Meta-algorithms to improve performance.

David G. Khachatrian

September 16, 2019

1 Sampling Methods.

In a learning problem, we normally assume that we have an i.i.d. sample $X \sim P$ of n data points from a presumably unknown distribution P. We then learn a model $h(x)$ using a loss function $L(x) = \sum_i L(x_i)$.

What factors can affect the final model we obtain? Some include:

1. The topology of the loss function L. If the loss function is not convex, there is no unique "best" classifier. This problem gets worse the more "bumpy" our loss function is. If the loss function looks "bumpy", then
2. The sample X. If P is unknown, X is our best representation of P. S
3. The hypothesis space H. If many different $h_i \in H$ minimize L equally as well, *the initial conditions of h* play an important role as to the final performance of the classifier. This could mean h_i gravely misclassifies certain samples or that it's highly sensitive to small changes (i.e. it has high variance depending on the random sample X).

In meta-algorithms meant to improve performance, we consider some ways of how to deal with these different issues.

1.1 Ensembling: Finding different "best" classifiers.

One issue was having several different classifiers work equally well on the same data. This might be because they're picking out different important factors in the data, especially if we know that L and H are complex. So, why not train a few of them and have them vote?

In ensembling, we train each of m copies of h with *different* initializations on the *same* sample X with the *same* loss function L. The final model is an aggregation $h = f(h_1, \cdots, h_m)$ (for example, a simple summation).

This is a simple way of making sure a model isn't ignoring features that may be important in "the real world" but weren't necessary while training (because the training set had some other feature that worked "well enough").

1.2 Bootstrap aggregating ("bagging"): Reducing variance.

Let's say instead our problem is that the variance of our model is too high (for example, if we change one datapoint in X slightly, the whole model's parameters/prediction behavior changes).

In bagging, we train each of m copies of h with *the same* initialization on *different* samples $X_1, \cdots, X_m \overset{iid}{\sim} X$ with the *same* loss function L. The final model is an aggregation $h = f(h_1, \cdots, h_m)$ (for example, a simple summation).

If we assume that each model has an expected bias of 0, then the variance of h's predictions depends on the covariances of $\{h_1, \cdots, h_m\}$: the less correlated the models are, the lower h's variance. Lower variance suggests more robustness to noise, which is desirable in many scenarios.

1.3 Boosting: Fitting weak learners to residuals.

Maybe our concern is that our model isn't complex enough to model the data well — it can do better than chance, but not by much. What if we make a bunch of models, where each new model h_j focuses more on the mistakes that the earlier models $h_1, \cdots, h_{j-1}$ made?

In boosting, we train each of m copies of h with *the same* initialization on the *same* sample X with the *different* loss functions L_i. The final model is an aggregation $h = f(h_1, \cdots, h_m)$ (for example, a simple summation). In this case, the loss function for h_i takes the form of:

$$L_i(y, x) = L(y - f_{i-1}(x; h_1, \cdots, h_{i-1}), x) \tag{1}$$

So h_i is being fit to the "residual" loss between the true value and what the "model so far" predicts. Depending on the method of residual fitting you perform, you get different forms of boosting:

- Fitting the subsequent model by weighting previously incorrectly predicted datapoints more leads to *adaptive boosting (AdaBoost)*.
- Fitting the subsequent model based on the gradients of incorrectly predicted points leads to *gradient boosting*.

1.4 Data Augmentation.

What if I'm feeding in data that isn't so easily described as a feature vector (say, images or text)?[1] Or what if I think I could get more out of each datapoint I have?

In data augmentation, we train one copy of h (a neural network) on an *augmented* sample $X_{aug} = X \cup Z, Z \sim X$ with the *same* loss function L. We generate realizations of Z by taking datapoints from X and distorting them in some way (for example: for an image classifier, by rotating an image x).

In this case, we're leveraging knowledge that we have about X and P to generate samples z_i that can also have been realized from P and whose target value we know as $Target(z_i) = f(Target(x_i))$ for known f (often the identity function). Because we're actually providing the model more data, the model should generalize better.

1.5 More specific changes: Random Forests and Dropout.

We said earlier that we may be having a problem where the model relies too much on individual features (which may be less important "in the wild"). So, why not try training the model without letting it see those features?

Also, let's be frank: the hot models are

- Decision trees/footnote Actually, usually gradient-boosted tree ensembles, which are essentially a combination of "random forests" and "gradient boosting". , for most data you can neatly arrange into tabular format.
- A neural network (of some sort), if you can't.

So what can we do to improve these most commonly used classes of models?

[1] Of course, one can produce feature representations of these sorts of data, but you still need to *make* that representation. If you make a model do it, you haven't really escaped the problem, just pushed it to an unsupervised framework where none of this applies anyway.

1.5.1 Random Forests.

In random forests, we train each of m copies of h (a decision tree) with *the same* initialization on *different* samples $X_i \sim Y_i, Y_i \sim X$ with the *same* loss function L. In this case, Y_i takes random datapoints *and* random features from X. The final model is an aggregation $h = f(h_1, \cdots, h_m)$ (for example, a simple summation).

By removing features at random, the final model (probably) wouldn't be able to rely on one specific feature as a crutch. This greatly improves the generalizability of decision trees.

1.5.2 Dropout.

When using dropout, we train one copy of h (a neural network) on the same sample X on the same loss function L, but *randomly zero out a fraction of signals as they propagate through the network (toward the output)*. At test time, we do not zero out any signals but scale the signals down according to the level of dropout applied during training. This essentially leads to two different modifications at the same time:

- Ensembling. In a sense, by turning off certain signals, we have effectively made a "new" architecture. This "new" architecture needs to learn the dataset without the information that had been carried by the turned-off neurons. Then, at test time, both the "old" and "new" architectures see the sample and propagate their signals.
- Random feature pruning (similar to random forests choosing a subset of features to train on). If you apply dropout to the first layer, you are making the network "not see" certain parts of the input. Once again, this makes a network be less able to "rely" on certain features that may be prevalent at training time but not as important at test time.

2 Conclusion.

There are a number of other methods one can use to improve performance. [2] But the above are some of the most commonly used ways to get more out of the limited quantity of high-quality datapoints you have for the learning problem in question.

Good links:

- https://www.youtube.com/watch?v=P76Gy2eg46A
- https://www.youtube.com/watch?v=uH4FDYv1ARk

-9/16/19

[2] The one that most immediately comes to mind is *transfer learning*, where you're modifying an already-existing model that required a lot of data to train and has already been trained on data that is "close enough" to your use-case to be relevant. Then, you use your own sample to "fine-tune" parameters for your use-case.