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.

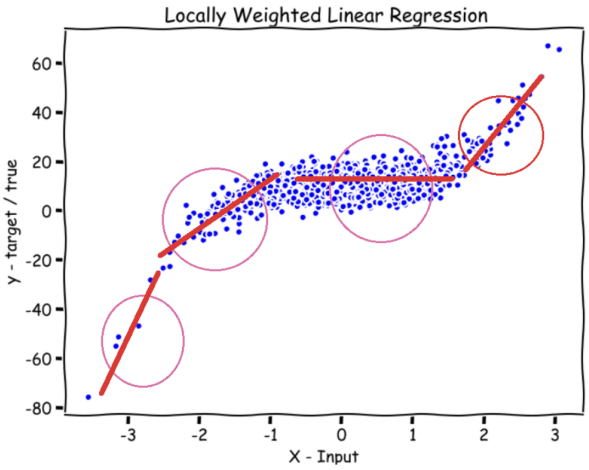

Figure 1: Sort of how LLR works, but you actually have a different straight line for each point.

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.

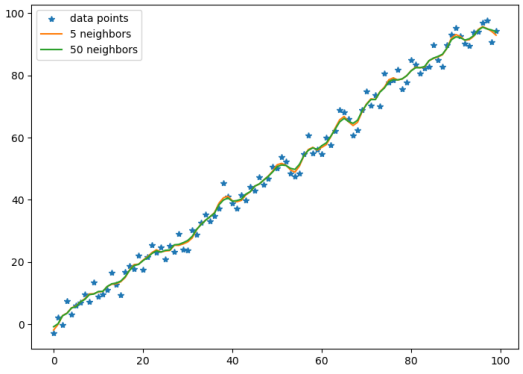

Figure 2: Illustrating LLR on some linear data with random Gaussian errors on the outcomes. This shows how the resulting function is nonlinear and why the first figure gives a false impression.

[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.

2 Decision Trees

Decision trees can either be classification (discrete) or regression (continuous) trees. The simplest starting point is a single-node classification tree that just gives you a yes or no answer.

Imagine having data of patient tumor size and whether they have cancer or not. Let's say the data says that if tumor size is less than $10cm^3$, then the tumor is benign (not cancerous), and if it's greater than that then it is cancerous. The easiest way of modeling this data is to setup a "tree" with the top decision node having the condition $x < 10cm^3$ and two outgoing branches that are end nodes (leaves) which give a yes or no answer depending on whether the condition is true or false.

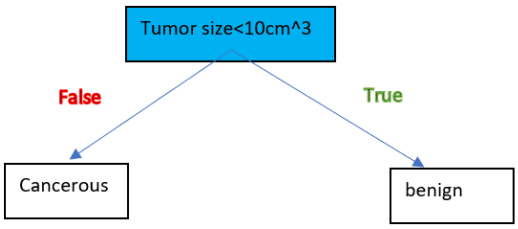

Figure 3: Basic classification tree.

This is called a classification tree because the outcome is a yes/no answer. If the outcome was continuous, it would be a regression tree. The "fitting" part is deciding the conditions on every condition node. This can be accomplished in a variety of ways, the simplest one being testing different conditions and seeing which one minimizes your cost function.

If there's more than one feature, then you can have different condition nodes for different features along the tree. I don't know how this is exactly done numerically in the most efficient way (more below). One can always use brute-force and just try different configurations of trees and compare cost functions, but this can be extremely inefficient, especially with large trees.

2.1 Information Gain

One good way of deciding the condition on a condition node is to use the idea of Information gain. This is done through defining Shannon's entropy at each node as

$$S = -\sum_i p_i \log p_i, \tag{7}$$

where i goes over the classes (or outcomes) and p_i is the probability of a random data point from that set (i.e. the set corresponding to the node) belonging to class i. Then, the optimal condition is the one that reduces the entropy most - or gains the most information, since $Info = -S$

$$IG = I_{children} - I_{parent} = S_{parent} - \sum_i w_i S_i, \tag{8}$$

where i goes over the children (2 in this case) and w_i is the weight associated with each child, given by the relative number of data points the child inherited. The tested conditions are basically all the ones possible with the given features (so one tests within one feature dimension all the possibilities, and also repeat for all other feature dimensions).

This makes sense because ideally one would want to find the tree that best classifies the data set. This will be done if at a leaf the probability of a certain class is maximized (best case scenario equal to 1), so

then a sample point that has the features of that leaf will be classified correctly with a high probability. The closer a single class probability goes to one and the rest go to zero, the lower the entropy will be, $S = 0$ in the ideal case.

Another way of calculating information gain is through defining the Gini impurity of a class, which is a measure of how often a randomly selected data point would be mislabeled if it was randomly labeled according to the distribution of classes. This is given by

$$G = p_i \sum_{k \neq i} p_k = p_i(1 - p_i). \tag{9}$$

Multiplying probabilities means an "and", so the above just gives the probability of the point belonging to class i and the probability of it being mislabeled. This avoids using a logarithm when compared to Shannon's entropy. To find the Gini impurity for the whole set, one has to add the above over all the classes, so $G_{set} = \sum_i p_i(1 - p_i) = 1 - \sum_i p_i^2$. The information gained is given by $IG = G_{parent} - \sum_i w_i G_i$, where i goes over the children.

This would clearly work for classification problems, but I'm not sure how it would work for regression. Maybe it's best to minimize square differences in that case. So, take a feature, and split the data according to that feature. For each child node, set the outcome as the average value of the y's in that node, and then calculate the square differences. Assign a total "square difference" to that split by taking the weighted average of the square differences of each node, like the weighted average of the entropy. Repeat over all the features and pick the decision that has the lowest square difference. To decide if the child nodes are going to become leaves or decision nodes, have a threshold for least square, or something of that sort.

2.2 Random Forest

A random forest is just a bunch of random decision trees giving different outcomes, and the final outcome is the average of all of them.

3 Gradient Boost

Gradient boost starts with one decision tree (usually just a leaf that's the average value of the outcomes), and then gradually adds other trees by predicting the residuals obtained from the previous tree. In this way, the new tree "fixes the mistakes" of the old one. Usually a learning rate is used to scale the contribution of the new tree, to avoid overfitting. Trees are added until a maximum specified tree count is achieved or adding trees no longer improves the outcome drastically (in this case, doesn't reduce the residuals). Here are the rigorous steps for gradient boosting, given an initial data set {(x,y)} and a differentiable loss function $L(y_i, F(x_i))$:

- Initialize the model by a leaf that has the outcome $F_0(x) = \arg\min_\gamma \sum_i^n L(y_i, \gamma)$, which basically means the value γ that minimizes the sum of the loss functions. In the case where L is difference squared, this can be solved analytically and γ would be the average of the outcomes.
- Start building the next tree by calculating the generalized residuals, $r_{im} = \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}$, where here $F(x)$ is the prediction from the previous tree. Again, for squared differences, this gives just the residual. m represents the m-th tree, so that r_{im} is the matrix of residuals for the m-th tree.
- Build a regression tree to predict the residuals r_{im}. Label the leaves of this tree as R_{jm}, where again m is the tree and j indexes the leaves.
- Calculate the output value of the leaves by computing $\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$.
- Make the new prediction $F_m(x) = F_{m-1}(x) + \lambda \gamma_{jm} (x \in R_{jm})$.
- repeat from step 2 until enough trees are made.

4 Neural Networks

4.1 Logistic Regression

Logistic regression is used to predict a yes or no outcome with a probability. To do this, a Sigmoid function is used, shown below. It has asymptotic values of 0 and 1 at the limits and it is $1/2$ for $z = 0$. This is typically called a hypothesis function and is denoted by h.

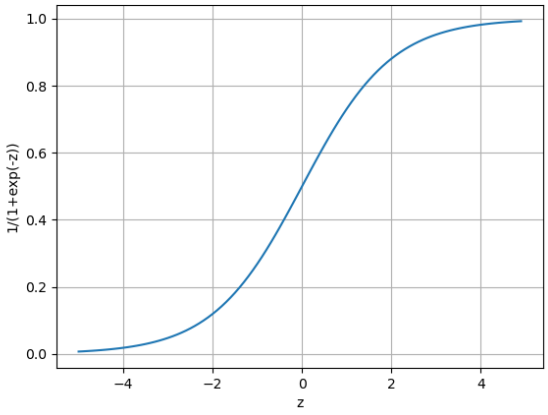

Figure 4: The Sigmoid function.

More properly, when predicting an outcome from a set of features, the argument of the hypothesis takes on the value of $\theta^T x$, where θ is a vector of parameters and x is a vector of features (this is not linear, so x can mean $x^2, x^5, ...$ etc.). So, given a data point with feature vectors x_i, the model predicts

$$h_\theta(x_i) = \frac{1}{1 + e^{-\theta^T x}}. \tag{10}$$

Using the squared difference as a cost function for this model would not work because it would be a non-convex function. This is a problem because then there might be many minima, and gradient decent wouldn't work well. The obvious solution to this is to use an alternative cost function (L), defined below

$$L_i(h_\theta(x_i), y_i) = -y_i \log(h_\theta(x_i)) - (y_i - 1) \log(1 - h_\theta(x_i)). \tag{11}$$

Here the outcomes y_i are either 0 or 1, so the cost function for each case is shown below.

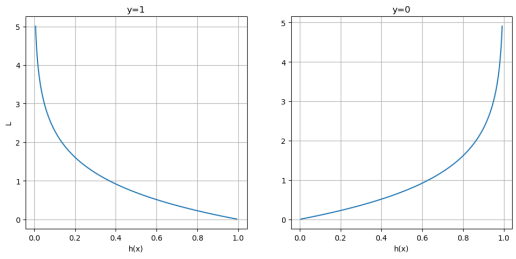

Figure 5: Logistic regression cost function.

To fit the model, the sum of the cost functions is minimized with respect to θ by gradient descent. Denote the total cost function by $J(\theta) = \sum_i L_i$. Gradient descent updates the values of θ iteratively to reach the minimum

$$\theta_j = \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}, \tag{12}$$

where the small index j denotes the j-th parameter in the vector θ, and α is a learning rate. With the above cost function, the derivative turns out to be

$$\frac{\partial J(\theta)}{\partial \theta_j} = \sum_i (h_\theta(x^i) - y^i) x_j^i. \tag{13}$$

The last x has an index j because the derivative picks out the feature multiplying θ_j.

4.2 Neural Networks

Neural networks are used for highly non-linear modeling. Just doing logistic regression is not feasible when the number of features is large and one has to make non-linear combinations of them. A single neuron only does logistic regression, and is represented by the below figure

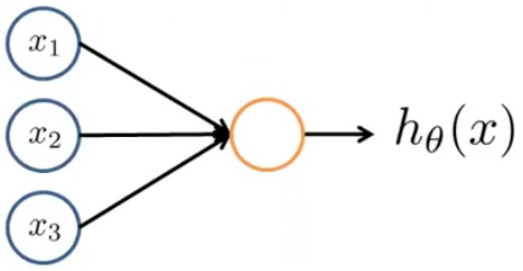

Figure 6: A single neuron represents logistic regression.

A neural network is a combination of neurons, as shown below. The first layer is called the input layer, the final layer is called the output layer, and the layers in between are called hidden layers. a_i^j represents the "activation" of unit i in layer j, and the whole network is parametrized by matrices of weights (parameters) Θ, with Θ^j corresponding to the weights that map layer j to layer $j+1$. The computations represented by this network is shown below it (g is the Sigmoid function).

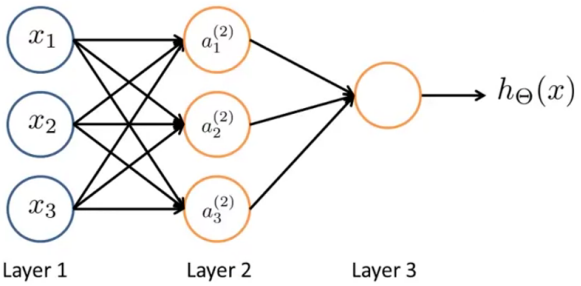

Figure 7: A simple neural network.

$$
\begin{aligned}
a_1^{(2)} &= g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \\
a_2^{(2)} &= g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \\
a_3^{(2)} &= g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3) \\
h_\Theta(x) = a_1^{(3)} &= g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})
\end{aligned}
$$

Since layer 3 uses the activations of layer 2 as features, I like thinking about the hidden layers of a neural network as learning what are the best feature combinations to use. With logistic regression, I have to think about which feature combinations would be helpful, e.g. $x_1^2 x_2^3, x_1 x_4^2$, etc., and there is no way of knowing which ones are the best besides testing them out. A neural network chooses the features by learning them through the hidden layers.

4.3 Training the Neural Network: Backpropagation

The cost function for a neural network will be a generalization of the cost function for logistic regression, because the output can be more than 1 (denote it by K)

$$
J(\Theta) = -\sum_{i}^{m} \sum_{k=1}^{K} y_k^i \log(h_\theta(x^i)_k) + (1 - y_k^i) \log(1 - (h_\theta(x^i))_k) + \lambda \sum_{l=1}^{L} \sum_{i}^{s_l} \sum_{j}^{s_{l+1}} (\Theta_{ji}^l)^2. \tag{14}
$$

To proceed with gradient descent, one needs to calculate the derivatives $\frac{\partial J(\Theta)}{\partial \Theta_{mn}^l}$. This is straightforward if l is the last layer. First, remembering that

$$
h_\theta(x)_k = g(\theta_{ki}^L a_i^L) = \frac{1}{1 + e^{-\theta_{ki}^L a_i^L}}, \tag{15}
$$

then the derivative of this is straightforward

$$
\frac{\partial g(\theta_{ki}^L a_i^L)}{\partial \theta_{mn}^L} = \frac{a_n^L e^{-\theta_{ki}^L a_i^L}}{(1 + e^{-\theta_{ki}^L a_i^L})^2} \delta_{m,k}. \tag{16}
$$

So, using this to evaluate the derivative, one ends up with

$$
\frac{\partial J(\Theta)}{\partial \Theta_{mn}^L} = a_n^L (h_\theta(x)_m - y_m) + \lambda \Theta_{mn}^L. \tag{17}
$$

Note that to do this, one has to start with a data point (x, y), forward propagate to evaluate all the activations and the output $h_\theta(x)$, and then proceed with the calculation. If only a single data point is used to evaluate the gradient and update the parameters, then that is called stochastic gradient descent. If all the data points are used, and the gradients are added and used to update the parameters, then that is called full-batch gradient descent. The in between, where only a subsample of points are used to update the parameters, is called a mini-batch gradient descent. Using tensorflow keras, this is controlled by the parameter batch_size.

The above is only the derivative with respect to the parameters of the last layer. The derivatives for the hidden layers get messy, because they are hidden in the activations (so lots of chain rule!). I will not go over the math, and will just show the algorithm (from Andrew Ng's course) that does that, and it's called backpropagation.

Backpropagation algorithm

Training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j). (used to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$)

For $i = 1$ to m ← $(x^{(i)}, y^{(i)})$.

Set $a^{(1)} = x^{(i)}$

Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$

Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$

Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$

$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ ← i, j

$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$.

$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$ if $j \neq 0$

$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$ if $j = 0$

$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$

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.

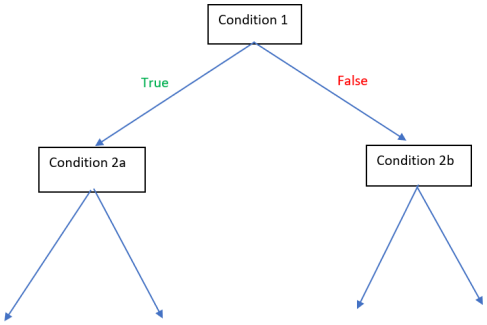

Figure 8: The basic idea of a k-d tree.

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.

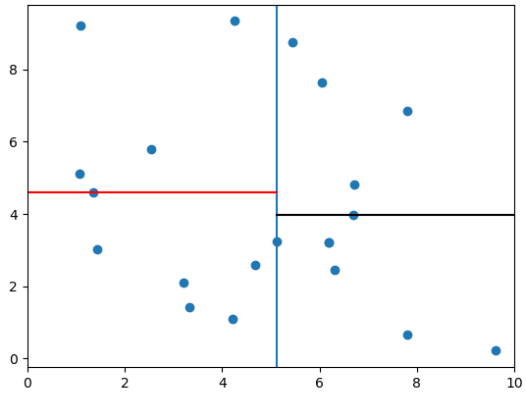

Figure 9: A bunch of random points and how they are separated by the first few conditions in a tree. The blue line would represent the top node, and the red and black ones would represent the 2nd and 3rd nodes.

B Regularization and Ridge Regression

When the number of features is too much, as compared to the number of data points, then it is easy for the model to overfit. One way of fixing this is by regularization: adding a term in the cost function that tends to make the fitting coefficients smaller. This can especially be useful if you have many features contributing a bit to predict the outcome. The term to add is (generally)

$$\lambda \sum_j \theta_j^2, \tag{18}$$

where λ is a regularization parameter and θ_j are the fitting coefficients. This is called L2 regularization because the coefficients are squared. L1 is when, instead of squaring, you just use $|\theta_j|$. I'm sure there are higher orders as well.

Ridge regression is basically regularization in the case of a linear model, though it seems that only the slope is penalized.