FYS-STK 4155 Project 2

Espen Lønes

December 15, 2021

Abstract

Machine learning regression applied to vinho verde wine samples. The datasets consists of (4898) white and (1599) red wine samples with 11 analytical tests as features. Feature analysis and selection was also done.

1 Introduction

Portugal is in the top 10 worldwide when it comes to wine production and export. Being responsible for 3.17% of the worlds wine in 2005 [5]. Export of its vinho verde wine has increased greatly in the past years [6]. In response to this growth the industry has invested in new technologies used in for production and sales purposes. In this context certification and quality assessment is vital. As a part of these processes the wine is often assessed using physiochemical and sensory tests [7]. The test preformed include measures of i.a. density, alcohol, pH. The sensory tests rely on a human expert. The relation between the sensory and analytical data is complex and not well understood [8]. Making wine quality regression/classification a difficult task.

In this project i attempt to use different ML regression methods to predict the sensory (expert) rating from the analytical measurements like atemplted by P. Cortez et al. [2]. As mentioned in their paper if classification/regression is possible one could extend this to the consumers. By recommending wines based on their personal sensory preferences.

2 Theory

Linear regression

Linear regression predicts labels for a continuous dependent variable. The prediction is done by calculating the weighted sum of input features. Having inputs $X = (x0, x1, ..., xN - 1)$ and output y. We get a prediction $\tilde{y}$.

$$\tilde{y} = \sum_{i=0}^{N-1} X_i \beta_i$$

where β_j are the unknown weights used to train the model in order to minimize the mean squared error.

Gradient descent

The gradient of a function f for a given variable x tells us how much the function changes with x. A function is often depended on multiple variables. It's gradient is then found by finding its partial derivatives.

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots \right)$$

Gradient decent is an iterative method where given a starting point, usually random but may be explicitly chosen. The method moves along the function gradient until it reaches a minimum in the search space (hopefully the global minimum). For a given task this search space is defined by chosen cost function, quantifying how far from the optimal/best solution our model is. Since our goal is to reach the minimum we move in the opposite direction of the gradient. This direction is found by calculating the gradient of the search space at the current point. We then move a given length in that direction to get our new solution. This repeats until we get to a minimum where the gradient is zero. If we are at point w_i with cost function $C(w)$ we get the iterative scheme.

$$w_{i+1} = w_i - \eta_k \nabla_w C(w_i)$$

Where η_k is the step length commonly names learning rate. The point w corresponds to a set of values we call weights.

As previously stated our goal is to reach the global minimum. But depending on cost function choice, leaning rate, initial guess and problem complexity we may (and usually will) end up in a local minimum instead. A dilemma with learning rate is that having to small a learning rate makes the training slow as we need many iterations until we reach a minimum. On the other hand if it is to large we will probably overshoot the minimum. A popular method for mitigating this is by having a decreasing learning rate. With the idea that we move fast towards a minimum, as we get close the learning rate decreases making it less likely we overshoot the minimum.

Stochastic gradient descent

Calculating the cost function gradient can be computationally heavy. We may instead use a version of gradient descent called stochastic gradient descent to speedup this process. SGD works by dividing the points into groups called mini-batches. We then choose a random batch and compute its gradient. If the sample size is large enough the average of these gradients will approximate the gradient calculated for all data points. The flow of the SGD method is therefore as follows. The SDG picks a random mini-batch of training points and calculates its gradient. Then the next mini-batch is chosen until all batches are used. This is called one epoch. We then use these as the gradient in gradient descent and may also then start another epoch. Since this method only approximates the gradients means that some randomness is introduced in the gradient. This has the added benefit of possibly getting us out of a local minima.

As a step to further increase the algorithms speed, SGD is often used with a momentum factor. The momentum increases the acceleration of the gradient giving a higher convergence rate.

Defining momentum $0 \leq \gamma \leq 1$ we get this scheme for our weights.

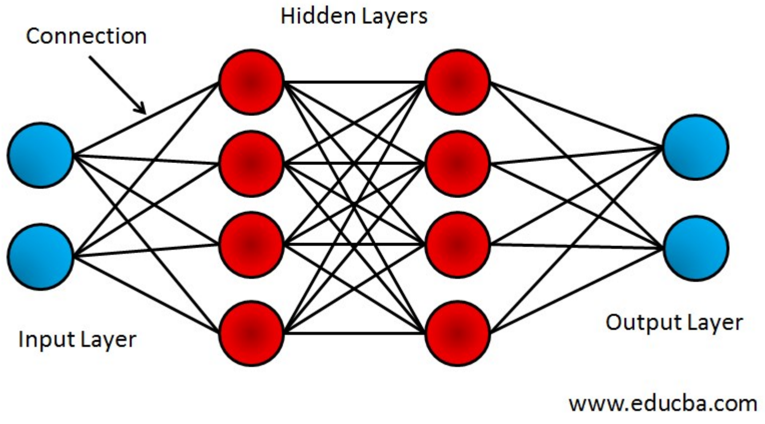
$$v_{i+1} = \gamma v_i + \eta_k \nabla C(w_i)$$

$$w_{i+1} = w_i - v_{i+1}$$

(Usualy having $v_{-1} = 0$)
High γ values gives large acceleration. With $\gamma = 0$ being ordinary SGD.

Neural networks

A neural network consists of layers of nodes, one input, one output and a any number of hidden layers (may have no hidden layers). Each layer has a set of neurons/nodes, for the input layer these are the input neurons where the number of neurons are determined by our problem and our representation of it (the features). The output neurons represent our solution/answer.



Feed forward neural network (FFNN)

A type of neural network where the output from one layer is used as input to the next. These are the simplest and easiest forms of neural network and use algorithms like back propagation to adjust the neuron weights.

Activation functions

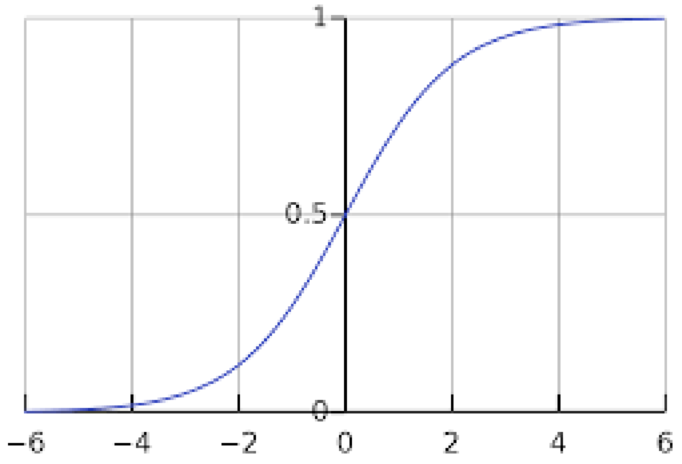
In neural networks the sum of the weighted inputs to a node, is used as input to that nodes activation function. Activation functions are divided in two main groups, linear and non-linear. The linear functions are lines/hyperplanes and are not bounded by any range, the identity function is such a function $\sigma(z) = z$.
The non-linear functions are more often used as they usually make the model better at generalizing and adapting to different data. (for non-linearly separable problems, which most are).

Sigmoid functions are a set of s-shaped functions often used as activation functions in neural networks. One specific sigmoid that has been used frequently in NNs is the logistic function, given by.

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

With the derivative

$$\sigma'(z) = \sigma(z)(1-\sigma(z))$$



Plot of the logistic function.

Its advantage lies in that it's values fall between 0 and 1. This is very useful when the output is a probability. Unfortunately the logistic function can halt the NN as the derivative is often very close to zero. We now often use functions like ReLU and leaky ReLU instead.

The ReLU (rectified linear unit) function is variant of the sigmoid.

$$\sigma(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ z & \text{if } z > 0 \end{cases} = max\{0, z\}$$

Derivative

$$\sigma'(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$

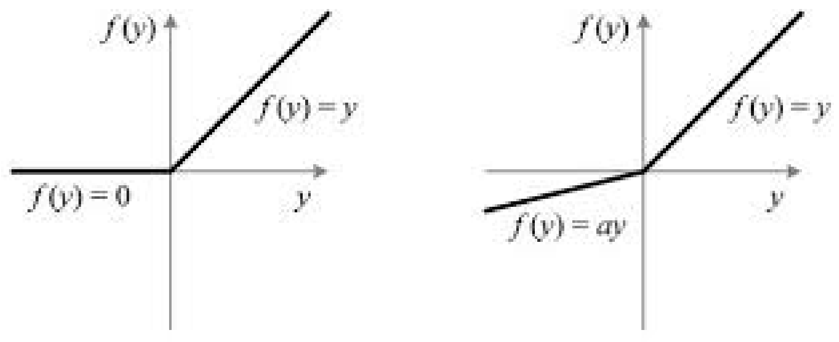
We see that for ReLU increasing the weights does not saturate the learning. But it does if the weights are negative. ReLU is often used for the hidden layers.

Another variation is the Leaky/Parametric ReLU that does not saturate for negative weights, instead of zero for negative input it multiples it with a (usually quite small e.g. 0.01).

$$\sigma(z) = \begin{cases} az & \text{if } z \leq 0 \\ z & \text{if } z > 0 \end{cases}$$

$$\sigma'(z) = \begin{cases} a & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$

As the gradient of leaky ReLU newer goes to zero the neurons never stop learning. Making this activation function very robust.



ReLU on the left. Parametric ReLU on the right.

Cost functions

Cost functions estimates how right or wrong a prediction is, by a given metric. It uses predicted values and a true values in order to quantify the quality of our current model. In neural networks this value is used to update the weights and biases. For liner regression we use the mean squared error (MSE), while for logistic regression cross entropy is used.

The MSE cost function.

$$C(w) = (Xw - z)^T(Xw - z) + \lambda w^T w$$

Where X is the design/feature matrix, z is the true labels, w is the weights and λ is the regularization parameter.

MSE takes the averages of the squares of the errors. This works well when fitting to a line. For gradient descent we also needs its derivative.

$$\nabla C(w) = 2X^T(X^T w - z) + 2\lambda w$$

R2???

Backpropagation

Backpropagation is an algorithm used in updating the weights and biases of a neural network. It computes the gradient of the cost function in respect to the current weights and biases. The aim of backpropagation is to calculate the partial derivatives. $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$
This is done using the chain rule. Where the cost function depends on the activation function a, which in turn depends on the weighted sum z, which again depends on the weights and biases. Giving us.

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial a}\frac{\partial C}{\partial z}\frac{\partial C}{\partial w}$$

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}\frac{\partial C}{\partial z}\frac{\partial C}{\partial b}$$

Doing this individually for all weights and biases is very slow. Instead we calculate the gradient for each layer in reverse order. By doing this we avoid unnecessary calculations and duplicate values. It also makes it easier to see how changing the weights and biases change the output when starting on the last layer. The gradient of the weighted input(s) to a given hidden layer is called δ^l, (often called the error). We also use k to denote the k-th node in the $(l-1)$-th layer. And j for the j-th node in the l-t layer.

Backpropagation consists of a set of four equations. The first being for the error δ^l in the output layer.

$$\delta_k^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$$

Thereafter the error for the hidden layers $L-1, L-2, \cdots, 2$ are computed recursively.

$$\delta_j^l = \sum_k \delta_j^{l+1} w_j^{l+1} \sigma'(z_j^l)$$

These errors are then used to calculate the cost functions partial derivatives.

$$\frac{\partial C}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

Lastly the partial derivatives are used to update the weights and biases for layers $L-1, L-2, \cdots, 2$ using gradient decent.

$$w_{jk}^l \leftarrow w_{jk}^l - \eta\frac{\partial C}{\partial w_{jk}^l} = w_{jk}^l - \eta\delta_j^l a_k^{l-1}$$

$$b_j^l \leftarrow b_j^l - \eta\frac{\partial C}{\partial b_j^l} = b_j^l - \eta\delta_j^l$$

In order to avoid overfitting a regularization parameter λ may be added. This penalises the weight matrices. A common regularization parameter is the L2. The penalty is added to the cost function. The regularization forces the weights towards zero but not all the way to zero. This way a simpler and more generalizable solution with the same error will likely be selected. Thus reducing overfitting. The cost function with L2 is defined by.

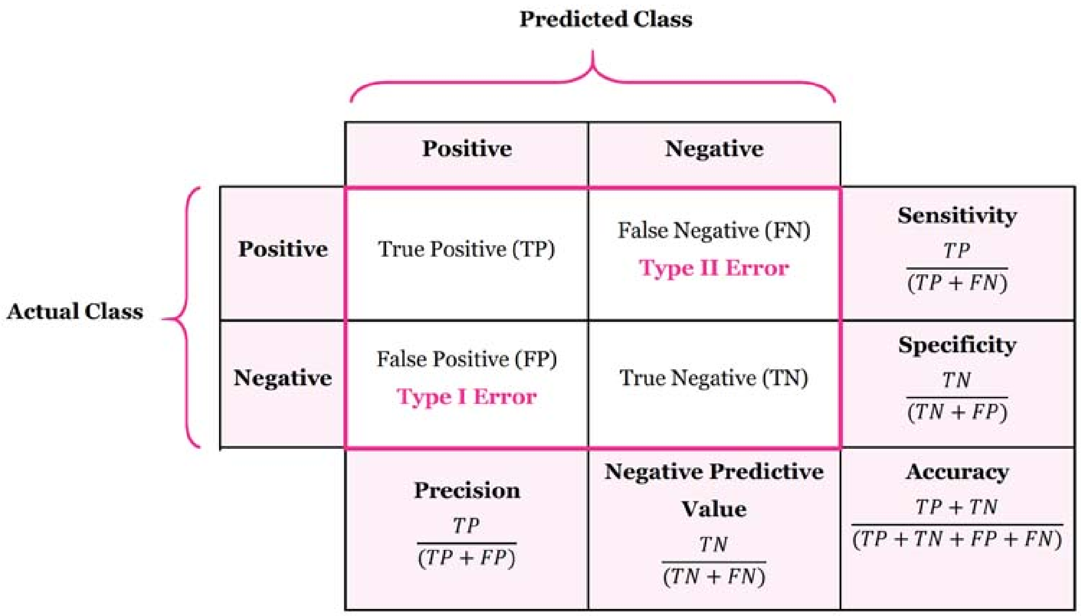
$$C(\theta) = L(\theta) + \lambda||w||_2^2$$

Having gradient:

$$\nabla C(\theta) = \nabla L(\theta) + 2\lambda w$$

Confusion matrix

The Confusion matrix is a simple yet extremely useful analysis tool when doing machine learning. It works for binary and multi-class problems by plotting how many of a given class was classified as the different classes.



Predicted Class

Actual Class

		Positive	Negative	
	Positive	True Positive (TP)	False Negative (FN) **Type II Error**	**Sensitivity** $\frac{TP}{(TP+FN)}$
	Negative	False Positive (FP) **Type I Error**	True Negative (TN)	**Specificity** $\frac{TN}{(TN+FP)}$
		Precision $\frac{TP}{(TP+FP)}$	**Negative Predictive Value** $\frac{TN}{(TN+FN)}$	**Accuracy** $\frac{TP+TN}{(TP+TN+FP+FN)}$

Here we also see three other metrics, sensitivity, specificity and accuracy. Which each provide a different look on how good our model is for different types of error.

3 Method

4 Results

5 Conclusion/Discussion

References

[1] `https://archive.ics.uci.edu/ml/datasets/Wine+Quality`

[2] P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 47(4):547-553, 2009. (`https://www.sciencedirect.com/science/article/pii/S0167923609001377?via%3Dihub`)

[3] Paulo Cortez, University of Minho, Guimarães, Portugal, http://www3.dsi.uminho.pt/pcortez A. Cerdeira, F. Almeida, T. Matos and J. Reis, Viticulture Commission of the Vinho Verde Region(CVRVV), Porto, Portugal @2009

[4] https://www.vinhoverde.pt/en/

[5] FAO
FAOSTAT — Food and Agriculture Organization Agriculture Trade Domain Statistics (July 2008)
http://faostat.fao.org/site/535/DesktopDefault.aspx?PageID=535

[6] CVRVV. Portuguese Wine — Vinho Verde.
Comissão de Viticultura da Região dos Vinhos Verdes (CVRVV),
http://www.vinhoverde.pt, July 2008.

[7] S. Ebeler Flavor Chemistry — Thirty Years of Progress, Kluwer Academic Publishers (1999), pp. 409-422 https://link.springer.com/chapter/10.1007%2F978-1-4615-4693-1_35

[8] A. Legin, A. Rudnitskaya, L. Luvova, Y. Vlasov, C. Natale, A. D'Amico Evaluation of Italian wine by the electronic tongue: recognition, quantitative analysis and correlation with human sensory perception Analytica Chimica Acta, 484 (1) (2003), pp. 33-34 https://www.scopus.com/record/display.uri?eid=2-s2.0-0038066312&origin=inward&txGid=cdf7d5eb8afe889f892d9b354a7e1317