Machine Learning

Ashod Khederlarian

Contents

1 Local Linear Regression

Local linear regression (LLR), as the words suggest, is basically linear regression but done locally. For a given point, you fit a linear model that best predicts (i.e. minimizes some cost function) the outcome at that point and some surrounding points. Sometimes it is also called weighted linear regression, because you assign a weight to every point and its neighbors. This weight should be decreasing as you go further from the point, otherwise it wouldn't be local. A suggestive but incorrect graph is shown below.

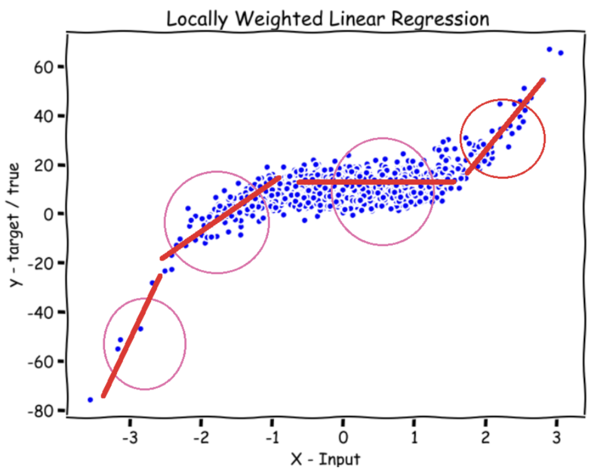

Figure 1: Sort of how LLR works, but you actually have a different straight line for each point.

It is incorrect because one actually fits a different linear model for each point, so in the above figure each point would have its own straight line. To make all of this more concrete, suppose your features are represented by an $N \times m_x$ matrix $\mathbf{x}$, where N is the number of available data points, and m_x is the number of features for each point, and also suppose that your outcomes are represented by an $N \times m_y$ matrix $\mathbf{y}$, with m_y being the number of outcomes for each point. So really when I use the word "data point", it is often a vector. To predict the outcomes for a point x_i, one uses a linear model

$$y_i = b_i + \mathbf{a}_i x_i. \tag{1}$$

x_i and y_i can in general be vectors representing a certain number of features and a certain number of outcomes, so $\mathbf{a}_i$ is a matrix. It is convenient to absorb the y-intercept term b_i into the matrix $\mathbf{a}_i$, and in doing so a column of ones should be added to the feature matrix $\mathbf{x}$. The new matrix of parameters is often represented by $\boldsymbol{\theta}$, and the above equation is re-written as

$$y_i = \boldsymbol{\theta}_i^T x_i, \tag{2}$$

with $\boldsymbol{\theta}_i$ being a matrix of dimension $(m_x + 1, m_y)$. Notice that the parameter matrix also has an index i because it is local; there is a different parameter matrix for every point x_i. Now this parameter matrix is obtained by minimizing a cost function, which often is the squared differences (now weighted)

$$C_i = \sum_{j \in NN} w_j (y_j - \boldsymbol{\theta}_i^T x_j)^2. \tag{3}$$

NN is the set of nearest neighbors. One can include all the points and let the weight deal with far neighbors, but since they shouldn't contribute much it is numerically favorable to only include a certain number of nearest neighbors. If one is dealing with a training set and just wants to fit that, then it is better to include the point x_i in the set NN and give it the largest weight. However, if the goal is to predict the outcome for a new point, then this is not possible, because the outcome is not known a priori. To find out who the nearest neighbors are, one needs a measure of "distance", and from what I've seen so far, mostly the Euclidean distance between two vectors x_i and x_j is used. This distance is also used to determine the weights, like inverse distance/distance squared[1], or exponentially decaying with distance.

Minimizing the cost function with respect to the parameters means setting the derivatives of C_i to zero. To do this, first I will rewrite it as

$$C_i = \sum_j w_j \sum_l (y_{j,l} - \theta_{i,ml} x_{j,m})^2 \tag{4}$$

then minimizing this is equivalent to

$$\frac{\partial C_i}{\partial \theta_{i,l'm'}} = \sum_j w_j 2(y_{j,l'} - \theta_{i,m'l'} x_{j,m'}) x_{j,m'} = 0 \tag{5}$$

This can be written in matrix form as

$$\boldsymbol{\theta} = (\mathbf{X}^T \mathbf{W} \mathbf{X})^{-1} (\mathbf{X}^T \mathbf{W} \mathbf{Y}) \tag{6}$$

with $\mathbf{X}$ being the matrix of features, $\mathbf{W}$ a diagonal matrix of the weights, and $\mathbf{Y}$ the matrix of outcomes. These are limited in size by the number of nearest neighbors, so the above matrix multiplication doesn't take that long. The most demanding thing numerically is finding the nearest neighbors, and that can be done rather quickly using k-d trees (see appendix). Below is a simple illustration of how things look like for a trivial case. I just generated straight-line data and put random Gaussian errors on the outcomes.

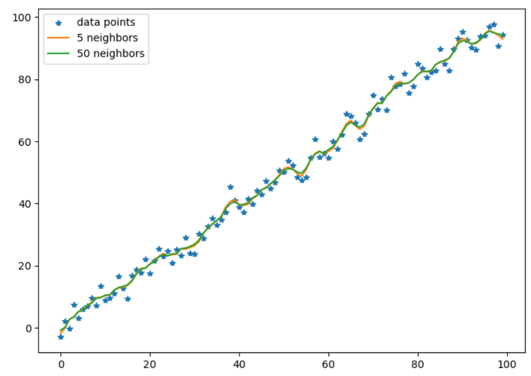

Figure 2: Illustrating LLR on some linear data with random Gaussian errors on the outcomes. This shows how the resulting function is nonlinear and why the first figure gives a false impression.

[1]The point x_i has zero distance with itself, so if it's included in NN, then it must be given an appropriate weight.

A K-d Tree

The k-d nearest neighbors algorithm is simple to understand but a bit difficult to write. For simplicity, suppose the data is d=2 dimensional, and label the features by (x_1, x_2). One of the most efficient ways of finding nearest neighbors is to first reorganize the data in a k-d tree, where each node is a condition that separates the data accordingly.

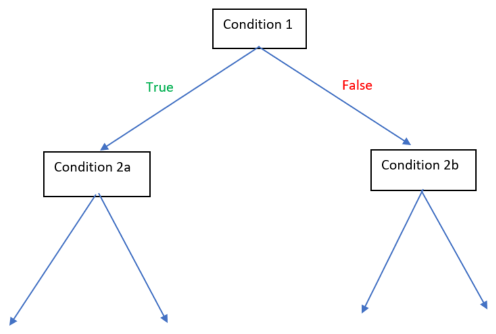

Figure 3: The basic idea of a k-d tree.

To use this for nearest neighbors, the conditions should be spatial. This way, each condition separates the data points by a line (or hyperplane in higher dimensions), and so to find the nearest neighbors of a given point, one only has to go down the tree and see where one ends up. The conditions should also alternate between the two dimensions. So, for the top node, a possible condition is $x_{1i} > \text{median}(x_1)$, which separates the data into left and right sections along x_1. After doing this, the new conditions should separate the left data into up and down, i.e. along x2, and the same for the right data. This is illustrated in the figure below.

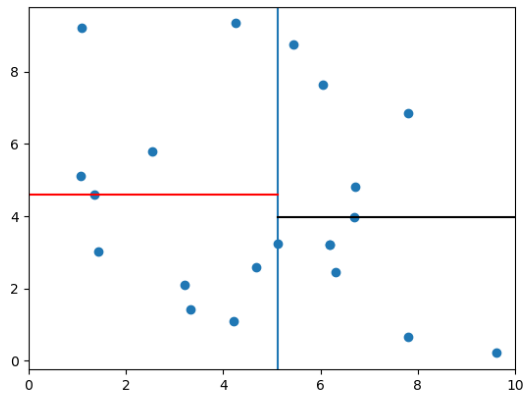

Figure 4: A bunch of random points and how they are separated by the first few conditions in a tree. The blue line would represent the top node, and the red and black ones would represent the 2nd and 3rd nodes.