Overfitting Avoidance and Complexity Control

To avoid overfitting, we control the complexity of the models induced from the data. Let's start by examining complexity control in tree induction, since tree induction has much flexibility and therefore will tend to overfit a good deal without some mechanism to avoid it. This discussion in the context of trees will lead us to a very general mechanism that will be applicable to other models.

Avoiding Overfitting with Tree Induction

The main problem with tree induction is that it will keep growing the tree to fit the training data until it creates pure leaf nodes. This will likely result in large, overly complex trees that overfit the data. We have seen how this can be detrimental. Tree induction commonly uses two techniques to avoid overfitting. These strategies are (i) to stop growing the tree before it gets too complex, and (ii) to grow the tree until it is too large, then "prune" it back, reducing its size (and thereby its complexity).

There are various methods for accomplishing both. The simplest method to limit tree size is to specify a minimum number of instances that must be present in a leaf. The idea behind this minimum-instance stopping criterion is that for predictive modeling, we essentially are using the data at the leaf to make a statistical estimate of the value of the target variable for future cases that would fall to that leaf. If we make predictions of the target based on a very small subset of data, we might expect them to be inaccurate —especially when we built the tree specifically to try to get pure leaves. A nice property of controlling complexity in this way is that tree induction will automatically grow the tree branches that have a lot of data and cut short branches that have fewer data—thereby automatically adapting the model based on the data distribution.

A key question becomes what threshold we should use. How few instances are we willing to tolerate at a leaf? Five instances? Thirty? One hundred? There is no fixed number, although practitioners tend to have their own preferences based on experience. However, researchers have developed techniques to decide the stopping point statistically. Statistics provides the notion of a "hypothesis test," which you might recall from a basic statistics class. Roughly, a hypothesis test tries to assess whether a difference in some statistic is not due simply to chance. In most cases, the hypothesis test is based on a "p-value," which gives a limit on the probability that the difference in statistic is due to chance. If this value is below a threshold (often 5%, but problem specific), then the hypothesis test concludes that the difference is likely not due to chance. So, for stopping tree growth, an alternative to setting a fixed size for the leaves is to conduct a hypothesis test at every leaf to determine whether the observed difference in (say) information gain could have been due to chance. If the hypothesis test concludes that it was likely not due to chance, then the split is accepted and the tree growing continues. (See "Sidebar: Beware of "multiple comparisons"" on page 139.)

The second strategy for reducing overfitting is to "prune" an overly large tree. Pruning means to cut off leaves and branches, replacing them with leaves. There are many ways to do this, and the interested reader can look into the data mining literature for details. One general idea is to estimate whether replacing a set of leaves or a branch with a leaf would reduce accuracy. If not, then go ahead and prune. The process can be iterated on progressive subtrees until any removal or replacement would reduce accuracy.

We conclude our example of avoiding overfitting in tree induction with the method that will generalize to many different data modeling techniques. Consider the following idea: what if we built trees with all sorts of different complexities? For example, say we stop building the tree after only one node. Then build a tree with two nodes. Then three nodes, etc. We have a set of trees of different complexities. Now, if only there were a way to estimate their generalization performance, we could pick the one that is (estimated to be) the best!

A General Method for Avoiding Overfitting

More generally, if we have a collection of models with different complexities, we could choose the best simply by estimating the generalization performance of each. But how could we estimate their generalization performance? On the (labeled) test data? There's one big problem with that: test data should be strictly *independent* of model building so that we can get an independent estimate of model accuracy. For example, we might want to estimate the ultimate business performance or to compare the best model we can build from one family (say, classification trees) against the best model from another family (say, logistic regression). If we don't care about comparing models or getting an independent estimate of the model accuracy and/or variance, then we could pick the best model based on the testing data.

However, even if we do want these things, we still can proceed. The key is to realize that there was nothing special about the first training/test split we made. Let's say we are saving the test set for a final assessment. We can take the training set and split it again into a training subset and a testing subset. Then we can build models on this training subset and pick the best model based on this testing subset. Let's call the former the *subtraining set* and the latter the *validation set* for clarity. The validation set is separate from the final test set, on which we are never going to make any modeling decisions. This procedure is often called *nested* holdout testing.

Returning to our classification tree example, we can induce trees of many complexities from the subtraining set, then we can estimate the generalization performance for each from the validation set. This would correspond to choosing the top of the inverted-U-shaped holdout curve in Figure 5-3. Say the best model by this assessment has a complexity of 122 nodes (the "sweet spot"). Then we could use this model as our best choice, possibly estimating the actual generalization performance on the final holdout test set. We also could add one more twist. This model was built on a subset of our training data,

since we had to hold out the validation set in order to choose the complexity. But once we've chosen the complexity, why not induce a *new* tree with 122 nodes from the whole, original training set? Then we might get the best of both worlds: using the subtraining/validation split to pick the best complexity without tainting the test set, *and* building a model of this best complexity on the entire training set (subtraining plus validation).

This approach is used in many sorts of modeling algorithms to control complexity. The general method is to choose the value for some complexity parameter by using some sort of nested holdout procedure. Again, it is nested because a second holdout procedure is performed on the training set selected by the first holdout procedure.

Often, nested cross-validation is used. Nested cross-validation is more complicated, but it works as you might suspect. Say we would like to do cross-validation to assess the generalization accuracy of a new modeling technique, which has an adjustable complexity parameter *C*, but we do not know how to set it. So, we run cross-validation as described above. However, before building the model for each fold, we take the training set (refer to Figure 5-9) and first run an experiment: we run another entire cross-validation on just that training set to find the value of *C* estimated to give the best accuracy. The result of that experiment is used only to set the value of *C* to build the actual model for that fold of the cross-validation. Then we build another model using the entire training fold, using that value for *C*, and test on the corresponding test fold. The only difference from regular cross-validation is that for each fold we first run this experiment to find *C*, using another, smaller, cross-validation.

If you understood all that, you would realize that if we used 5-fold cross-validation in both cases, we actually have built 30 total models in the process (yes, *thirty*). This sort of experimental complexity-controlled modeling only gained broad practical application over the last decade or so, because of the obvious computational burden involved.

This idea of using the data to choose the complexity experimentally, as well as to build the resulting model, applies across different induction algorithms and different sorts of complexity. For example, we mentioned that complexity increases with the size of the feature set, so it is usually desirable to cull the feature set. A common method for doing this is to run with many different feature sets, using this sort of nested holdout procedure to pick the best.

For example, *sequential forward selection* (SFS) of features uses a nested holdout procedure to first pick the best individual feature, by looking at all models built using just one feature. After choosing a first feature, SFS tests all models that add a second feature to this first chosen feature. The best pair is then selected. Next the same procedure is done for three, then four, and so on. When adding a feature does not improve classification accuracy on the validation data, the SFS process stops. (There is a similar procedure called *sequential backward elimination* of features. As you might guess, it works by starting with all features and discarding features one at a time. It continues to discard features as long as there is no performance loss.)

This is a common approach. In modern environments with plentiful data and computational power, the data scientist routinely sets modeling parameters by experimenting using some tactical, nested holdout testing (often nested cross-validation).

The next section shows a different way that this method applies to controlling overfitting when learning numerical functions (as described in Chapter 4). We urge you to at least skim the following section because it introduces concepts and vocabulary in common use by data scientists these days.

* Avoiding Overfitting for Parameter Optimization

As just described, avoiding overfitting involves complexity control: finding the "right" balance between the fit to the data and the complexity of the model. In trees we saw various ways for trying to keep the tree from getting too big (too complex) when fitting the data. For equations, such as logistic regression, that unlike trees do not automatically select what attributes to include, complexity can be controlled by choosing a "right" set of attributes.

Chapter 4 introduced the popular family of methods that builds models by explicitly optimizing the fit to the data via a set of numerical parameters. We discussed various linear members of this family, including linear discriminant learners, linear regression, and logistic regression. Many nonlinear models are fit to the data in exactly the same way.

As might be expected given our discussion so far in this chapter and the figures in "Example: Overfitting Linear Functions" on page 119, these procedures also can overfit the data. However, their explicit optimization framework provides an elegant, if technical, method for complexity control. The general strategy is that instead of just optimizing the fit to the data, we optimize some combination of fit and simplicity. Models will be better if they fit the data better, but they also will be better if they are simpler. This general methodology is called *regularization*, a term that is heard often in data science discussions.

The rest of this section discusses briefly (and slightly technically) how regularization is done. Don't worry if you don't really understand the technical details. Do remember that regularization is trying to optimize not just the fit to the data, but a combination of fit to the data and simplicity of the model.

Recall from Chapter 4 that to fit a model involving numeric parameters w to the data we find the set of parameters that maximizes some "objective function" indicating how well it fits the data:

$$\arg\max_{\mathbf{w}} \ \text{fit}(\mathbf{x}, \mathbf{w})$$

(The $\arg\max_{\mathbf{w}}$ just means that you want to maximize the fit over all possible arguments **w**, and are interested in the particular argument **w** that gives the maximum. These would be the parameters of the final model.)

Complexity control via regularization works by adding to this objective function a penalty for complexity:

$$\arg\max_{\mathbf{w}} \ [\text{fit}(\mathbf{x}, \mathbf{w}) - \lambda \cdot \text{penalty}(\mathbf{w})]$$

The λ term is simply a weight that determines how much importance the optimization procedure should place on the penalty, compared to the data fit. At this point, the modeler has to choose λ and the penalty function.

So, as a concrete example, recall from "* Logistic Regression: Some Technical Details" on page 99 that to learn a standard logistic regression model, from data, we find the numeric parameters **w** that yield the linear model most likely to have generated the observed data—the "maximum likelihood" model. Let' represent that as:

$$\arg\max_{\mathbf{w}} \ g_{\text{likelihood}}(\mathbf{x}, \mathbf{w})$$

To learn a "regularized" logistic regression model we would instead compute:

$$\arg\max_{\mathbf{w}} \ [g_{\text{likelihood}}(\mathbf{x}, \mathbf{w}) - \lambda \cdot \text{penalty}(\mathbf{w})]$$

There are different penalties that can be applied, with different properties.[6] The most commonly used penalty is the sum of the squares of the weights, sometimes called the "L2-norm" of *w*. The reason is technical, but basically functions can fit data better if they are allowed to have very large positive and negative weights. The sum of the squares of the weights gives a large penalty when weights have large absolute values.

6. The book *The Elements of Statistical Learning* (Hastie, Tibshirani, & Friedman, 2009) contains an excellent technical discussion of these.

If we incorporate the L2-norm penalty into standard least-squares linear regression, we get the statistical procedure called *ridge regression*. If instead we use the sum of the absolute values (rather than the squares), known as the L1-norm, we get a procedure known as the *lasso* (Hastie et al., 2009). More generally, this is called L1-regularization. For reasons that are quite technical, L1-regularization ends up zeroing out many coefficients. Since these coefficients are the multiplicative weights on the features, L1-regularization effectively performs an automatic form of feature selection.

Now we have the machinery to describe in more detail the linear support vector machine, introduced in "Support Vector Machines, Briefly" on page 91. There we waved our hands and told you that the support vector machine "maximizes the margin" between the classes by fitting the "fattest bar" between the classes. Separately we discussed that it uses hinge loss (see "Sidebar: Loss functions" on page 94) to penalize errors. We now can connect these together, and directly to logistic regression. Specifically, linear support vector machine learning is almost equivalent to the L2-regularized logistic regression just discussed; the only difference is that a support vector machine uses hinge loss instead of likelihood in its optimization. The support vector machine optimizes this equation:

$$\arg\max_{\mathbf{w}} \left[- g_{\text{hinge}}(\mathbf{x}, \mathbf{w}) - \lambda \cdot \text{penalty}(\mathbf{w}) \right]$$

where g_{hinge}, the hinge loss term, is negated because lower hinge loss is better.

Finally, you may be saying to yourself: all this is well and good, but a lot of magic seems to be hidden in this λ parameter, which the modeler has to choose. How in the world would the modeler choose that for some real domain like churn prediction, or online ad targeting, or fraud detection?

It turns out that we already have a straightforward way to choose λ. We've discussed how a good tree size and a good feature set can be chosen via nested cross-validation on the training data. We can choose λ the same way. This cross-validation would essentially conduct automated experiments on subsets of the training data and find a good λ value. Then this λ would be used to learn a regularized model on all the training data. This has become the standard procedure for building numerical models that give a good balance between data fit and model complexity. This general approach to optimizing the parameter values of a data mining procedure is known as grid search.

Sidebar: Beware of "multiple comparisons"

Consider the following scenario. You run an investment firm. Five years ago, you wanted to have some marketable small-cap mutual fund products to sell, but your analysts had been awful at picking small-cap stocks. So you undertook the following procedure. You started 1,000 different mutual funds, each including a small set of stocks randomly chosen from those that make up the Russell 2000 index (the main index for small-cap stocks). Your firm invested in all 1,000 of these funds, but told no one about them. Now, five years later, you look at their performance. Since they have different stocks in them, they will have had different returns. Some will be about the same as the index, some will be worse, and some will be better. The best one might be a lot better. Now, you liquidate all the funds but the best few, and you present these to the public. You can "honestly" claim that their 5-year return is substantially better than the return of the Russell 2000 index.

So, what's the problem? The problem is that you randomly chose the stocks! You have no idea whether the stocks in these "best" funds performed better because they indeed are fundamentally better, or because you cherry-picked the best from a large set that simply varied in performance. If you flip 1,000 fair coins many times each, one of them will have come up heads much more than 50% of the time. However, choosing that coin as the "best" of the coins for later flipping obviously is silly. These are instances of "the problem of multiple comparisons," a very important statistical phenomenon that business analysts and data scientists should always keep in mind. Beware whenever someone does many tests and then picks the results that look good. Statistics books will warn against running multiple statistical hypothesis tests, and then looking at the ones that give "significant" results. These usually violate the assumptions behind the statistical tests, and the actual significance of the results is dubious.

The underlying reasons for overfitting when building models from data are essentially problems of multiple comparisons (Jensen & Cohen, 2000). Note that even the procedures for avoiding overfitting themselves undertake multiple comparisons (e.g., choosing the best complexity for a model by comparing many complexities). There is no silver bullet or magic formula to truly get "the optimal" model to fit the data. Nonetheless, care can be taken to reduce overfitting as much as possible, by using the holdout procedures described in this chapter and if possible by looking carefully at the results before declaring victory. For example, if the fitting graph truly has an inverted-U-shape, one can be much more confident that the top represents a "good" complexity than if the curve jumps around randomly.