised learning, where there is no human ”intervention”

Although the latter example is very interesting in its own right, including methods such as *clustering*, where the end goal is to group similar data points together based on some metric, we are in the following going to focus on supervised learning.

Supervised Learning

In supervised learning, a model is *trained* on an already labeled dataset in order to determine the optimal parameters within the model. The labeled dataset is often produced by hand, e g a human has labeled each training example (*dog*, *cat*, *cucumber* in case of the image recognition example used above).

To formalise this idea in a more mathematical way, one could say that supervised learning infers a *conditional probability distribution* $P(Y|X)$, given labeled data X, whereas unsupervised learning intends to infer an *a priori probability distribution* $P(Y)$. The idea of feeding data into the model is really quite different from classical mathematical modelling. One thing that all of these modelling techniques often have in common is the fact that they rely on optimisation theory to produce accurate results. This will be studied in more detail in the following.

Let's now get back to how a supervised model is trained. Firstly, as is the case for most mathematical models, some model parameters are involved. During the modelling phase, we need to *fit these parameters* in some way so that the model amply describes the system it was intended to describe. In Machine Learning, this is called *training* the model, using already labeled data (also referred to as the *training set*). The trained model is then used to make *predictions* about previously unseen data.

As an example, say that we are given the task of predicting the housing prices in Stockholm, and are given a large dataset containing price history. Let's say that this dataset contains a number of variables (e g area code, number of rooms, number of square feet, etc). We could then start off with a simple, linear model like so:

$$\hat{y} = \beta_0 + \beta_1 x_1,$$

where x_1 is the number of rooms of the house and $\hat{y}$ is the predicted price. β_1 and β_0 (also called the *intercept*) are model parameters. The variable x_1 is called a *predictor* or, more commonly with the realm of Machine Learning, *feature*. The prediction is in this context called a *target variable*. Noticed that we've only chosen to use one predictor out of the many supplied to us in the dataset. This selection process is called *feature engineering* in the Machine Learning world, or sometimes *variable selection*.

Although linear models (such as the *linear regression* above) are very powerful in their ease of use and transparency, they are not the right tool for the job in all situations. Let's say instead that we are faced with the challenge of building a model that can classify emails as spam or not. Given linear regression, the target variable could attain any value on $\mathbf{R}^+$, and we would thus need to map that continuous variable into a binary outcome (spam, no spam). Another kind of model is now introduced that can help us overcome the unboundedness of linear regression.

Logistic Regression

Given that we want a binary outcome (0 or 1) for our predictions, but still want to keep some of the nice properties of a linear model, what can we do? As linear regression is unbounded, we would need a function that can map the real numbers to a finite interval, say $[0, 1]$, i e

$$f\colon \mathbf{R} \to [0, 1]$$

One such function that seems to fit the bill is the *sigmoid function*, defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}, x \in \mathbf{R}$$

Apart from being a continuously differentiable function on $\mathbf{R}$, we can easily see that the following properties hold:

$$\begin{aligned}
\lim_{x \to -\infty} \sigma(x) &= 0 \\
\lim_{x \to +\infty} \sigma(x) &= 1 \\
\sigma(0) = \frac{1}{2}
\end{aligned}$$

These properties should give you an idea about what the function looks like already, but if you feel so inclined, feel free to plot it using your favourite tool. (I suggest using Python's `matplotlib` package that you can install from the Jupyter Notebook you will use for the final project.)

Hypothesis & Estimated Probability

Now, if we were to apply this function to a general linear model, we would get the following relationships:

$$\begin{aligned}
z &= \theta_0 + \theta_1 x_1 + \ldots + \theta_N x_N \\
h_\theta(\mathbf{x}) &= \sigma(z)
\end{aligned}$$

We call function h the *hypothesis*, which is the estimated probability. It is used to infer how confident we can be that the predicted value is the actual value for a given input $\mathbf{x}$, where $\mathbf{x}$ is a vector of features. This can be written in vector format like so:

$$\begin{aligned} z &= \theta^T \mathbf{x} \\ h_\theta(\mathbf{x}) &= \sigma(z) \end{aligned}$$

where

$$\begin{aligned} \theta &= \begin{pmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_N \end{pmatrix} \\ \mathbf{x} &= \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_N \end{pmatrix} \end{aligned}$$

Note the leading 1 in the feature vector to account for the intercept θ_0. Formalising these relationships in vector form will greatly simplify not only readability but also make it easier to implement in a computational environment such as Python. Vectorisation is also a great way to speed up your implementation and keeping your code small and concise. This will be useful when working on the final course project.

If we let y be the actual value (0 or 1), then this can be written as the following probabilities:

$$\begin{aligned} h_\theta(\mathbf{x}) &= P(y = 1|\mathbf{x}, \theta), \\ P(y = 0|\mathbf{x}, \theta) &= 1 - P(y = 1|\mathbf{x}, \theta) = 1 - h_\theta(\mathbf{x}), \end{aligned}$$

thanks to basic axioms from probability theory. These relationships make it clearer how the estimated probability relates to the actual value.

Decision boundary

As a final step, we need to find a way to turn the estimated probability into a binary value (0 or 1) to make the model useful. Given the above probability relationships, it is natural to define the predicted outcome, $\hat{y}$, as

$$\hat{y} = \begin{cases} 0, & \text{if } h_\theta(\mathbf{x}) < \frac{1}{2} \\ 1, & \text{if } h_\theta(\mathbf{x}) \geq \frac{1}{2} \end{cases}$$

This is sometimes referred to as the *decision boundary.*

Cost Function

Now that we have a mathematical formulation of the model, it is time to look into how we can fit the model's parameters to optimise the accuracy of our model. For this purpose, we introduce a *cost function* that estimates how well a set of parameters performs in terms of model accuracy.

Assume that we have a training set with M labeled training examples, that is, feature vectors $\mathbf{x^{(1)}}, \dots, \mathbf{x^{(M)}}$ with corresponding labels $y_1, \dots, y_M$. We would like to take into account all of these training examples in order to produce model parameters θ that yields the best accuracy of the model. A natural way to measure this would be to use the *mean squarred error* as cost function, namely

$$J(\theta) = \frac{1}{M} \sum_{i=1}^{M} \left(h_\theta(\mathbf{x^{(i)}}) - y_i \right)^2$$

Although this is very easy to interpret, we would run into trouble using this cost function during the optimisation stage if we are aiming to use *gradient descent*. Given that the cost function cannot be guaranteed to be convex, we might end up with a local optimum. However, if we would have a convex cost function, we would be sure that gradient descent would converge towards the global optimum instead.

For this purpose, the following cost function is usually used in logistic modelling:

$$J(\theta) = -\frac{1}{M} \sum_{i=1}^{M} \left(y^{(i)} \log\left(h_\theta(x^{(i)}) \right) + \left(1 - y^{(i)} \right) \log\left(1 - h_\theta(x^{(i)}) \right) \right)$$

Although this looks a bit daunting, let's keep in mind that $y^{(i)}$ is either 0 or 1, which means that many terms will disappear. Hopefully we will have time to show that this function is indeed convex in class, but let's at least note for now that it is differentiable.

Optimisation Using Gradient Descent

Given the last section we now have the following optimisation problem at hand:

$$\min_\theta J(\theta), \quad \theta \in \mathbf{R}^n$$

Recall from the theory on gradient descent optimisation that the parameters are updated according to

$$\theta_j = \theta_j - \alpha \frac{\partial J}{\partial \theta_j},$$

where α is the *learning rate*.

It can be proven that the partial derivates are

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{M} \sum_{i=1}^{M} \left(h_\theta \left(\mathbf{x}^{(\mathbf{i})} \right) - y^{(i)} \right) x_j^{(i)}$$

Note that it is starting to become quite messy with all the subscripts and superscripts. Luckily, the partial derivatives can be formalised as vector computations, much like we did for the hypothesis function h_θ. It will also be very useful when doing the Python implementation of gradient descent during the project work. I will leave the details to you as an exercise.

Access to Final Project

Using the theory presented above, your task is now to implement a fully-functioning Machine Learning model utilising logistic regression. The project description and access to the computational environment (Python) can be found following this link:

`https://github.com/kvinell/math-modelling-1-project`

In case of issues, don't hesitate to contact me via email: `kristoffer.vinell@gmail.com`.

Appendices

Derivation of Partial Derivatives for Cost Function

We are in the following going to derive the partial derivates for the cost function J for an arbitrary θ_i.

$$\begin{aligned}
\frac{\partial J}{\partial \theta_j} &= \frac{\partial}{\partial \theta_j}\left(-\frac{1}{M}\sum_{i=1}^{M}\left(y^{(i)}\log\left(h_\theta(x^{(i)})\right)+\left(1-y^{(i)}\right)\log\left(1-h_\theta(x^{(i)})\right)\right)\right)\\
&= -\frac{1}{M}\sum_{i=1}^{M}\left(y^{(i)}\frac{\partial}{\partial \theta_j}\left(\log\left(h_\theta(x^{(i)})\right)\right)+\left(1-y^{(i)}\right)\frac{\partial}{\partial \theta_j}\left(\log\left(1-h_\theta(x^{(i)})\right)\right)\right)\\
&= -\frac{1}{M}\sum_{i=1}^{M}\left(y^{(i)}\left(\frac{1}{h_\theta(x^{(i)})}\frac{\partial h_\theta}{\partial \theta_j}(x^{(i)})\right)+\left(1-y^{(i)}\right)\left(\frac{-1}{1-h_\theta(x^{(i)})}\frac{\partial h_\theta}{\partial \theta_j}(x^{(i)})\right)\right)\\
&= -\frac{1}{M}\sum_{i=1}^{M}\left(y^{(i)}\left(\frac{1}{h_\theta(x^{(i)})}\frac{\partial h_\theta}{\partial \theta_j}(x^{(i)})\right)-\left(1-y^{(i)}\right)\left(\frac{1}{1-h_\theta(x^{(i)})}\frac{\partial h_\theta}{\partial \theta_j}(x^{(i)})\right)\right)\\
&= -\frac{1}{M}\sum_{i=1}^{M}\left(\frac{y^{(i)}}{h_\theta(x^{(i)})}-\left(\frac{1-y^{(i)}}{1-h_\theta(x^{(i)})}\right)\right)\frac{\partial h_\theta}{\partial \theta_j}(x^{(i)})
\end{aligned}$$

Now, let's focus on the partial derivative in the last step. Remember that

$$\begin{aligned}
z &= \theta^T\mathbf{x}\\
h_\theta(\mathbf{x}) &= \sigma(z)
\end{aligned}$$

So,

$$\begin{aligned}
\frac{\partial h_\theta}{\partial \theta_j} &= \frac{d\sigma}{dz}\frac{\partial z}{\partial \theta_j}\\
&= \sigma(z)\left(1-\sigma(z)\right)\frac{\partial z}{\partial \theta_j}\\
&= \sigma(z)\left(1-\sigma(z)\right)x_j\\
&= h_\theta(\mathbf{x})\left(1-h_\theta(\mathbf{x})\right)x_j
\end{aligned}$$

Using this result above yields

$$\begin{aligned}
\frac{\partial J}{\partial \theta_j} &= -\frac{1}{M}\sum_{i=1}^{M}\left(\frac{y^{(i)}}{h_\theta(x^{(i)})} - \left(\frac{1-y^{(i)}}{1-h_\theta(x^{(i)})}\right)\right) h_\theta(x^{(i)})\left(1-h_\theta(x^{(i)})\right) x_j \\
&= -\frac{1}{M}\sum_{i=1}^{M}\left(\frac{y^{(i)}h_\theta(x^{(i)})\left(1-h_\theta(x^{(i)})\right)}{h_\theta(x^{(i)})} - \left(\frac{\left(1-y^{(i)}\right)h_\theta(x^{(i)})\left(1-h_\theta(x^{(i)})\right)}{1-h_\theta(x^{(i)})}\right)\right) x_j \\
&= -\frac{1}{M}\sum_{i=1}^{M}\left(y^{(i)}\left(1-h_\theta(x^{(i)})\right) - \left(\left(1-y^{(i)}\right)h_\theta(x^{(i)})\right)\right) x_j \\
&= -\frac{1}{M}\sum_{i=1}^{M}\left(y^{(i)} - y^{(i)}h_\theta(x^{(i)}) - \left(h_\theta(x^{(i)}) - h_\theta(x^{(i)})y^{(i)}\right)\right) x_j \\
&= -\frac{1}{M}\sum_{i=1}^{M}\left(y^{(i)} - y^{(i)}h_\theta(x^{(i)}) - h_\theta(x^{(i)}) + h_\theta(x^{(i)})y^{(i)}\right) x_j \\
&= -\frac{1}{M}\sum_{i=1}^{M}\left(y^{(i)} - h_\theta(x^{(i)})\right) x_j \\
&= \frac{1}{M}\sum_{i=1}^{M}\left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)}\right) x_j
\end{aligned}$$