# Least Surprised Algorithm

by Sven Nilsen, 2017

*Here is an idea for predictive machine learning. Instead of optimizing for a utility function, this algorithm optimizes for less surprises. It does so relative to a capability of recognizing patterns.*

The general idea is to steal chunks of information from the environment and synthesize new experiences. When the environment generates new input, the synthesized experiences are compared to the input to see whether it is surprising or not. An input is least surprising when the algorithm could not think of anything else. By synthesizing anticipated input, the algorithm can not cheat by inconsistency.

For example, by picking a random number between 0 and 1, the algorithm outputs a character that it might see next. Repeating this process many times increases the chance that one of the characters guessed by the algorithm matches the new input character. The algorithm is given N chances to guess the next character. Assuming it guesses one, it can then condition on that choice and guess what it will see next, because it generates its own input in an “imaginary scenario”. This is how it can guess whole words and sentences, simply by repeating the new guess based on earlier guesses.

The probability that the algorithm guesses a character, is how frequently it occurs among N chances to guess the next character, divided by N. This means the algorithm needs to spend computer power to sample the probability, which gets more accurate by using more computer power. This has the advantage that the probability is consistent with the actual guessing, instead of assuming infinite computing power which requires proof. Depending on the specific implementation of the algorithm, the probability can be indeterministic or deterministic, inaccurate or accurate. If the algorithm is given no time to think, then it makes no guesses, so that probability that it guesses the next character is 0%.

A particular version of the Least Surprised Algorithm is one that uses rule-based guesses. For example, one character `i` is followed by another character `s` (e.g. the word `is`). There can be multiple rules guessing the same character. To make guesses, the rules are given a weight, such that a more successful rule gets more guesses than an unsuccessful rule. A rule can be non-deterministic or deterministic, even trained by a neural network to self-improve over time. It can also pick a character randomly.

To balance the rules properly, one must do the following:

1. All rules that make correct guesses share 1 point in increased `points`
2. All rules have a counter `used` that tracks how many times it has guessed before

This prevents identical rules from outnumbering the others, and it makes sure that making guesses in many situations with less success is not rewarded.

The weight of a rule is then calculated (which is invariant for linear values of `used`):

$$\text{weight}[i] = \text{rule}[i].\text{points} / \text{rule}[i].\text{used} / \sum j \{ \text{rule}[j].\text{points} / \text{rule}[j].\text{used} \}$$

This number is used to pick out a rule that makes a guess for the next character. Repeating this N times, one gets a sample of the probability of guessing the character.