JoYo's Awesome Fun-Times Supervised Analytics Platform (of Doom)

Joseph Victor

May 2, 2014

How Might a Real-Life User Want to do Supervised Learning

I speculate that many people want to be able to make supervised real-time predictions on a certain class of not-quite-i.i.d. observations. Examples include: given a user profile, predict if they'd like my product or click my add, or given the profile of a house predict the price it sells for. In real time, both new observations and new requests for predictions are coming in. We want to predict a the price of a house upon query, but we also observe houses being sold, and we want to predict a user's interest in a product, but after presenting the user with the product we observe they click-through. Another (somewhat harder) example: given the time of day and the current state of a cluster, can we predict which nodes will fail next and when (or which nodes have already failed)? Here we are constantly observing the state of a cluster, and we want to predict the state, say, a minute in the future.

We wish to, given a set of observations, which are (x, y) pairs, build a model predict y from x. We do not assume that the (x, y) pairs are i.i.d., but instead that the model changes over time (perhaps according to some stochastic process we don't bother to model). Thus temporally nearby observations will be correlated, and more recent observations will somehow be more relevant to predictions now. However, the massive warehouse of historical data should not be ignored; some aspects of the population don't change (or don't change very fast).

The basic idea is this. The user has new data coming in all the time, which they presumably store in cheap, fast storage for a while before shipping it off to the warehouse. The model then should be the same: have cheap, online, adaptive algorithms looking at the new data coming to try to capture and exploit transient features and build an ever changing model, while training a more robust, expensive, offline model on the historical data to capture the features of the population which remain important for longer periods of time. At a semantics level then the model is perhaps tied to some database tables: perhaps you say at table creation time "I want this table to build a model to predict columns $y_1, ..., y_k$ from columns $x_1, ..., x_p$", and the system trains whenever you do an insert and does a prediction whenever you request one.

What properties does the model need?

The model should be

1. Very black-box (no expertise required, no parameters to manually tune).
2. Very high accuracy.
3. Robust in the face of not-quite-true statistical assumptions.
4. Adding unnecessary predictor columns doesn't make the predictions worse (so you can just throw all your columns in without thinking in you want)
5. Handles missing data gracefully.
6. Allows both numeric and enum columns.
7. Scales to big data, and can be updated without having to refit the entire model.
8. Predictions can be made fast on a cluster.

There is an algorithm called Random Forests, which does all these things awesomely, and every other algorithm I can think of screws up one of these things (especially the first one). Random Forests are known for being one of the most generically accurate machine learning algorithm out there, and they are basically idiot proof, although they're pretty trivial to break with synthetic datasets that exhibit behavior that most real-life applications don't have.

A very quick rundown of random forests

A random forest is made of trees. A tree, also known as CART, is a very simple algorithm that goes like this. Pick a predictor variable x_i and partition your dataset into two based on the value of x_i. Then, for each partition, pick a another variable and go again. this picture should make it abundantly clear what the final model looks like (though not quite how to find it necessarily, but it doesn't matter).

CART is cute and has some nice properties (it is not very biased), but has very high variance (that is, if the observations were slightly different, the tree would be quite different). Accuracy will suffer. However, there is a miracle. If we throw some randomness into the way we build a CART tree (the specifics aren't important, but randomize the picking of the splitting variables and which observations we train on), and we build two trees, those trees will be approximately uncorrelated! That is, on average they will predict the same thing (the right answer, since because unbiased), but they're still two different predictions. What happens when you average two uncorrelated estimators with the same mean? The variance goes down! In fact, the variance will go down more than it went up by adding the randomness to the fitting (and adding the randomness made the trees faster to grow)! Thus we can grow a hundred trees, or a thousand trees, and just average the results! This algorithm routinely wins prediction competitions on kaggle, because it basically gets as accurate as you want.

The reason this is so great is because with most algorithms, there are tons of parameters that require expertise to tune (that is, problems arise if the parameters are two high or two low). With trees, the only real parameter is the number of trees, and there is no expertise required. If you have twice as many trees, although it takes twice as much time and memory, the predictions don't suffer! So you just set the number of trees to as high as you can afford and not worry about the consequences!

Other reasons this is great, you can parallels the building of such a forest (obviously), store a few trees on each leaf node. No need for replication, if you lose a tree it totally doesn't matter, and if a node is non-responsive when the prediction is requested you just don't use that part of the model.

Another huge advantage of random forests is you have an unbiased estimator of how well each individual tree will do in later predictions (this does not give an estimator of how well the forest will do since deviation doesn't commute with averages).

Online Random Forests

There is a version of online random forests (I prototyped it, it seems to work) where each time a new observation comes in, you update some of the trees in a fast, cheap way. We create new trees throughout the day, and as you go, you keep track of how well various individual online trees are doing, and if they are doing poorly, you chop-em down (delete em). Thus you have a constantly evolving forest that should be able to track transient ripples. But we can do better, since recall that we have the offline forest trained on historical data to fall back on. Thus we don't have to worry too much about aggressively chopping down trees when they become obsolete. Further still, we create new trees more often during periods of rapid population change (which we can track by tracking the performance of the online trees versus the offline ones). There are a constant number of parameters governing this, so they can be trivially learned as we go.

Offline Random Forest

The online predictions are (weighted) averaged with an online model, which is partially trained, say, once a day (or week, or hour, depending on how often is required). It is probably impractical to train a random forest on a bajillion datapoints, so instead what we'd do is just delete a random subset (possibly weighted by there error from the most recent period) of the trees in the old forest and grow new ones using a mix of the data newly added to the warehouse and some old data. We can take some more time to do this to make sure the trees are really good, and use that ensemble for prediction.