Hidden learnings from “happy projects” - Part 1

Is it worth it for a data scientist to post how they solved a "Happy" project in their portfolio?

It depends.

In most of the articles I've read about creating portfolios to showcase Data science projects, almost all the authors agree on one thing: they strongly recommend not publishing "Happy" projects. A "Happy" project has been designed to run smoothly to teach a particular topic. The main idea for this recommendation is that no employer will be shocked knowing that you have been able to solve them.

However, in my opinion, even if your only objective is to sell yourself as a professional, some of these projects can find hidden learnings that are extremely valuable if they can be identified. They are like small breadcrumbs that we can pick up little by little. But why are they so valuable? For two simple reasons:

1. They are generally not visible to the naked eye and you will have to get dirty to find them, so they will better prepare you for future projects. They will make you a little wiser each time you manage to identify them.

2. They make you a desirable professional.

Considering the above, the complete answer to the initial question would be: if you have worked on a "happy" project and found some hidden learning that is valuable, it may be a good idea to publish your work if you think it can add some additional value.

Considering the above, I implemented some examples from the wonderful book "Neural Networks for Applied Sciences and Engineering" by Dr. Sandhya Samarasinghe, one of the best books in the area of neural networks. It very well exposes the necessary mathematical rigor with real case studies from the work and research activity of Dr. Samarasinghe.

Case: Identifyng the origing of fish from the growth-ring diameter of scales [1]

This is a simple case using real data and was solved using a single neuron perceptron.

"Environmental authorities concerned with the depletion of salmon stocks decided to regulates the catches. To do this, it is necessary to identify whether a fish is of Alaskan or Canadian origin. Fifty fish from each place of origin were caught, and the growth-ring diameter of scale was measured for the time they lived in freshwater and the subsequent time they lived in saltwater. The aim is to identify the origin of a fish from its growth-ring diameter in freshater and saltwater."

I couldn't find the data for this case on the web, however, since there is a scatter plot in the book, I was able to approximate it with a ruler and pencil.

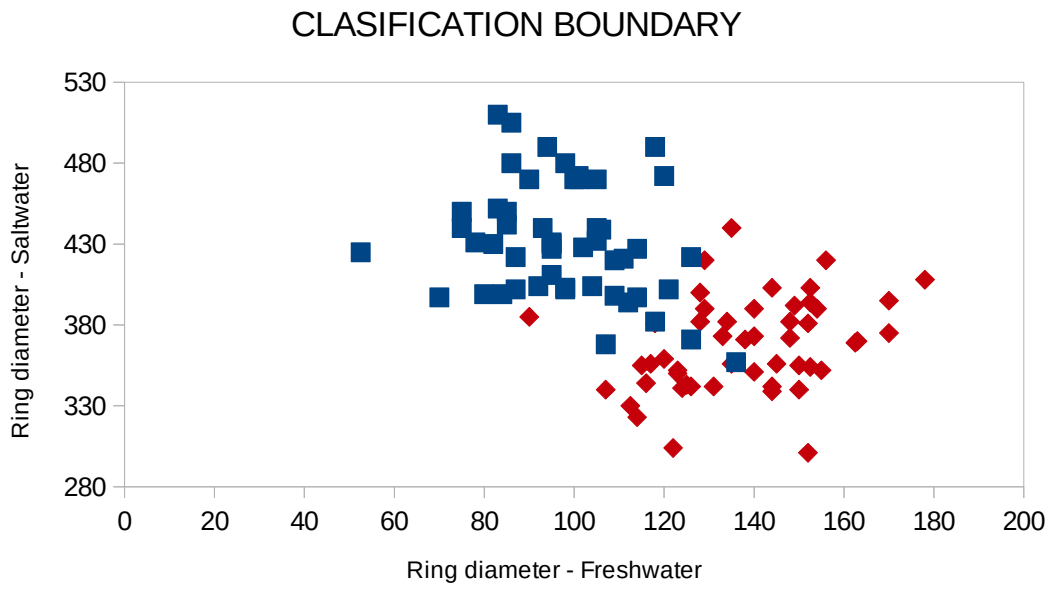

Fig 1. The growth-ring diameter of salmon in freshwater and saltwater for Canadian (red) and Alaskan (blue). Source: Samarasinghe S. [1]. Own elaboration.

Canadian fish are class 0 and Alaskan fish are class 1.

To identify both classes, I programmed a single neuron perceptron (with Hebb rule) using python from scratch. In just 80 periods and with a learning rate $\alpha = 0.00001$, it was possible to separate both classes with total correctness of 96% (98% Canadian and 94% Alaskan). Same result as Dr. Samarasinghe.

For comparison, I programmed the perceptron with the delta rule (which uses MSE as the error function), but contrary to expectations, the accuracy did not improve. The best result I got was 92% (99% Canadian and 93% Alaskan) also in 80 epochs. In fig. 2 you can see the classification boundaries on the data.

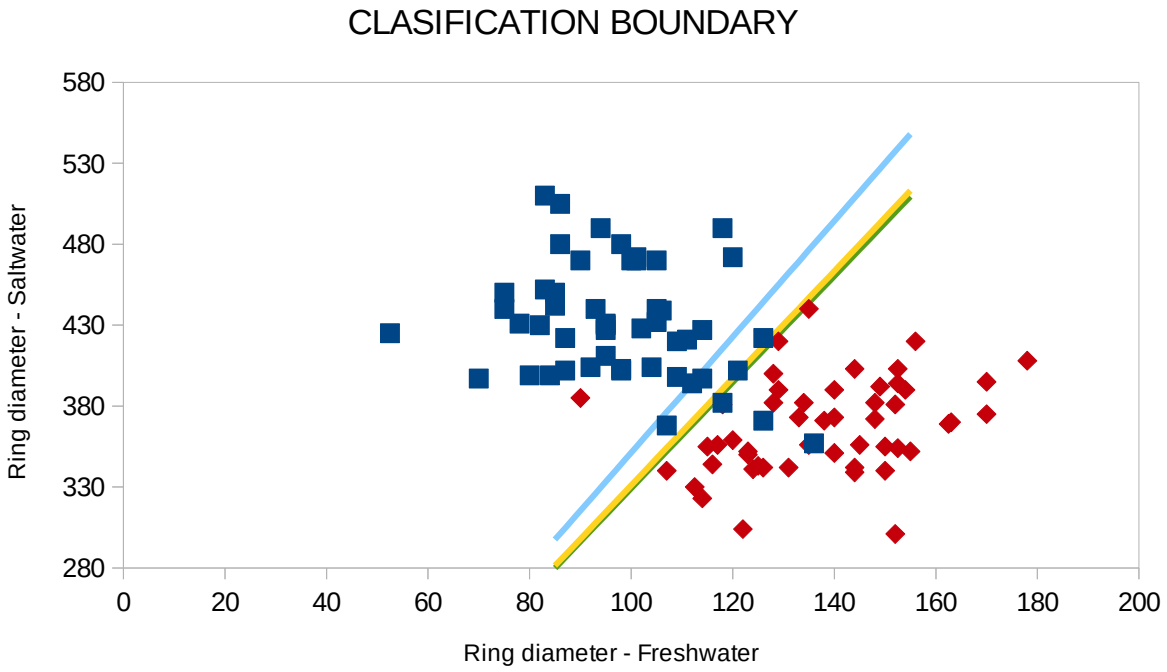

Fig 2. Samarasinghe's classifier (yellow, 96% acc), my classifier using Hebb rule (green, 96% acc), and my classifier using the delta rule (blue, 92% acc) superimposed on the data. Source: Samarasinghe S. [1]. Own elaboration.

The equation of the classification boundary (yellow line) obtained by Dr. Samarasinghe is:

$$x_2 = 0.026 + 3.31x_1$$

The equation for the classification boundary (green line) using the Hebb rule is:

$$x_2 = 1.16 + 3.28x_1$$

The equation for the classification boundary (blue line) using the delta rule is:

$$x_3 = -6.81 + 3.58x_1$$

Since the data is linearly separable, it is sufficient to use a perceptron to do the job. However, I was curious to see if using an MLP (Multilayer Perceptron) could improve the result. To do this, I created a model (python only) with a hidden neuron and an output neuron (1-1 architecture), both with a sigmoid activation function.

The first executions were surprising because the error made in the first epochs did not decrease, that is, the algorithm never converged. This happens because in an MLP Backpropagation is used which in turn uses the derivative of the sigmoid function. The derivative of the sigmoid has the particularity that it is almost zero for values greater than 4, therefore, the error practically does not decrease. To avoid this problem I had to normalize the data using the Simple Range Scaling method:

$$x_{ni} = (x_i - x_{min})/(x_{max} - x_{min}).$$

I ran the algorithm with various α and epochs settings. The results can be seen in table 1.

Epochs	**α**	**Total Error**	**Wrong classifications**		**Total Accuracy**
			Class 0	**Class 1**	
20,000	0.5	2.25	5	1	94%
20,000	0.1	2.17	5	1	94%
1,000	0.1	2.67	5	1	94%
80	0.1	11.26	45	0	55%

Table 1. Results of the 1-1 MLP with different hyperparameters. Accuracy is worse than perceptron's. Own elaboration.

Adding a hidden neuron to the perceptron did not help improve accuracy. On the contrary, the results were poorer than the perceptron. Also, to get decent results, it was necessary to train the network much more than with the perceptron. An α below 0.1 makes training much slower and the error remains high. Using the same number of epochs required by the perceptron (80) gives an unacceptable result. I also tried other setups by adding more neurons to the model but the results were even poorer. This less efficient behavior than the perceptron is because backpropagation was designed to identify non-linear patterns.

To make sure that these results were not the product of bad programming on my part, I implemented an MLP using the Keras library. The advantage of using this tool is that all its functions are optimized and it has no bugs. I obtained the best result with a 2-2 architecture, with sigmoid and softmax functions respectively, categorical_crossentropy as error function, Stochastic Gradient Descent as an optimizer, 80 epochs, a learning rate α = 0.00001, and normalizing the data. This model failed to exceed 93% accuracy (94% Canadian and 92% Alaskan). Something important to note is that not even using Keras does the algorithm find the optimal solution in a single execution for a particular hyperparameter configuration: several runs must be performed to obtain a good result. The accuracy ranged from 19%

to 93%. The randomness of the weights plays an important role in each execution of the algorithm since a new space is generated for the error function with each execution, and therefore, a new gradient and a new local minimum to find. Depending on the configuration of (α, epochs) chosen, the convergence will be more or less efficient for each execution.

This exercise is very simple, however, by working on it I got some important hidden learnings:

1. When using the perceptron algorithms (Hebb rule or delta rule), completely different results can be obtained each time they are executed (even without changing the hyperparameters and using linearly separable data). The results can vary diametrically from practically no success to 96% correctness obtained with the perceptron and the Hebb rule. In the latter case, the reason is that the search for a local minimum is not guaranteed for the error function used (error = prediction - target), and since the weights are initialized randomly in each execution, very dissimilar results are obtained. When using the delta rule, it must converge to the solution if the data is linearly separable as in the case studied. However, convergence can be extremely slow. The same happens with backpropagation in an MLP. For this reason, it is necessary to execute each algorithm several times until you are sure that you have obtained the best percentage of hits for each configuration of the selected hyperparameters (on average, I performed between 6 and 8 executions for each configuration). You may also use regularization to prevent excessive weight growth and diminish this behavior (even though success isn't guaranteed).

2. If Backpropagation is used with the sigmoid as the activation function, it is recommended to normalize the data if the values of the input patterns are much greater than 4.

3. Neural network theory holds that the delta rule offers a significant improvement over the Hebb rule for linearly separable data. Based on this fact, it is advisable to use it as a benchmark case over other algorithms. However, given the simplicity of the Hebb rule, and the fact that for certain cases (such as the one presented by Dr. Samarasinghe and worked on in this article) it performs better than the delta, it is suggested to use it for comparison purposes.

[1] Samarasinghe, S. *Neural Networks for Applied Sciences and Engineering. From Fundamentals to Complex Pattern Recognition*, Auerbach Publications, pp.35-40, 2007.