Done in 7 Days? Let the Neural Net Decide

Using AI Neural Network and JAVA to Predict Story Completion Times

"The best way to predict the future is to invent it."

Alan Kay

We all work agile – or at least are familiar with it. We all have that moment at the start of the sprint when we do the estimation of the tasks and, based on that, decide what to be picked up in the next sprint or not.

And we all had - during the sprint or at the end of it - moments when we have realized that some stories /tasks were underestimated mostly due to our optimistic enthusiasm…

How about letting the machine step in? Wouldn't be interesting to have in Jira for each story a textbox where, after team estimation to see also *an AI prediction based on historical data?*

This article is about a proof of concept on training and using a neural network to predict if a git hub task/story will be completed in a certain amount of time.

For accomplishing that we will have to query github repository for a list of tasks with the status "DONE", select the relevant fields from their internals (*titleLength, commentsFirstDay, hasCodeBlock, isCollaborator, labelBug*) and use them to train a Logistic Regression Neural Network.

While doing that will try to explain some interesting core AI concepts like what is a Neural Network, what are its core internals and how the learning process takes place.

So, without further ado, let's start with describing the magic world of Neural Networks

Logistic Regression – The Learning Intuition

"Everything should be made as simple as possible, but not simpler."

Albert Einstein

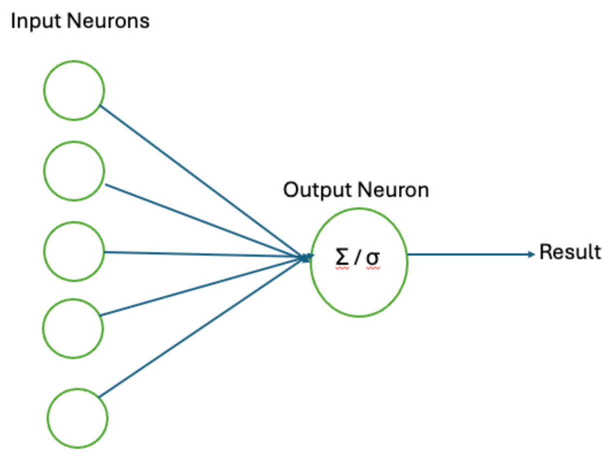
Let’s start by having a look on how the human brain works. Billions of neurons bounded together, all communicating via electrical impulses to achieve a common goal.

Basically, one neuron collects all electrical impulses from the neurons connected to it, then it process them internally in order to prepare its own electrical impulse, which is then propagated further.

Although far less complex than the human brain, a **neural network** is a mathematical model that processes numbers instead of impulses.

The most fitting model for our problem is a **Logistic Regression binary classifier** - having in mind the result of our prediction being 1 if the task will be finished in 7 days and 0 if not - which is the simplest of them all because it has minimum processing (only input and output layers).

We will design our neural network starting from the tasks / stories internals we decided to use for our prediction.



The 5 input neurons are:

- *titleLength*
- *commentsFirstDay*
- *hasCodeBlock*
- *isCollaborator*
- *labelBug*

What we want to achieve is to have a model that can predict if a jira item can be closed in 7 days based on the historical data, using the above parameters.

For that we use as a training set a sufficiently large set of task/story examples — the more, the better of task/story items, each with the above fields.

Now remember about the human brain we discussed earlier. Well, in the neural network the information is gathered at each neuron level, converting using activation function and then passed along. Each neuron has a weight (w). The model also used a bias (b) value

The prediction ($\hat{y}$) formula for each training set element is:

$$\hat{y} = \sigma(W^T X + b), \text{ where}$$

$$W^T X = \sum_{i=0}^{n} w(i)x(i)$$

{MATH ASIDE}

- **W** is a set of weights that tell the model how important each input feature is. How? by training process - we will get back to this later.
- **X** is the input vector.
- **σ(...)** is the sigmoid/activation function, which squashes the result into a value between 0 and 1 — giving us a probability.
- W and X have the same size – they are reffering to the input data used (in our case set of *titleLength, commentsFirstDay, hasCodeBlock, isCollaborator, labelBug* for each task or story we use for training.
- **b** is the bias — think of it like a starting point or offset that shifts the prediction.Think of bias as a thermostat in a room. It sets the baseline temperature regardless how many people enter or leave.Similarly, bias shifts the prediction baseline before considering the input features.

So the model takes the input, *weighs it*, adds some bias, and then passes it through a probability lens. This lens — the sigmoid — allows us to interpret the model's output as a probability of success.

If the output ŷ is close to 1, the model believes the task will likely complete in 7 days. Closer to 0? Not so likely.

Let’s explain a bit more this core concept of activation function.

The activation function decides how strongly a neuron should "fire." In logistic regression, we use the **sigmoid function**, which turns any input into a value between 0 and 1 — like a smooth on/off switch. It helps us interpret the output as a probability.

The sigmoid function formula is

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

…where e represents Euler’s number, a mathematical constant approximately equal to **2.718**.

We can see that each input of z >1 will be converted by activation function to a number between 0 and 1, closed to 0 meaning the probability to “probably never occurs” and 1 – “likely to happen”.

{MATH ASIDE}

Think of the sigmoid as a smooth ramp — no matter how far you push up or down the input, the output always levels off gently between 0 and 1, like a volume knob that never quite hits silence or maximum but smoothly adjusts the sound.

Having this calculated for each neuron - we have already completed a good part of what theory defines as “FORWARD PROPAGATION” .

With that in mind, fasten your seat belts as we are getting closer to the core learning mechanism of Neural Network model.

Inside the AI Mind: Learning Process Demystified

"Paths are made by walking."

Kafka (Franz, not Apache!!!)

Each element of training set has a vector of input parameters X :[x1…xn] and a result output (y)

We know for instance what are the elements of each story/task used in our internal set and we also know if it has been finished in 7 days (y=1) or not (y=0).

Each node of the neural network and also the bias will be initialized with a default value.In case of Logistic Regression it can be 0 as well.Then, starting with the training set, we calculate the prediction ($\hat{y}$) using the previously explained formula $\hat{y} = \sigma(W^T X + b)$ and we determined you far are we from expected (y)

This is done by computing the *Loss Function:*

$$Loss(y, \hat{y}) = -[y\log(\hat{y})+(1-y)\log(1-\hat{y})]$$

…which, taking into account that (y) can either be 0 or 1 – it can be simplified as follows:

If y = 1 and *$\hat{y} \approx 0$, then the function value being $-\log(\hat{y})$ → very large loss*

If y = 0 and *$\hat{y} \approx 1$, then the function value being $-\log(1-\hat{y})$ → also very large loss*

To add more clarity, let's run few examples:

If y = 1 but $\hat{y}$ = 0.01, loss = -log(0.01) ≈ 4.6 → BAD

If y = 1 and $\hat{y}$ = 0.99, loss = -log(0.99) ≈ 0.01 →GOOD

If y = 0 but $\hat{y}$ = 0.99, loss = -log(1 - 0.99) = -log(0.01) → 4.6 BAD AGAIN!

That is a very concise explanation of why this function works. The model is "punished" when it's confident but wrong. Why? Because the loss function is logarithmic:

$$Loss = \begin{cases} -\log(\hat{y}) \ if \ y = 1 \\ -\log(1-\hat{y}) \ if \ y = 0 \end{cases}$$

The closer $\hat{y}$ is to the **wrong class**, the closer the loss gets to **infinity.**

{MATH ASIDE}

The loss function is a kind of 'error alarm.' If your prediction is wrong, the alarm sounds loudly to tell you to adjust your steps. If it's right, the alarm is quiet. The goal is to silence the alarm as much as possible.

The **Cost function** simply extends the **Loss function** by averaging it across the entire training set:

$$J(W,b) = (\frac{1}{n}) \sum_{i=1}^{n} (L(y(i), \hat{y}(i)))$$

{MATH ASIDE}

It looks complicated, but is not. Actually while the loss function is applied to individual samples, the cost aggregates as average the losses for the entire training set.

Think of the **loss** as a student’s grade on a single test.

The **cost** is then **average grade** across all the tests.

Now the stage is set for BACKWARD PROPAGATION. All we can do is to train the neural network model by updating repeatetly the weight of each neuron and the bias until for each element of the training set we will get a prediction acceptably closed to the expected (*ŷ ≈ y).* How? With something called GRADIENT DESCENT.

Gradient Descent – How the Model Learns

“You can't connect the dots looking forward; you can only connect them looking backward."   Steve Jobs

Gradient descent is an optimization algorithm used to minimize the cost function by iteratively adjusting the model’s weights and bias

Iteratively over training set:

- update weights

  W: W – αdJ(W)/dW

- and also bias:

  b: b – αdJ(b)/db

- then recalculate *ŷ, loss and cost functions again*

… until for all training set elements *ŷ ≈ y.*

*Here the gradient descent values are closed to 0. At this point, we say the model has **converged** — it has learned as much as it can from the data.*

α = learning rate parameter (controls the step size, usually set to 0,1 or 0,01)

dJ(W)/dW – partial derivative of the cost with respect to weight

dJ(b)/db – partial derivative of the cost with respect to bias

{MATH ASIDE}

Imagine you're standing on a hilltop in thick fog, blindfolded. You can't see the bottom, but you can *feel* the slope under your feet. You want to reach the lowest point — the valley — but you have to move step by step.

This is **Gradient Descent.** At each step:

- You measure how steep the slope is (compute the gradient),
- Take a step *downhill* (update your weights in the negative gradient direction),
- Repeat — until the ground feels flat (the gradient is close to zero)

The **Learning Rate** α is a *hyperparameter* that controls how big the learning steps are. If it’s too small, learning is painfully slow. If it’s too big, the model might overshoot the minimum or even diverge.

Think of it like adjusting the size of your blindfolded steps down a hill — too small and you crawl, too big and you trip.

… and that’s the so-called 'complicated' process behind how neural networks actually learn — one careful, blindfolded step at a time.