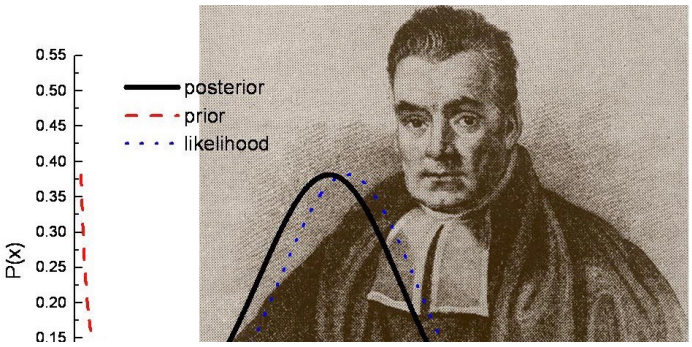
Andrew Greatorex Follow
Machine Learning and Poetry.
Mar 11, 2017 · 5 min read



Feature Engineering: Bayesian Methods for Binning

One of the most crucial pieces of any data science puzzle is perhaps also the least glamorous: feature engineering. It can be protracted and frustrating, but if it's not done right, it can spell disaster for any modelling or analysis that follows. In this post, I hope to shed some light on a delightful inference technique that can be used.

> *"Applied machine learning" is basically feature engineering—Andrew Ng*

It's not an uncommon tactic to 'bucket' together data points within a feature. Depending on how you want a feature bucketed, it can range from the simple to the laborious. Consider the case where you want to one hot encode a feature with a large number of distinct values. Bucketing based on each distinct value may leave you cursed with a large feature space and future models susceptible to overfitting.

So how to bucket then? Several rules of thumb have been proposed, including *Scott's Rule*, *Knuth's Rule* and the *Freedman-Diaconis Rule*. The issue with these methods is that they all assume the same size of bins, which is problematic and not necessarily optimal.

Time to bust some myths. First, that data points must be binned to make sense of them. Second, that bins must be of an equal size. Third, that bins must be large enough so each has a 'statistically significant' sample size.

Many people are stuck in the mindset that the best way to reduce noise is to smooth their data, and that the best way to deal with data points is to bucket that data up. Such approaches to binning are problematic because they discard information and introduce dependence on the following parameters: (1) degree of smoothing and (2) bin size—and specifying parameters is bad. Ideally, the bin width should be

dependent only on the properties of the data itself, and the resulting histogram should not be fixed a priori.

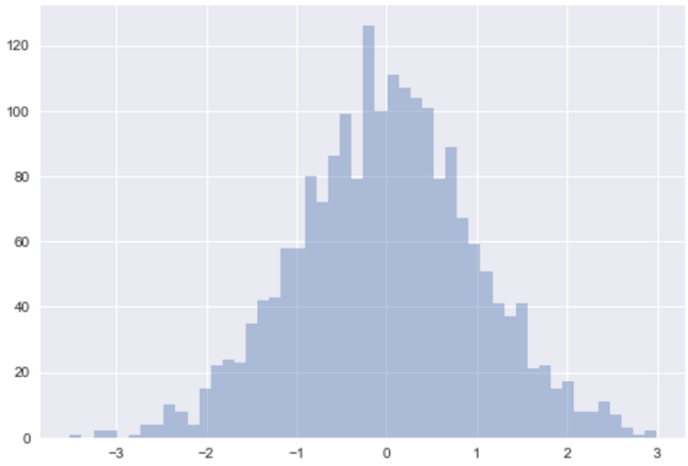
"There must be *something* we can do!" I hear you cry, wide eyed with Einsteinian hair. There is.

. . .

Bayesian Blocks

The Bayesian Block framework is an adaptive method to automate the process of identifying bin sizes and partitioning data. Like other non-parametric methods, Bayesian Blocks seeks a generic representation, avoiding assumptions about smoothness and shape which provide priori limitations.

The optimal segmentation is is a simple yet effective optimisation problem, whereby the goal is to maximise a quantitive expression of the criterion, or alternatively to minimise an error measure. What's more, it's able to compute a global optimum, not just a local one.



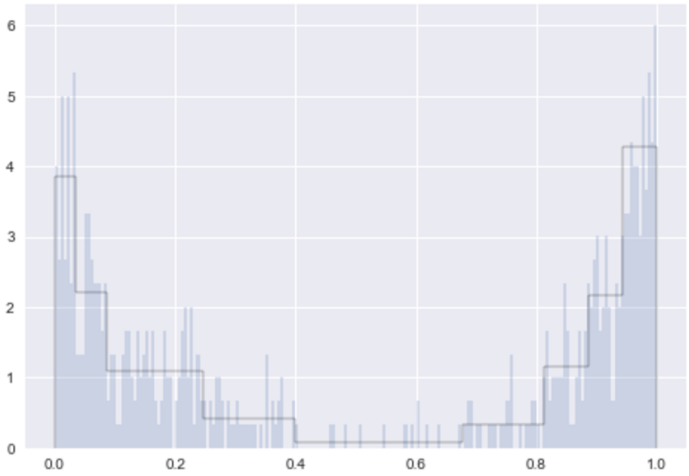
On a high level: given a histogram of an arbitrary but fixed *block* size, like the one shown to the left, we may use a Bayesian likelihood framework which uses a *fitness function* that depends only upon (1) the width of each block and (2) the number of data points within each block. The *best* binning schema will be the block configuration, based on the edges (known as *change-points*, which will vary) between the blocks, that has the maximum fitness.

Note: any attempt at scaling a brute force version of this method would be in vein. The number of possible configurations grows exponentially, by order 2^N. Were it not for the beauty of dynamic programming techniques, such a would be impossible.

To clarify, a change point is a point in a series which undergoes an abrupt transition, by one or more of its parameters jumping instantaneously to a new value. Here, we take the block fitness to be the maximised relevant likelihood with respect to the block heigh

A fitness function is a convenient measure of how well a constant level represents the data within a block, which is also only dependent on the data within that block and not on model parameters other than the ones specifying block edge locations. There are different ways of defining the fitness function, whose merit will vary based on what is most appropriate for the data at hand.

What follows is essentially mathematical induction. First, we order our data. Next, we begin with the first data cell, and iterate. At each step one more cell is added until all data points have been analysed (for optimisation there is a trigger described in the paper linked below, halting when the first change-point is detected). At step R, for the first R cells, we identify the *optimal cell partition*, at each step storing the optimal *fitness* (denoted as F(r) in an array called **best**) and the location of the last change-point of the optimal partition (in an array called **last**).



Bayesian Blocks fitted to two concatenated, randomly sampled beta distributions

The optimal partition at cell location R+1 can be computed by first considering, for a given cell location r, the set of all partitions of the first R+1 cells, and using the knowledge that the only member of this set that could possibly be optimal is the optimal partition of the first R, followed by the last block at R+1. The fitness of this partition is the sum of F(r).

$$A(r) = F(r) + \begin{cases} 0 & r = 1 \\ \texttt{best}(r-1), & r = 2, 3, \ldots, R+1 \end{cases} .$$

The fitness of all partitions that can possibly be optimal, over the range r, are now stored. The value of r that yields the most optimal solution is simply computed by finding the maximum A(r).

After all points have been iterated over, we now need to fetch the locations of the change-points for the optimal partition. For this, we use the last value in the **last** array to determine the last change-point of the optimal partition. We 'peel off' the section corresponding to the last block and repeat.

And that's pretty much it.

. . .

Thanks for stopping by. For more information about Bayesian blocks, be sure to check out the original paper by Jeffrey D. Scargle and the cool post by Jake Vanderplas.

Studies in Astronomical Time Series Analysis. VI. Bayesian Block Representations—Jeffrey D. Scargle

Dynamic Programming in Python: Bayesian Blocks—Jake Vanderplas