Chapter 7

Mathematical Models

> All models are wrong, but some models are useful.
>
> – George Box

So far in this book, a variety of tools have been developed to manipulate and interpret data. But we haven't really dealt with *modeling*, which is the process of encapsulating information into a tool which can forecast and make predictions.

Predictive models are structured around some idea of what causes future events to happen. Extrapolating from recent trends and observations assumes a world view that the future will be like the past. More sophisticated models, such as the laws of physics, provide principled notions of causation; fundamental explanations of why things happen.

This chapter will concentrate on designing and validating models. Effectively formulating models requires a detailed understanding of the space of possible choices.

Accurately evaluating the performance of a model can be surprisingly hard, but it is essential for knowing how to interpret the resulting predictions. The best forecasting system is not necessarily the most accurate one, but the model with the best sense of its boundaries and limitations.

7.1 Philosophies of Modeling

Engineers and scientists are often leery of the p-word (philosophy). But it pays to think in some fundamental way about what we are trying to do, and why. Recall that people turn to data scientists for wisdom, instead of programs.

In this section, we will turn to different ways of thinking about models to help shape the way we build them.

7.1.1 Occam's Razor

Occam's razor is the philosophical principle that "the simplest explanation is

© The Author(s) 2017
S.S. Skiena, *The Data Science Design Manual,*
Texts in Computer Science, https://doi.org/10.1007/978-3-319-55444-0_7

the best explanation." According to William of Occam, a 13th-century theologian, given two models or theories which do an equally accurate job of making predictions, we should opt for the simpler one as sounder and more robust. It is more likely to be making the right decision for the right reasons.

Occam's notion of simpler generally refers to reducing the number of assumptions employed in developing the model. With respect to statistical modeling, Occam's razor speaks to the need to minimize the parameter count of a model. *Overfitting* occurs when a model tries too hard to achieve accurate performance on its training data. This happens when there are so many parameters that the model can essentially memorize its training set, instead of generalizing appropriately to minimize the effects of error and outliers.

Overfit models tend to perform extremely well on training data, but much less accurately on independent test data. Invoking Occam's razor requires that we have a meaningful way to evaluate how accurately our models are performing.

Simplicity is not an absolute virtue, when it leads to poor performance. *Deep learning* is a powerful technique for building models with millions of parameters, which we will discuss in Section 11.6. Despite the danger of overfitting, these models perform extremely well on a variety of complex tasks. Occam would have been suspicious of such models, but come to accept those that have substantially more predictive power than the alternatives.

Appreciate the inherent trade-off between accuracy and simplicity. It is almost always possible to "improve" the performance of any model by kludging-on extra parameters and rules to govern exceptions. Complexity has a cost, as explicitly captured in machine learning methods like LASSO/ridge regression. These techniques employ penalty functions to minimize the features used in the model.

> *Take-Home Lesson*: Accuracy is not the best metric to use in judging the quality of a model. Simpler models tend to be more robust and understandable than complicated alternatives. Improved performance on specific tests is often more attributable to variance or overfitting than insight.

7.1.2 Bias–Variance Trade-Offs

This tension between model complexity and performance shows up in the statistical notion of the *bias–variance trade-off*:

- *Bias* is error from incorrect assumptions built into the model, such as restricting an interpolating function to be linear instead of a higher-order curve.
- *Variance* is error from sensitivity to fluctuations in the training set. If our training set contains sampling or measurement error, this noise introduces variance into the resulting model.

Errors of bias produce *underfit* models. They do not fit the training data as tightly as possible, were they allowed the freedom to do so. In popular discourse,

I associate the word "bias" with *prejudice*, and the correspondence is fairly apt: an apriori assumption that one group is inferior to another will result in less accurate predictions than an unbiased one. Models that perform lousy on both training and testing data are underfit.

Errors of variance result in *overfit* models: their quest for accuracy causes them to mistake noise for signal, and they adjust so well to the training data that noise leads them astray. Models that do much better on testing data than training data are overfit.[1]

Take-Home Lesson: Models based on first principles or assumptions are likely to suffer from bias, while data-driven models are in greater danger of overfitting.

7.1.3 What Would Nate Silver Do?

Nate Silver is perhaps the most prominent public face of data science today. A quantitative fellow who left a management consulting job to develop baseball forecasting methods, he rose to fame through his election forecast website `http://www.fivethirtyeight.com`. Here he used quantitative methods to analyze poll results to predict the results of U.S. presidential elections. In the 2008 election, he accurately called the winner of 49 of the 50 states, and improved in 2012 to bag 50 out of 50. The results of the 2016 election proved a shock to just about everyone, but alone among public commentators Nate Silver had identified a substantial chance of Trump winning the electoral college while losing the popular vote. This indeed proved to be the case.

Silver wrote an excellent book *The Signal and the Noise: Why so many predictions fail – but some don't* [Sil12]. There he writes sensibly about state-of-the-art forecasting in several fields, including sports, weather and earthquake prediction, and financial modeling. He outlines principles for effective modeling, including:

- *Think probabilistically*: Forecasts which make concrete statements are *less* meaningful than those that are inherently probabilistic. A forecast that Trump has only 28.3% chance of winning is more meaningful than one that categorically states that he will lose.

  The real world is an uncertain place, and successful models recognize this uncertainty. There are always a range of possible outcomes that can occur with slight perturbations of reality, and this should be captured in your model. Forecasts of numerical quantities should not be single numbers, but instead report probability distributions. Specifying a standard deviation σ along with the mean prediction μ suffices to describe such a distribution, particularly if it is assumed to be normal.

[1] To complete this taxonomy, models that do better on testing data than the training data are said to be *cheating*.

Several of the machine learning techniques we will study naturally provide probabilistic answers. Logistic regression provides a confidence along with each classification it makes. Methods that vote among the labels of the k nearest neighbors define a natural confidence measure, based on the consistency of the labels in the neighborhood. Collecting ten of eleven votes for blue means something stronger than seven out of eleven.

- *Change your forecast in response to new information*: Live models are much more interesting than dead ones. A model is *live* if it is continually updating predictions in response to new information. Building an infrastructure that maintains a live model is more intricate than that of a one-off computation, but much more valuable.

  Live models are more intellectually honest than dead ones. Fresh information *should* change the result of any forecast. Scientists should be open to changing opinions in response to new data: indeed, this is what separates scientists from hacks and trolls.

  Dynamically-changing forecasts provide excellent opportunities to evaluate your model. Do they ultimately converge on the correct answer? Does uncertainty diminish as the event approaches? Any live model should track and display its predictions over time, so the viewer can gauge whether changes accurately reflected the impact of new information.

- *Look for consensus*: A good forecast comes from multiple distinct sources of evidence. Data should derive from as many different sources as possible. Ideally, multiple models should be built, each trying to predict the same thing in different ways. You should have an opinion as to which model is the best, but be concerned when it substantially differs from the herd.

  Often third parties produce competing forecasts, which you can monitor and compare against. Being different doesn't mean that you are wrong, but it does provide a reality check. Who has been doing better lately? What explains the differences in the forecast? Can your model be improved?

  Google's Flu Trends forecasting model predicted disease outbreaks by monitoring key words on search: a surge in people looking for *aspirin* or *fever* might suggest that illness is spreading. Google's forecasting model proved quite consistent with the Center for Disease Control's (CDC) statistics on actual flu cases for several years, until they embarrassingly went astray.

  The world changes. Among the changes was that Google's search interface began to suggest search queries in response to a user's history. When offered the suggestion, many more people started searching for *aspirin* after searching for *fever*. And the old model suddenly wasn't accurate anymore. Google's sins lay in not monitoring its performance and adjusting over time.

Certain machine learning methods explicitly strive for consensus. *Boosting* algorithms combine large numbers of weak classifiers to produce a strong one. Ensemble decision tree methods build many independent classifiers, and vote among them to make the best decision. Such methods can have a robustness which eludes more single-track models.

- *Employ Baysian reasoning*: Bayes' theorem has several interpretations, but perhaps most cogently provides a way to calculate how probabilities change in response to new evidence. When stated as

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

it provides a way to calculate how the probability of event A changes in response to new evidence B.

Applying Bayes' theorem requires a *prior* probability $P(A)$, the likelihood of event A *before* knowing the status of a particular event B. This might be the result of running a classifier to predict the status of A from other features, or background knowledge about event frequencies in a population. Without a good estimate for this prior, it is very difficult to know how seriously to take the classifier.

Suppose A is the event that person x is actually a terrorist, and B is the result of a feature-based classifier that decides if x looks like a terrorist. When trained/evaluated on a data set of 1,000 people, half of whom were terrorists, the classifier achieved an enviable accuracy of, say, 90%. The classifier now says that Skiena looks like a terrorist. What is the probability that Skiena *really* is a terrorist?

The key insight here is that the prior probability of "x is a terrorist" is really, really low. If there are a hundred terrorists operating in the United States, then $P(A) = 100/300,000,000 = 3.33 \times 10^{-7}$. The probability of the terrorist detector saying yes, $P(B) = 0.5$, while the probability of the detector being right when it says yes $P(B|A) = 0.9$. Multiplying this out gives a still very tiny probability that I am a bad guy,

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} = \frac{(0.9)(3.33 \times 10^{-7})}{(0.5)} = 6 \times 10^{-7}$$

although admittedly now greater than that of a random citizen.

Factoring in prior probabilities is essential to getting the right interpretation from this classifier. Bayesian reasoning starts from the prior distribution, then weighs further evidence by how strongly it should impact the probability of the event.

7.2 A Taxonomy of Models

Models come in, well, many different models. Part of developing a philosophy of modeling is understanding your available degrees of freedom in design and

implementation. In this section, we will look at model types along several different dimensions, reviewing the primary technical issues which arise to distinguish each class.

7.2.1 Linear vs. Non-Linear Models

Linear models are governed by equations that weigh each feature variable by a coefficient reflecting its importance, and sum up these values to produce a score. Powerful machine learning techniques, such as linear regression, can be used to identify the best possible coefficients to fit training data, yielding very effective models.

But generally speaking, the world is not linear. Richer mathematical descriptions include higher-order polynomials, logarithms, and exponentials. These permit models that fit training data much more tightly than linear functions can. Generally speaking, it is much harder to find the best possible coefficients to fit non-linear models. But we don't *have* to find the best possible fit: deep learning methods, based on neural networks, offer excellent performance despite inherent difficulties in optimization.

Modeling cowboys often sneer in contempt at the simplicity of linear models. But linear models offer substantial benefits. They are readily understandable, generally defensible, easy to build, and avoid overfitting on modest-sized data sets. Occam's razor tells us that "the simplest explanation is the best explanation." I am generally happier with a robust linear model, yielding an accuracy of $x\%$, than a complex non-linear beast only a few percentage points better on limited testing data.

7.2.2 Blackbox vs. Descriptive Models

Black boxes are devices that do their job, but in some unknown manner. Stuff goes in and stuff comes out, but how the sausage is made is completely impenetrable to outsiders.

By contrast, we prefer models that are *descriptive*, meaning they provide some insight into why they are making their decisions. Theory-driven models are generally descriptive, because they are explicit implementations of a particular well-developed theory. If you believe the theory, you have a reason to trust the underlying model, and any resulting predictions.

Certain machine learning models prove less opaque than others. Linear regression models are descriptive, because one can see exactly which variables receive the most weight, and measure how much they contribute to the resulting prediction. Decision tree models enable you to follow the exact decision path used to make a classification. "Our model denied you a home mortgage because your income is less than $10,000 per year, you have greater than $50,000 in credit card debt, and you have been unemployed over the past year."

But the unfortunate truth is that blackbox modeling techniques such as deep learning can be extremely effective. Neural network models are generally completely opaque as to *why* they do what they do. Figure 7.1 makes this

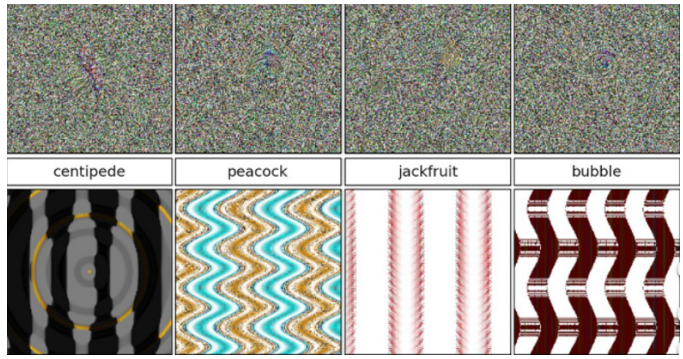

Figure 7.1: Synthetic images that are mistakenly recognized as objects by state-of-the-art Deep Learning neural networks, each with a confidence greater than 99.6%. *Source*: [NYC15].

clear. It shows images which were very carefully constructed to fool state-of-the-art neural networks. They succeeded brilliantly. The networks in question had $\geq 99.6\%$ confidence that they had found the right label for every image in Figure 7.1.

The scandal here is *not* that the network got the labels wrong on these perverse images, for these recognizers are very impressive systems. Indeed, they were much more accurate than dreamed possible only a year or two before. The problem is that the creators of these classifiers had no idea why their programs made such terrible errors, or how they could prevent them in the future.

A similar story is told of a system built for the military to distinguish images of cars from trucks. It performed well in training, but disastrously in the field. Only later was it realized that the training images for cars were shot on a sunny day and those of trucks on a cloudy day, so the system had learned to link the sky in the background with the class of the vehicle.

Tales like these highlight why visualizing the training data and using descriptive models can be so important. You must be convinced that your model has the information it needs to make the decisions you are asking of it, particularly in situations where the stakes are high.

7.2.3 First-Principle vs. Data-Driven Models

First-principle models are based on a belief of how the system under investigation really works. It might be a theoretical explanation, like Newton's laws of motion. Such models can employ the full weight of classical mathematics: calculus, algebra, geometry, and more. The model might be a discrete event simulation, as will be discussed in Section 7.7. It might be seat-of-the-pants reasoning from an understanding of the domain: voters are unhappy if the economy is bad, therefore variables which measure the state of the economy should help us predict who will win the election.

In contrast, *data-driven models* are based on observed correlations between input parameters and outcome variables. The same basic model might be used to predict tomorrow's weather or the price of a given stock, differing only on the data it was trained on. Machine learning methods make it possible to build an effective model on a domain one knows nothing about, provided we are given a good enough training set.

Because this is a book on data science, you might infer that my heart lies more on the side of data-driven models. But this isn't really true. Data science is also about *science*, and things that happen for understandable reasons. Models which ignore this are doomed to fail embarrassingly in certain circumstances.

There is an alternate way to frame this discussion, however. *Ad hoc models* are built using domain-specific knowledge to guide their structure and design. These tend to be brittle in response to changing conditions, and difficult to apply to new tasks. In contrast, machine learning models for classification and regression are *general*, because they employ no problem-specific ideas, only specific data. Retrain the models on fresh data, and they adapt to changing conditions. Train them on a different data set, and they can do something completely different. By this rubric, general models sound much better than ad hoc ones.

The truth is that the best models are a mixture of both theory and data. It is important to understand your domain as deeply as possible, while using the best data you can in order to fit and evaluate your models.

7.2.4 Stochastic vs. Deterministic Models

Demanding a single deterministic "prediction" from a model can be a fool's errand. The world is a complex place of many realities, with events that generally would not unfold in exactly the same way if time could be run over again. Good forecasting models incorporate such thinking, and produce probability distributions over all possible events.

Stochastic is a fancy word meaning "randomly determined." Techniques that explicitly build some notion of probability into the model include logistic regression and Monte Carlo simulation. It is important that your model observe the basic properties of probabilities, including:

- *Each probability is a value between 0 and 1*: Scores that are not constrained to be in this range do not directly estimate probabilities. The solution is often to put the values through a logit function (see Section 4.4.1) to turn them into probabilities in a principled way.

- *That they must sum to 1*: Independently generating values between 0 and 1 does not mean that they together add up to a unit probability, over the full event space. The solution here is to scale these values so that they do, by dividing each by the partition function. See Section 9.7.4. Alternately, rethink your model to understand why they didn't add up in the first place.

- *Rare events do not have probability zero*: Any event that is possible must have a greater than zero probability of occurrence. *Discounting* is a way of evaluating the likelihood of unseen but possible events, and will be discussed in Section 11.1.2.

Probabilities are a measure of humility about the accuracy of our model, and the uncertainty of a complex world. Models must be honest in what they do and don't know.

There are certain advantages of deterministic models, however. First-principle models often yield only one possible answer. Newton's laws of motion will tell you *exactly* how long a mass takes to fall a given distance.

That deterministic models always return the same answer helps greatly in debugging their implementation. This speaks to the need to optimize *repeatability* during model development. Fix the initial seed if you are using a random number generator, so you can rerun it and get the same answer. Build a regression test suite for your model, so you can confirm that the answers remain identical on a given input after program modifications.

7.2.5 Flat vs. Hierarchical Models

Interesting problems often exist on several different levels, each of which may require independent submodels. Predicting the future price for a particular stock really should involve submodels for analyzing such separate issues as (a) the general state of the economy, (b) the company's balance sheet, and (c) the performance of other companies in its industrial sector.

Imposing a hierarchical structure on a model permits it to be built and evaluated in a logical and transparent way, instead of as a black box. Certain subproblems lend themselves to theory-based, first-principle models, which can then be used as features in a general data-driven model. Explicitly hierarchical models are descriptive: one can trace a final decision back to the appropriate top-level subproblem, and report how strongly it contributed to making the observed result.

The first step to build a hierarchical model is explicitly decomposing our problem into subproblems. Typically these represent mechanisms governing the underlying process being modeled. What *should* the model depend on? If data and resources exist to make a principled submodel for each piece, great! If not, it is OK to leave it as a null model or baseline, and explicitly describe the omission when documenting the results.

Deep learning models can be thought of as being both flat and hierarchical, at the same time. They are typically trained on large sets of unwashed data, so there is no explicit definition of subproblems to guide the subprocess. Looked at as a whole, the network does only one thing. But because they are built from multiple nested layers (the *deep* in deep learning), these models presume that there are complex features there to be learned from the lower level inputs.

I am always reluctant to believe that machine learning models prove better than me at inferring the basic organizing principles in a domain that I un-

derstand. Even when employing deep learning, it pays to sketch out a rough hierarchical structure that likely exists for your network to find. For example, any image processing network should generalize from patches of pixels to edges, and then from boundaries to sub-objects to scene analysis as we move to higher layers. This influences the architecture of your network, and helps you validate it. Do you see evidence that your network is making the right decisions for the right reasons?

7.3 Baseline Models

A wise man once observed that a broken clock is right twice a day. As modelers we strive to be better than this, but proving that we are requires some level of rigorous evaluation.

The first step to assess the complexity of your task involves building *baseline models*: the simplest reasonable models that produce answers we can compare against. More sophisticated models *should* do better than baseline models, but verifying that they really do and, if so by how much, puts its performance into the proper context.

Certain forecasting tasks are inherently harder than others. A simple baseline ("yes") has proven very accurate in predicting whether the sun will rise tomorrow. By contrast, you could get rich predicting whether the stock market will go up or down 51% of the time. Only after you decisively beat your baselines can your models really be deemed effective.

7.3.1 Baseline Models for Classification

There are two common tasks for data science models: *classification* and *value prediction*. In classification tasks, we are given a small set of possible labels for any given item, like (spam or not spam), (man or woman), or (bicycle, car, or truck). We seek a system that will generate a label accurately describing a particular instance of an email, person, or vehicle.

Representative baseline models for classification include:

- *Uniform or random selection among labels*: If you have absolutely no prior distribution on the objects, you might as well make an arbitrary selection using the broken watch method. Comparing your stock market prediction model against random coin flips will go a long way to showing how hard the problem is.

  I think of such a blind classifier as *the monkey*, because it is like asking your pet to make the decision for you. In a prediction problem with twenty possible labels or classes, doing substantially better than 5% is the first evidence that you have some insight into the problem. You first have to show me that you can beat the monkey before I start to trust you.

- *The most common label appearing in the training data*: A large training set usually provides some notion of a prior distribution on the classes.

Selecting the most frequent label is better than selecting them uniformly or randomly. This is the theory behind the sun-will-rise-tomorrow baseline model.

- *The most accurate single-feature model*: Powerful models strive to exploit all the useful features present in a given data set. But it is valuable to know what the best single feature can do. Building the best classifier on a single numerical feature x is easy: we are declaring that the item is in class 1 if $x \geq t$, and class 2 if otherwise. To find the best threshold t, we can test all n possible thresholds of the form $t_i = x_i + \epsilon$, where x_i is the value of the feature in the ith of n training instances. Then select the threshold which yields the most accurate classifier on your training data.

  Occam's razor deems the simplest model to be best. Only when your complicated model beats all single-factor models does it start to be interesting.

- *Somebody else's model*: Often we are not the first person to attempt a particular task. Your company may have a legacy model that you are charged with updating or revising. Perhaps a close variant of the problem has been discussed in an academic paper, and maybe they even released their code on the web for you to experiment with.

  One of two things can happen when you compare your model against someone else's work: either you beat them or you don't. If you beat them, you now have something worth bragging about. If you don't, it is a chance to learn and improve. *Why* didn't you win? The fact that you lost gives you certainty that your model can be improved, at least to the level of the other guy's model.

- *Clairvoyance*: There are circumstances when even the best possible model cannot theoretically reach 100% accuracy. Suppose that two data records are exactly the same in feature space, but with contradictory labels. There is no deterministic classifier that could ever get both of these problems right, so we're doomed to less than perfect performance. But the tighter upper bound from an optimally clairvoyant predictor might convince you that your baseline model is better than you thought.

  The need for better upper bounds often arises when your training data is the result of a human annotation process, and multiple annotators evaluate the same instances. We get inherent contradictions whenever two annotators disagree with each other. I've worked on problems where 86.6% correct was the highest possible score. This lowers expectations. A good bit of life advice is to expect little from your fellow man, and realize you will have to make do with a lot less than that.

7.3.2 Baseline Models for Value Prediction

In value prediction problems, we are given a collection of feature-value pairs (f_i, v_i) to use to train a function F such that $F(v_i) = v_i$. Baseline models for value prediction problems follow from similar techniques to what were proposed for classification, like:

- *Mean or median*: Just ignore the features, so you can always output the consensus value of the target. This proves to be quite an informative baseline, because if you can't substantially beat always guessing the mean, either you have the wrong features or are working on a hopeless task.
- *Linear regression*: We will thoroughly cover linear regression in Section 9.1. But for now, it suffices to understand that this powerful but simple-to-use technique builds the best possible linear function for value prediction problems. This baseline enables you to better judge the performance of non-linear models. If they do not perform substantially better than the linear classifier, they are probably not worth the effort.
- *Value of the previous point in time*: Time series forecasting is a common task, where we are charged with predicting the value $f(t_n, x)$ at time t_n given feature set x and the observed values $f'(t_i)$ for $1 \leq i < n$. But today's weather is a good guess for whether it will rain tomorrow. Similarly, the value of the previous observed value $f'(t_{n-1})$ is a reasonable forecast for time $f(t_n)$. It is often surprisingly difficult to beat this baseline in practice.

Baseline models must be fair: they should be simple but not stupid. You want to present a target that you hope or expect to beat, but not a sitting duck. You should feel relieved when you beat your baseline, but not boastful or smirking.

7.4 Evaluating Models

Congratulations! You have built a predictive model for classification or value prediction. Now, how good is it?

This innocent-looking question does not have a simple answer. We will detail the key technical issues in the sections below. But the informal *sniff test* is perhaps the most important criteria for evaluating a model. Do you *really* believe that it is doing a good job on your training and testing instances?

The formal evaluations that will be detailed below reduce the performance of a model down to a few summary statistics, aggregated over many instances. But many sins in a model can be hidden when you only interact with these aggregate scores. You have no way of knowing whether there are bugs in your implementation or data normalization, resulting in poorer performance than it should have. Perhaps you intermingled your training and test data, yielding much better scores on your testbed than you deserve.

		Predicted Class	
		Yes	No
Actual Class	Yes	True Positives (TP)	False Negatives (FN)
	No	False Positives (FP)	True Negatives (TN)

Figure 7.2: The confusion matrix for binary classifiers, defining different classes of correct and erroneous predictions.

To really know what is happening, you need to do a sniff test. My personal sniff test involves looking carefully at a few example instances where the model got it right, and a few where it got it wrong. The goal is to make sure that I understand why the model got the results that it did. Ideally these will be records whose "names" you understand, instances where you have some intuition about what the right answers should be as a result of exploratory data analysis or familiarity with the domain.

Take-Home Lesson: Too many data scientists only care about the evaluation statistics of their models. But good scientists have an understanding of whether the errors they are making are defensible, serious, or irrelevant.

Another issue is your degree of surprise at the evaluated accuracy of the model. Is it performing better or worse than you expected? How accurate do you think *you* would be at the given task, if you had to use human judgment.

A related question is establishing a sense of how valuable it would be if the model performed just a little better. An NLP task that classifies words correctly with 95% accuracy makes a mistake roughly once every two to three sentences. Is this good enough? The better its current performance is, the harder it will be to make further improvements.

But the best way to assess models involves *out-of-sample* predictions, results on data that you never saw (or even better, did not exist) when you built the model. Good performance on the data that you trained models on is very suspect, because models can easily be overfit. Out of sample predictions are the key to being honest, provided you have enough data and time to test them. This is why I had my *Quant Shop* students build models to make predictions of future events, and then forced them to watch and see whether they were right or not.

7.4.1 Evaluating Classifiers

Evaluating a classifier means measuring how accurately our predicted labels match the gold standard labels in the evaluation set. For the common case of two distinct labels or classes (binary classification), we typically call the smaller and more interesting of the two classes as *positive* and the larger/other class as *negative*. In a spam classification problem, the spam would typically be positive and the ham (non-spam) would be negative. This labeling aims to ensure that

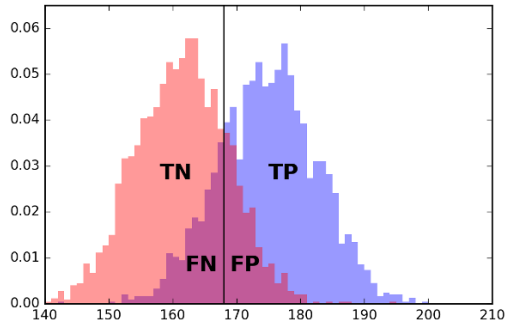

Figure 7.3: What happens if we classify everyone of height ≥ 168 centimeters as male? The four possible results in the confusion matrix reflect which instances were classified correctly (TP and TN) and which ones were not (FN and FP).

identifying the positives is at least as hard as identifying the negatives, although often the test instances are selected so that the classes are of equal cardinality.

There are four possible results of what the classification model could do on any given instance, which defines the *confusion matrix* or *contingency table* shown in Figure 7.2:

- *True Positives (TP)*: Here our classifier labels a positive item as positive, resulting in a win for the classifier.
- *True Negatives (TN)*: Here the classifier correctly determines that a member of the negative class deserves a negative label. Another win.
- *False Positives (FP)*: The classifier mistakenly calls a negative item as a positive, resulting in a "type I" classification error.
- *False Negatives (FN)*: The classifier mistakenly declares a positive item as negative, resulting in a "type II" classification error.

Figure 7.3 illustrates where these result classes fall in separating two distributions (men and women), where the decision variable is height as measured in centimeters. The classifier under evaluation labels everyone of height ≥ 168 centimeters as male. The purple regions represent the intersection of both male and female. These tails represent the incorrectly classified elements.

Accuracy, Precision, Recall, and F-Score

There are several different evaluation statistics which can be computed from the true/false positive/negative counts detailed above. The reason we need so many statistics is that we must defend our classifier against two baseline opponents, the sharp and the monkey.

The *sharp* is the opponent who knows what evaluation system we are using, and picks the baseline model which will do best according to it. The sharp will try to make the evaluation statistic look bad, by achieving a high score with a useless classifier. That might mean declaring all items positive, or perhaps all negative.

In contrast, the *monkey* randomly guesses on each instance. To interpret our model's performance, it is important to establish by how much it beats both the sharp and the monkey.

The first statistic measures the *accuracy* of classifier, the ratio of the number of correct predictions over total predictions. Thus:

$$accuracy = \frac{TP + TN}{TP + TN + FN + FP}$$

By multiplying such fractions by 100, we can get a percentage accuracy score.

Accuracy is a sensible number which is relatively easy to explain, so it is worth providing in any evaluation environment. How accurate is the monkey, when half of the instances are positive and half negative? The monkey would be expected to achieve an accuracy of 50% by random guessing. The same accuracy of 50% would be achieved by the sharp, by always guessing positive, or (equivalently) always guessing negative. The sharp would get a different half of the instances correct in each case.

Still, accuracy alone has limitations as an evaluation metric, particularly when the positive class is much smaller than the negative class. Consider the development of a classifier to diagnose whether a patient has cancer, where the positive class has the disease (i.e. tests positive) and the negative class is healthy. The prior distribution is that the vast majority of people are healthy, so

$$p = \frac{|positive|}{|positive| + |negative|} \ll \frac{1}{2}$$

The expected accuracy of a fair-coin monkey would still be 0.5: it should get an average of half of the positives and half the negatives right. But the sharp would declare everyone to be healthy, achieving an accuracy of $1 - p$. Suppose that only 5% of the test takers really had the disease. The sharp could brag about her accuracy of 95%, while simultaneously dooming all members of the diseased class to an early death.

Thus we need evaluation metrics that are more sensitive to getting the positive class right. *Precision* measures how often this classifier is correct when it dares to say positive:

$$precision = \frac{TP}{(TP + FP)}$$

Achieving high precision is impossible for either a sharp or a monkey, because the fraction of positives ($p = 0.05$) is so low. If the classifier issues too many positive labels, it is doomed to low precision because so many bullets miss their mark, resulting in many false positives. But if the classifier is stingy with positive labels, very few of them are likely to connect with the rare positive

	Monkey	
	predicted	class
	yes	no
yes	$(pn)q$	$(pn)(1-q)$
no	$((1-p)n)q$	$((1-p)n)(1-q)$

	Balanced Classifier	
	predicted	class
	yes	no
yes	$(pn)q$	$(pn)(1-q)$
no	$((1-p)n)(1-q)$	$((1-p)n)q$

Figure 7.4: The expected performance of a monkey classifier on n instances, where $p \cdot n$ are positive and $(1-p) \cdot n$ are negative. The monkey guesses positive with probability q (left). Also, the expected performance of a balanced classifier, which somehow correctly classifies members of each class with probability q (right).

	Monkey		Sharp		Balanced Classifier				
q	0.05	0.5	0.0	1.0	0.5	0.75	0.9	0.99	1.0
accuracy	0.905	0.5	0.95	0.05	0.5	0.75	0.9	0.99	1.
precision	0.05	0.05	—	0.05	0.05	0.136	0.321	0.839	1.
recall	0.05	0.5	0.	1.	0.5	0.75	0.9	0.99	1.
F score	0.05	0.091	—	0.095	0.091	0.231	0.474	0.908	1.

Figure 7.5: Performance of several classifiers, under different performance measures.

instances, so the classifier achieves low true positives. These baseline classifiers achieve precision proportional to the positive class probability $p = 0.05$, because they are flying blind.

In the cancer diagnosis case, we might be more ready to tolerate false positives (errors where we scare a healthy person with a wrong diagnosis) than false negatives (errors where we kill a sick patient by misdiagnosing their illness). *Recall* measures how often you prove right on all positive instances:

$$recall = \frac{TP}{(TP + FN)}$$

A high recall implies that the classifier has few false negatives. The easiest way to achieve this declares that *everyone* has cancer, as done by a sharp always answering yes. This classifier has high recall but low precision: 95% of the test takers will receive an unnecessary scare. There is an inherent trade-off between precision and recall when building classifiers: the braver your predictions are, the less likely they are to be right.

But people are hard-wired to want a single measurement describing the performance of their system. The *F-score* (or sometimes F1-score) is such a combination, returning the harmonic mean of precision and recall:

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

F-score is a very tough measure to beat. The harmonic mean is always less than or equal to the arithmetic mean, and the lower number has a dispropor-

tionate large effect. Achieving a high F-score requires both high recall and high precision. None of our baseline classifiers manage a decent F-score despite high accuracy and recall values, because their precision is too low.

The F-score and related evaluation metrics were developed to evaluate meaningful classifiers, not monkeys or sharps. To gain insight in how to interpret them, let's consider a class of magically *balanced* classifiers, which somehow show equal accuracy on both positive and negative instances. This isn't usually the case, but classifiers selected to achieve high F-scores must balance precision and recall statistics, which means they must show decent performance on both positive and negative instances.

Figure 7.5 summarizes the performance of both baseline and balanced classifiers on our cancer detection problem, benchmarked on all four of our evaluation metrics. The take away lessons are:

- *Accuracy is a misleading statistic when the class sizes are substantially different*: A baseline classifier mindlessly answering "no" for every instance achieved an accuracy of 95% on the cancer problem, better even than a balanced classifier that got 94% right on each class.

- *Recall equals accuracy if and only if the classifiers are balanced*: Good things happen when the accuracy for recognizing both classes is the same. This doesn't happen automatically during training, when the class sizes are different. Indeed, this is one reason why it is generally a good practice to have an equal number of positive and negative examples in your training set.

- *High precision is very hard to achieve in unbalanced class sizes*: Even a balanced classifier that gets 99% accuracy on both positive and negative examples cannot achieve a precision above 84% on the cancer problem. This is because there are twenty times more negative instances than positive ones. The false positives from misclassifying the larger class at a 1% rate remains substantial against the background of 5% true positives.

- *F-score does the best job of any single statistic, but all four work together to describe the performance of a classifier*: Is the precision of your classifier greater than its recall? Then it is labeling too few instances as positives, and so perhaps you can tune it better. Is the recall higher than the precision? Maybe we can improve the F-score by being less aggressive in calling positives. Is the accuracy far from the recall? Then our classifier isn't very balanced. So check which side is doing worse, and how we might be able to fix it.

A useful trick to increase the precision of a model at the expense of recall is to give it the power to say "I don't know." Classifiers typically do better on easy cases than hard ones, with the difficulty defined by how far the example is from being assigned the alternate label.

Defining a notion of *confidence* that your proposed classification is correct is the key to when you should pass on a question. Only venture a guess when your

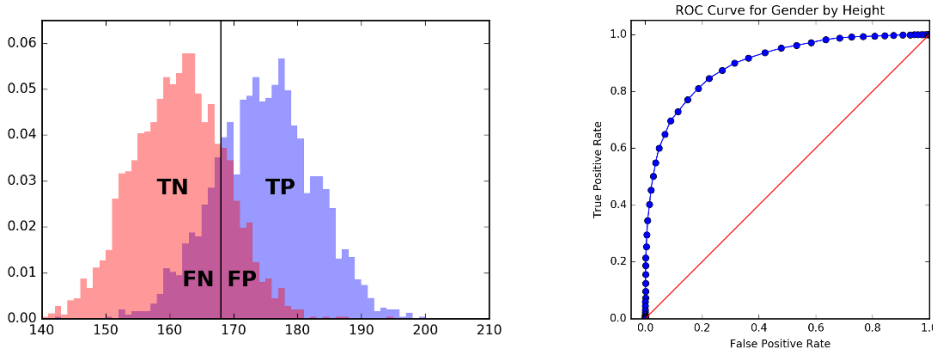

Figure 7.6: The ROC curve helps us select the best threshold to use in a classifier, by displaying the trade-off between true positives and false positive at every possible setting. The monkey ROCs the main diagonal here.

confidence is above a given threshold. Patients whose test scores are near the boundary would generally prefer a diagnosis of "borderline result" to "you've got cancer," particularly if the classifier is not really confident in its decision.

Our precision and recall statistics must be reconsidered to properly accommodate the new indeterminate class. There is no need to change the precision formula: we evaluate only on the instances we call positive. But the denominator for recall must explicitly account for all elements we refused to label. Assuming we are accurate in our confidence measures, precision will increase at the expense of recall.

7.4.2 Receiver-Operator Characteristic (ROC) Curves

Many classifiers come with natural knobs that you can tweak to alter the trade-off between precision and recall. For example, consider systems which compute a numerical score reflecting "in classness," perhaps by assessing how much the given test sample looks like cancer. Certain samples will score more positively than others. But where do we draw the line between positive and negative?

If our "in classness" score is accurate, then it should generally be higher for positive items than negative ones. The positive examples will define a different score distribution than the negative instances, as shown in Figure 7.6 (left). It would be great if these distributions were completely disjoint, because then there would be a score threshold t such that all instances with scores $\geq t$ are positive and all $< t$ are negative. This would define a perfect classifier.

But it is more likely that the two distributions will overlap, to at least some degree, turning the problem of identifying the best threshold into a judgment call based on our relative distaste towards false positives and false negatives.

The *Receiver Operating Characteristic* (ROC) curve provides a visual representation of our complete space of options in putting together a classifier. Each

point on this curve represents a particular classifier threshold, defined by its false positive and false negative rates.[2] These rates are in turn defined by the count of errors divided by the total number of positives in the evaluation data, and perhaps multiplied by one hundred to turn into percentages.

Consider what happens as we sweep our threshold from left to right over these distributions. Every time we pass over another example, we either increase the number of true positives (if this example was positive) or false positives (if this example was in fact a negative). At the very left, we achieve true/false positive rates of 0%, since the classifier labeled nothing as positive at that cutoff. Moving as far to the right as possible, all examples will be labeled positively, and hence both rates become 100%. Each threshold in between defines a possible classifier, and the sweep defines a staircase curve in true/false positive rate space taking us from (0%,0%) to (100%,100%).

Suppose the score function was defined by a monkey, i.e. an arbitrary random value for each instance. Then as we sweep our threshold to the right, the label of the next example should be positive or negative with equal probability. Thus we are equally likely to increase our true positive rate as false, and the ROC curve should cruise along the main diagonal.

Doing better than the monkey implies an ROC curve that lies above the diagonal. The best possible ROC curve shoots up immediately from (0%,0%) to (0%,100%), meaning it encounters all positive instances before any negative ones. It then steps to the right with each negative example, until it finally reaches the upper right corner.

The *area under the ROC curve* (AUC) is often used as a statistic measuring the quality of scoring function defining the classifier. The best possible ROC curve has an area of $100\% \times 100\% \rightarrow 1$, while the monkey's triangle has an area of 1/2. The closer the area is to 1, the better our classification function is.

7.4.3 Evaluating Multiclass Systems

Many classification problems are non-binary, meaning that they must decide among more than two classes. Google News has separate sections for U.S. and world news, plus business, entertainment, sports, health, science, and technology. Thus the article classifier which governs the behavior of this site must assign each article a label from eight different classes.

The more possible class labels you have, the harder it is to get the classification right. The expected accuracy of a classification monkey with d labels is $1/d$, so the accuracy drops rapidly with increased class complexity.

This makes properly evaluating multiclass classifiers a challenge, because low success numbers get disheartening. A better statistic is the *top-k success rate*, which generalizes accuracy for some specific value of $k \geq 1$. How often was the right label among the top k possibilities?

This measure is good, because it gives us partial credit for getting close to the right answer. How close is good enough is defined by the parameter k. For

[2]The strange name for this beast is a legacy of its original application, in tuning the performance of radar systems.

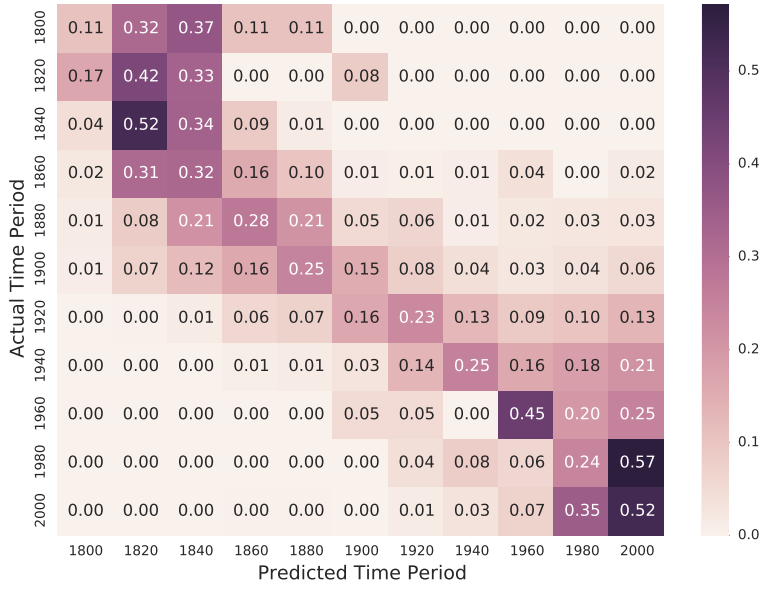

Figure 7.7: Confusion matrix for a document dating system: the main diagonal reflects accurate classification.

$k = 1$, this reduces to accuracy. For $k = d$, *any* possible label suffices, and the success rate is 100% by definition. Typical values are 3, 5, or 10: high enough that a good classifier should achieve an accuracy above 50% and be visibly better than the monkey. But not too much better, because an effective evaluation should leave us with substantial room to do better. In fact, it is a good practice to compute the top k rate for all k from 1 to d, or at least high enough that the task becomes easy.

An even more powerful evaluation tool is the *confusion matrix* C, a $d \times d$ matrix where $C[x, y]$ reports the number (or fraction) of instances of class x which get labeled as class y.

How do we read a confusion matrix, like the one shown in Figure 7.7? It is taken from the evaluation environment we built to test a document dating classifier, which analyzes texts to predict the period of authorship. Such document dating will be the ongoing example on evaluation through the rest of this chapter.

The most important feature is the main diagonal, $C[i, i]$, which counts how many (or what fraction of) items from class i were correctly labeled as class i. We hope for a heavy main diagonal in our matrix. Ours in a hard task, and Figure 7.7 shows a strong but not perfect main diagonal. There are several places where documents are more frequently classified in the neighboring period than the correct one.

But the most interesting features of the confusion matrix are the large counts

$C[i,j]$ that do *not* lie along the main diagonal. These represent commonly confused classes. In our example, the matrix shows a distressingly high number of documents (6%) from 1900 classified as 2000, when none are classified as 1800. Such asymmetries suggest directions to improve the classifier.

There are two possible explanations for class confusions. The first is a bug in the classifier, which means that we have to work harder to make it distinguish i from j. But the second involves humility, the realization that classes i and j may overlap to such a degree that it is ill-defined what the right answer should be. Maybe writing styles don't really change that much over a twenty-year period?

In the Google News example, the lines between the science and technology categories is very fuzzy. Where should an article about commercial space flights go? Google says science, but I say technology. Frequent confusion might suggest merging the two categories, as they represent a difference without a distinction.

Sparse rows in the confusion matrix indicate classes poorly represented in the training data, while sparse columns indicate labels which the classifier is reluctant to assign. Either indication is an argument that perhaps we should consider abandoning this label, and merge the two similar categories.

The rows and columns of the confusion matrix provide analogous performance statistics to those of Section 7.4.1 for multiple classes, parameterized by class. Precision$_i$ is the fraction of all items declared class i that were in fact of class i:

$$precision_i = C[i,i] / \sum_{j=1}^{d} C[j,i].$$

Recall$_i$ is the fraction of all members of class i that were correctly identified as such:

$$recall_i = C[i,i] / \sum_{j=1}^{d} C[i,j].$$

7.4.4 Evaluating Value Prediction Models

Value prediction problems can be thought of as classification tasks, but over an infinite number of classes. However, there are more direct ways to evaluate regression systems, based on the distance between the predicted and actual values.

Error Statistics

For numerical values, *error* is a function of the difference between a forecast $y' = f(x)$ and the actual result y. Measuring the performance of a value prediction system involves two decisions: (1) fixing the specific individual error function, and (2) selecting the statistic to best represent the full error distribution.

The primary choices for the individual error function include:

- *Absolute error*: The value $\Delta = y' - y$ has the virtue of being simple and symmetric, so the sign can distinguish the case where $y' > y$ from

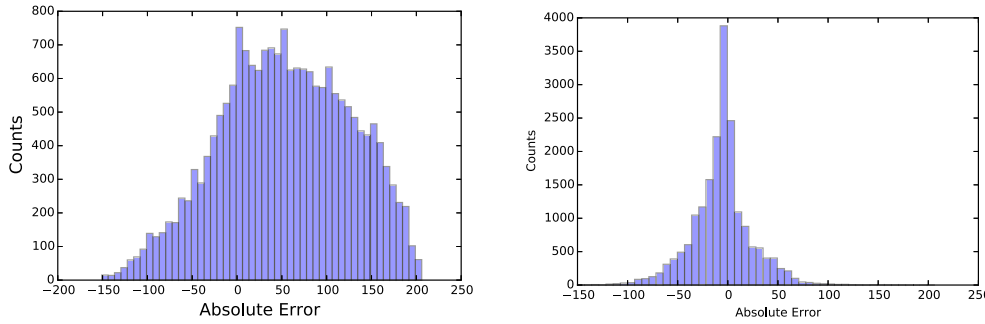

Figure 7.8: Error distribution histograms for random (left) and naive Bayes classifiers predicting the year of authorship for documents (right).

$y > y'$. The problem comes in aggregating these values into a summary statistic. Do offsetting errors like -1 and 1 mean that the system is perfect? Typically the absolute value of the error is taken to obliterate the sign.

- *Relative error*: The absolute magnitude of error is meaningless without a sense of the units involved. An absolute error of 1.2 in a person's predicted height is good if it is measured in millimeters, but terrible if measured in miles.

  Normalizing the error by the magnitude of the observation produces a unit-less quantity, which can be sensibly interpreted as a fraction or (multiplied by 100%) as a percentage: $\epsilon = (y - y')/y$. Absolute error weighs instances with larger values of y as more important than smaller ones, a bias corrected when computing relative errors.

- *Squared error*: The value $\Delta^2 = (y' - y)^2$ is always positive, and hence these values can be meaningfully summed. Large errors values contribute disproportionately to the total when squaring: Δ^2 for $\Delta = 2$ is four times larger than Δ^2 for $\Delta = 1$. Thus outliers can easily come to dominate the error statistic in a large ensemble.

It is a very good idea to plot a histogram of the absolute error distribution for any value predictor, as there is much you can learn from it. The distribution *should* be symmetric, and centered around zero. It *should* be bell-shaped, meaning small errors are more common than big errors. And extreme outliers *should* be rare. If any of the conditions are wrong, there is likely a simple way to improve the forecasting procedure. For example, if it is not centered around zero, adding an constant offset to all forecasts will improve the consensus results.

Figure 7.8 presents the absolute error distributions from two models for predicting the year of authorship of documents from their word usage distribution. On the left, we see the error distribution for the monkey, randomly guessing a

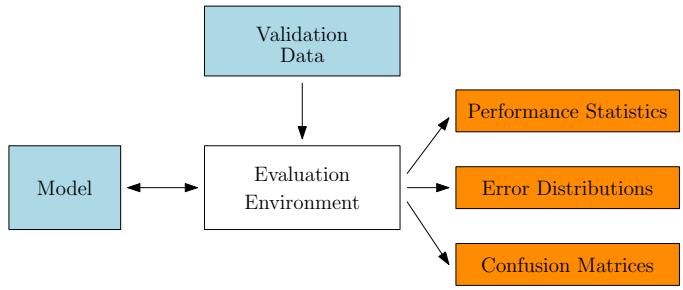

Figure 7.9: Block diagram of a basic model evaluation environment.

year from 1800 to 2005. What do we see? The error distribution is broad and bad, as we might have expected, but also asymmetric. Far more documents produced positive errors than negative ones. Why? The test corpus apparently contained more modern documents than older ones, so $(year - monkey\_year)$ is more often positive than negative. Even the monkey can learn something from seeing the distribution.

In contrast, Figure 7.8 (right) presents the error distribution for our naive Bayes classifier for document dating. This looks much better: there is a sharp peak around zero, and much narrower tails. But the longer tail now resides to the left of zero, telling us that we are still calling a distressing number of very old documents modern. We need to examine some of these instances, to figure out why that is the case.

We need a summary statistic reducing such error distributions to a single number, in order to compare the performance of different value prediction models. A commonly-used statistic is *mean squared error* (MSE), which is computed

$$MSE(Y, Y') = \frac{1}{n}\sum_{i=1}^{n}(y_i' - y_i)^2$$

Because it weighs each term quadratically, outliers have a disproportionate effect. Thus median squared error might be a more informative statistic for noisy instances.

Root mean squared (RMSD) error is simply the square root of mean squared error:

$$RMSD(\Theta) = \sqrt{MSE(Y, Y')}.$$

The advantage of RMSD is that its magnitude is interpretable on the same scale as the original values, just as standard deviation is a more interpretable quantity than variance. But this does not eliminate the problem that outlier elements can substantially skew the total.

	Dataset	Method	MAE	MedAE	Acc
0	NYTimes	Random	73.335463	65.0	0.004895
1	COHA_Fiction_100	Random	79.865017	72.0	0.005287
2	COHA_Fiction_500	Random	80.505849	74.0	0.003825
3	COHA_Fiction_1000	Random	80.604837	72.0	0.003825
4	COHA_Fiction_2000	Random	79.845332	72.0	0.005737
5	COHA_News_100	Random	66.539239	59.0	0.005461
6	COHA_News_500	Random	66.267091	59.0	0.005461
7	COHA_News_1000	Random	66.077670	57.5	0.004956
8	COHA_News_2000	Random	66.225526	58.0	0.005057

	Dataset	Method	MAE	MedAE	Acc
0	NYTimes	NB	21.306301	14	0.029728
1	COHA_Fiction_100	NB	32.302025	22	0.041732
2	COHA_Fiction_500	NB	25.428234	14	0.050056
3	COHA_Fiction_1000	NB	23.493926	13	0.053656
4	COHA_Fiction_2000	NB	22.493363	12	0.054781
5	COHA_News_100	NB	19.384001	14	0.030845
6	COHA_News_500	NB	16.657565	12	0.034891
7	COHA_News_1000	NB	16.282261	12	0.035093
8	COHA_News_2000	NB	16.220065	12	0.035599

Figure 7.10: Evaluation environment results for predicting the year of authorship for documents, comparing the monkey (left) to a naive Bayes classifier (right).

7.5 Evaluation Environments

A substantial part of any data science project revolves around building a reasonable evaluation environment. In particular, you need a *single-command program* to run your model on the evaluation data, and produce plots/reports on its effectiveness, as shown in Figure 7.9.

Why single command? If it is not easy to run, you won't try it often enough. If the results are not easy to read and interpret, you will not glean enough information to make it worth the effort.

The input to an evaluation environment is a set of instances with the associated output results/labels, plus a model under test. The system runs the model on each instance, compares each result against this gold standard, and outputs summary statistics and distribution plots showing the performance it achieved on this test set.

A good evaluation system has the following properties:

- It produces error distributions in addition to binary outcomes: how close your prediction was, not just whether it was right or wrong. Recall Figure 7.8 for inspiration.
- It produces a report with multiple plots about several different input distributions automatically, to read carefully at your leisure.
- It outputs the relevant summary statistics about performance, so you can quickly gauge quality. Are you doing better or worse than last time?

As an example, Figure 7.10 presents the output of our evaluation environment for the two document-dating models presented in the previous section. Recall that the task is to predict the year of authorship of a given document from word usage. What is worth noting?

- *Test sets broken down by type*: Observe the the evaluation environment partitioned the inputs into nine separate subsets, some news and some fiction, and of lengths from 100 to 2000 words. Thus at a glance we could see separately how well we do on each.

- *Logical progressions of difficulty*: It is obviously harder to make age determination from shorter documents than longer ones. By separating the harder and smaller cases, we better understand our source of errors. We see a big improvement in naive Bayes as we move from 100 to 500 words, but these gains saturate before 2000 words.

- *Problem appropriate statistics*: We did not print out every possible error metric, only mean and median absolute error and accuracy (how often did we get the year exactly right?). These are enough for us to see that news is easier than fiction, that our model is much better than the monkey, and that our chances of identifying the actual year correctly (measured by accuracy) are still too small for us to worry about.

This evaluation gives us the information we need to see how we are doing, without overwhelming us with numbers that we won't ever really look at.

7.5.1 Data Hygiene for Evaluation

An evaluation is only meaningful when you don't fool yourself. Terrible things happen when people evaluate their models in an undisciplined manner, losing the distinction between training, testing, and evaluation data.

Upon taking position of a data set with the intention of building a predictive model, your first operation should be to partition the input into three parts:

- *Training data*: This is what you are completely free to play with. Use it to study the domain, and set the parameters of your model. Typically about 60% of the full data set should be devoted to training.

- *Testing data*: Comprising about 20% of the full data set, this is what you use to evaluate how good your model is. Typically, people experiment with multiple machine learning approaches or basic parameter settings, so testing enables you to establish the relative performance of all these different models for the same task.

  Testing a model usually reveals that it isn't performing as well as we would like, thus triggering another cycle of design and refinement. Poor performance on test data relative to how it did on the training data suggests a model which has been overfit.

- *Evaluation data*: The final 20% of the data should be set aside for a rainy day: to confirm the performance of the final model right before it goes into production. This works only if you never opened the evaluation data until it was really needed.

The reason to enforce these separations should be obvious. Students would do much better on examinations if they were granted access to the answer key in advance, because they would know exactly what to study. But this would not reflect how much they actually had learned. Keeping testing data separate from training enforces that the tests measure something important about what the model understands. And holding out the final evaluation data to use only after the model gets stable ensures that the specifics of the test set have not leaked into the model through repeated testing iterations. The evaluation set serves as out-of-sample data to validate the final model.

In doing the original partitioning, you must be careful not to create undesirable artifacts, or destroy desirable ones. Simply partitioning the file in the order it was given is dangerous, because any structural difference between the populations of the training and testing corpus means that the model will not perform as well as it should.

But suppose you were building a model to predict future stock prices. It would be dangerous to randomly select 60% of the samples over all history as the training data, instead of all the samples over the first 60% of time. Why? Suppose your model "learned" which would be the up and down days in the market from the training data, and then used this insight to make virtual predictions for other stocks on these same days. This model would perform far better in testing than in practice. Proper sampling techniques are quite subtle, and discussed in Section 5.2.

It is essential to maintain the veil of ignorance over your evaluation data for as long as possible, because you spoil it as soon as you use it. Jokes are never funny the second time you hear them, after you already know the punchline. If you do wear out the integrity of your testing and evaluation sets, the best solution is to start from fresh, out-of-sample data, but this is not always available. Otherwise, randomly re-partition the full data set into fresh training, testing, and evaluation samples, and retrain all of your models from scratch to reboot the process. But this should be recognized as an unhappy outcome.

7.5.2 Amplifying Small Evaluation Sets

The idea of rigidly partitioning the input into training, test, and evaluation sets makes sense only on large enough data sets. Suppose you have 100,000 records at your disposal. There isn't going to be a qualitative difference between training on 60,000 records instead of 100,000, so it is better to facilitate a rigorous evaluation.

But what if you only have a few dozen examples? As of this writing, there have been only 45 U.S. presidents, so any analysis you can do on them represents very small sample statistics. New data points come very slowly, only once every four years or so. Similar issues arise in medical trials, which are very expensive to run, potentially yielding data on well under a hundred patients. Any application where we must pay for human annotation means that we will end up with less data for training than we might like.

What can you do when you cannot afford to give up a fraction of your data

for testing? *Cross-validation* partitions the data into k equal-sized chunks, then trains k distinct models. Model i is trained on the union of all blocks $x \neq i$, totaling $(k-1)/k$th of the data, and tested on the held out ith block. The average performance of these k classifiers stands in as the presumed accuracy for the full model.

The extreme case here is *leave one out cross-validation*, where n distinct models are each trained on different sets of $n-1$ examples, to determine whether the classifier was good or not. This maximizes the amount of training data, while still leaving something to evaluate against.

A real advantage of cross validation is that it yields a standard deviation of performance, not only a mean. Each classifier trained on a particular subset of the data will differ slightly from its peers. Further, the test data for each classifier will differ, resulting in different performance scores. Coupling the mean with the standard deviation and assuming normality gives you a performance distribution, and a better idea of how well to trust the results. This makes cross-validation very much worth doing on *large* data sets as well, because you can afford to make several partitions and retrain, thus increasing confidence that your model is good.

Of the k models resulting from cross validation, which should you pick as your final product? Perhaps you could use the one which performed best on its testing quota. But a better alternative is to retrain on *all* the data and trust that it will be at least as good as the less lavishly trained models. This is not ideal, but if you can't get enough data then you must do the best with what you've got.

Here are a few other ideas that can help to amplify small data sets for training and evaluation:

- *Create negative examples from a prior distribution*: Suppose one wanted to build a classifier to identify who would be qualified to be a candidate for president. There are very few real examples of presidential candidates (positive instances), but presumably the elite pool is so small that a random person will almost certainly be unqualified. When positive examples are rare, all others are very likely negative, and can be so labeled to provide training data as necessary.

- *Perturb real examples to create similar but synthetic ones*: A useful trick to avoid overfitting creates new training instances by adding random noise to distort labeled examples. We then preserve the original outcome label with the new instance.

  For example, suppose we are trying to train an optical character recognition (OCR) system to recognize the letters of some alphabet in scanned pages. An expensive human was originally given the task of labeling a few hundred images with the characters that were contained in them. We can amplify this to a few million images by adding noise at random, and rotating/translating/dilating the region of interest. A classifier trained on

this synthetic data should be far more robust than one restricted to the original annotated data.

- *Give partial credit when you can*: When you have fewer training/testing examples than you want, you must squeeze as much information from each one as possible.

  Suppose that our classifier outputs a value measuring its confidence in its decision, in addition to the proposed label. This confidence level gives us additional resolution with which to evaluate the classifier, beyond just whether it got the label right. It is a bigger strike against the classifier when it gets a confident prediction wrong, than it is on an instance where it thought the answer was a tossup. On a presidential-sized problem, I would trust a classifier that got 30 right and 15 wrong with accurate confidence values much more than one with 32 right and 13 wrong, but with confidence values all over the map.

7.6 War Story: 100% Accuracy

The two businessmen looked a little uncomfortable at the university, out of place with their dark blue suits to our shorts and sneakers. Call them Pablo and Juan. But they needed us to make their vision a reality.

"The business world still works on paper," Pablo explained. He was the one in the darker suit. "We have a contract to digitize all of Wall Street's financial documents that are still printed on paper. They will pay us a fortune to get a computer to do the scanning. Right now they hire people to type each document in three times, just to make sure they got it absolutely right."

It sounded exciting, and they had the resources to make it happen. But there was one caveat. "Our system cannot make any errors. It can say 'I don't know' sometimes. But whenever it calls a letter it has to be 100% correct."

"No problem." I told them. "Just let me say *I don't know* 100% of the time, and I can design a system to meet your specification."

Pablo frowned. "But that will cost us a fortune. In the system we want to build, images of the *I don't knows* will go to human operators for them to read. But we can't afford to pay them to read everything."

My colleagues and I agreed to take the job, and in time we developed a reasonable OCR system from scratch. But one thing bothered me.

"These Wall Street guys who gave you the money are smart, aren't they?," I asked Pablo one day.

"Smart as a whip," he answered.

"Then how could they possibly believe it when you said you could build an OCR system that was 100% accurate."

"Because they thought that I had done it before," he said with a laugh.

It seems that Pablo's previous company had built a box that digitized price data from the television monitors of the time. In this case, the letters were exact patterns of bits, all written in exactly the same font and the same size.

The TV signal was digital, too, with no error at all from imaging, imperfectly-formed blobs of printers ink, dark spots, or folds in the paper. It was trivial to test for an exact match between a perfect pattern of bits (the image) and another perfect pattern of bits (the character in the device's font), since there is no source of uncertainty. But this problem had nothing to do with OCR, even if both involved reading letters.

Our reasonable OCR system did what it could with the business documents it was given, but of course we couldn't get to 100%. Eventually, the Wall Street guys took their business back to the Philippines, where they paid three people to type in each document and voted two out of three if there was any disagreement.

We shifted direction and got into the business of reading handwritten survey forms submitted by consumers, lured by the promise of grocery store coupons. This problem was harder, but the stakes not as high: they were paying us a crummy $0.22 a form and didn't expect perfection. Our competition was an operation that used prison labor to type in the data. We caught a break when one of those prisoners sent a threatening letter to an address they found on a survey form, which then threw the business to us. But even our extensive automation couldn't read these forms for less than $0.40 a pop, so the contract went back to prison after we rolled belly up.

The fundamental lesson here is that no pattern recognition system for any reasonable problem will bat 100% all the time. The only way never to be wrong is to never make a prediction. Careful evaluation is necessary to measure how well your system is working and where it is making mistakes, in order to make the system better.

7.7 Simulation Models

There is an important class of first-principle models which are not primarily data driven, yet which prove very valuable for understanding widely diverse phenomena. *Simulations* are models that attempt to replicate real-world systems and processes, so we can observe and analyze their behavior.

Simulations are important for demonstrating the validity of our understanding of a system. A simple simulation which captures much of the behavioral complexity of a system must explain how it works, by Occam's razor. Famous physicist Richard Feynman said, "What I cannot create, I do not understand." What you cannot simulate, and get some level of accuracy in the observed results, you do not understand.

Monte Carlo simulations use random numbers to synthesize alternate realities. Replicating an event millions of times under slightly perturbed conditions permits us to generate a probability distribution on the set of outcomes. This was the idea behind permutation tests for statistical significance. We also saw (in Section 5.5.2) that random coin flips could stand in for when a batter got hits or made out, so we could simulate an arbitrary number of careers and observe what happened over the course of them.

The key to building an effective Monte Carlo simulation is designing an

appropriate discrete event model. A new random number is used by the model to replicate each decision or event outcome. You may have to decide whether to go left or go right in a transportation model, so flip a coin. A health or insurance model may have to decide whether a particular patient will have a heart attack today, so flip an appropriately weighted coin. The price of a stock in a financial model can either go up or down at each tick, which again can be a coin flip. A basketball player will hit or miss a shot, with a likelihood that depends upon their shooting skill and the quality of their defender.

The accuracy of such a simulation rests on the probabilities that you assign to heads and tails. This governs how often each outcome occurs. Obviously, you are not restricted to using a fair coin, meaning 50/50. Instead, the probabilities need to reflect assumptions of the likelihood of the event given the state of the model. These parameters are often set using statistical analysis, by observing the distribution of events as they occurred in the data. Part of the value of Monte Carlo simulations is that they let us play with alternate realities, by changing certain parameters and seeing what happens.

A critical aspect of effective simulation is evaluation. Programming errors and modeling inadequacies are common enough that no simulation can be accepted on faith. The key is to hold back one or more classes of observations of the system from direct incorporation in your model. This provides behavior that is out of sample, so we can compare the distribution of results from the simulation with these observations. If they don't jibe, your simulation is just jive. Don't let it go live.

7.8 War Story: Calculated Bets

Where there is gambling there is money, and where there is money there will be models. During our family trips to Florida as a kid, I developed a passion for the sport of jai-alai. And, as I learned how to build mathematical models as a grown-up, I grew obsessed with developing a profitable betting system for the sport.

Jai-alai is a sport of Basque origin where opposing players or teams alternate hurling a ball against the wall and catching it, until one of them finally misses and loses the point. The throwing and catching is done with an enlarged basket or *cesta*, the ball or *pelota* is made of goat skin and hard rubber, and the wall is of granite or concrete; ingredients which lead to fast and exciting action captured in Figure 7.11. In the United States, jai-alai is most associated with the state of Florida, which permits gambling on the results of matches.

What makes jai-alai of particular interest is its unique and *very* peculiar scoring system. Each jai-alai match involves eight players, named 1 through 8 to reflect their playing order. Each match starts with players 1 and 2 playing, and the rest of the players waiting patiently in line. All players start the game with zero points each. Each point in the match involves two players; one who will win and one who will lose. The loser will go to the end of the line, while the winner will add to his point total and await the next point, until he has

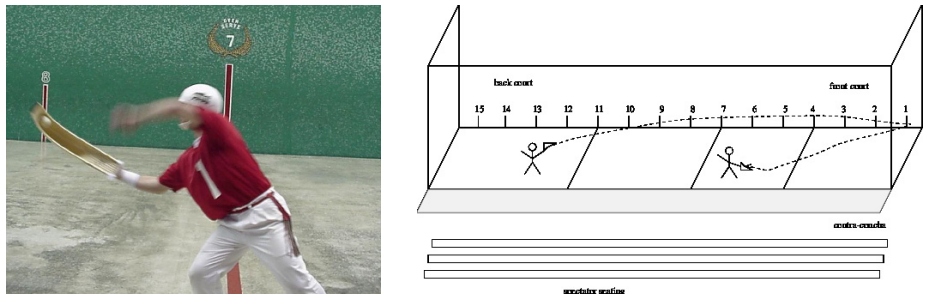

Figure 7.11: Jai-alai is a fast, exciting ball game like handball, but you can bet on it.

accumulated enough points to claim the match.

It was obvious to me, even as a kid, that this scoring system would not be equally fair to all the different players. Starting early in the queue gave you more chances to play, and even a kludge they added to double the value of points later in the match couldn't perfectly fix it. But understanding the strength of these biases could give me an edge in betting.

My quest to build a betting system for jai-alai, started by simulating this very peculiar scoring system. A jai-alai match consists of a sequence of discrete events, described by the following flow structure:

Initialize the current players to 1 and 2.
Initialize the queue of players to $\{3, 4, 5, 6, 7, 8\}$.
Initialize the point total for each player to zero.
So long as the current winner has less than 7 points:
    Pick a random number to decide who wins the next point.
    Add one (or if beyond the seventh point, two) to the
        total of the simulated point winner.
    Put the simulated point loser at the end of the queue.
    Get the next player off the front of the queue.
End *So long as*.
Identify the current point winner as the winner of the match.

The only step here which needs more elaboration is that of simulating a point between two players. If the purpose of our simulation is to see how biases in the scoring system affect the outcome of the match, it makes the most sense to consider the case in which all players are equally skillful. To give every player a 50/50 chance of winning each point he is involved in, we can flip a simulated coin to determine who wins and who loses.

I implemented the jai-alai simulation in my favorite programming language, and ran it on 1,000,000 jai-alai games. The simulation produced a table of statistics, telling me how often each betting outcome paid off, assuming that all the players were equally skillful. Figure 7.12 reports the number of simulated

	Simulated		Observed	
Position	Wins	% Wins	Wins	% Wins
1	162675	16.27%	1750	14.1%
2	162963	16.30%	1813	14.6%
3	139128	13.91%	1592	12.8%
4	124455	12.45%	1425	11.5%
5	101992	10.20%	1487	12.0%
6	102703	10.27%	1541	12.4%
7	88559	8.86%	1370	11.1%
8	117525	11.75%	1405	11.3%
	1,000,000	100.00%	12,383	100.0%

Figure 7.12: Win biases observed in the jai-alai simulations match well with results observed in actual matches.

wins for each of the eight starting positions. What insights can we draw from this table?

- Positions 1 and 2 have a substantial advantage over the rest of the field. Either of the initial players are almost twice as likely to come first, second, or third than the poor shlub in position 7.

- Positions 1 and 2 win at essentially the same frequency. This is as it should be, since both players start the game on the court instead of in the queue. The fact that players 1 and 2 have very similar statistics increases our confidence in the correctness of the simulation.

- Positions 1 and 2 do not have *identical* statistics because we simulated "only" one million games. If you flip a coin a million times, it almost certainly won't come up exactly half heads and half tails. However, the *ratio* of heads to tails should keep getting closer to 50/50 the more coins we flip.

  The simulated gap between players 1 and 2 tells us something about the *limitations* on the accuracy of our simulation. We shouldn't trust any conclusions which depends upon such small differences in the observed values.

To validate the accuracy of the simulation, we compared our results to statistics on the actual outcomes of over 12,000 jai-alai matches, also in Figure 7.12. The results basically agree with the simulation, subject to the limits of the small sample size. Post positions 1 and 2 won most often in real matches, and position 7 least often.

Now we knew the probability that each possible betting opportunity in jai-alai paid off. Were we now ready to start making money? Unfortunately not.

Even though we have established that post position is a major factor in determining the outcome of jai-alai matches, perhaps the dominant one, we still had several hurdles to overcome before we could bet responsibly:

- *The impact of player skills*: Obviously, a good player is more likely to win than a bad one, regardless of their post positions. It is clear that a better model for predicting the outcome of jai-alai matches would factor relative skills into the queuing model.
- *The sophistication of the betting public*: Many people had noticed the impact of post-position bias before I did. Indeed, data analysis revealed the jai-alai betting public had largely factored the effect of post position in the odds. Fortunately for us, however, largely did not mean completely.
- *The house cut* – Frontons keep about 20% of the betting pool as the house percentage, and thus we had to do much better then the average bettor just to break even.

My simulation provided information on which outcomes were most likely. It did not by itself identify which are the best bets. A good bet depends both upon the likelihood of the event occurring and the payoff when it occurs. Payoffs are decided by the rest of the betting public. To find the best bets to make, we had to work a lot harder:

- We had to analyze past match data to determine who were the better players. Once we knew who was better, we could bias the simulated coin tosses in their favor, to make our simulation more accurate for each individual match.
- We had to analyze payoff data to build a model of *other* bettor's preferences. In jai-alai, you are betting against the public, so you need to be able to model their thinking in order to predict the payoffs for a particular bet.
- We had to model the impact of the house's cut on the betting pool. Certain bets, which otherwise might have been profitable, go into the red when you factor in these costs.

The bottom line is that we did it, with 544% returns on our initial stake. The full story of our gambling system is reported in my book *Calculated Bets* [Ski01]. Check it out: I bet you will like it. It is fun reading about successful models, but even more fun to build them.

7.9 Chapter Notes

Silver [Sil12] is an excellent introduction to the complexities of models and forecasting in a variety of domains. Textbooks on mathematical modeling issues include Bender [Ben12] and Giordano [GFH13].

The Google Flu Trends project is an excellent case study in both the power and limitation of big data analysis. See Ginsberg et al. [GMP+09] for the original description, and Lazer et al. [LKKV14] for a fascinating post-mortem on how it all went wrong.

Technical aspects of the OCR system presented in Section 7.6 is reported in Sazaklis et. al. [SAMS97]. The work on year of authorship detection (and associated evaluation environment example) is from my students Vivek Kulkarni, Parth Dandiwala, and Yingtao Tian [KTDS17].

7.10 Exercises

Properties of Models

7-1. *[3]* Quantum physics is much more complicated than Newtonian physics. Which model passes the Occam's Razor test, and why?

7-2. *[5]* Identify a set of models of interest. For each of these, decide which properties these models have:

(a) Are they discrete or continuous?

(b) Are they linear or non-linear?

(c) Are they blackbox or descriptive?

(d) Are they general or ad hoc?

(e) Are they data driven or first principle?

7-3. *[3]* Give examples of first-principle and data-driven models used in practice.

7-4. *[5]* For one or more of the following *The Quant Shop* challenges, discuss whether principled or data-driven models seem to be the more promising approach:

- *Miss Universe.*
- *Movie gross.*
- *Baby weight.*
- *Art auction price.*
- *Snow on Christmas.*
- *Super Bowl/college champion.*
- *Ghoul pool.*
- *Future gold/oil price.*

7-5. *[5]* For one or more of the following *The Quant Shop* challenges, partition the full problem into subproblems that can be independently modeled:

- *Miss Universe.*
- *Movie gross.*
- *Baby weight.*
- *Art auction price.*

- *Snow on Christmas.*
- *Super Bowl/college champion.*
- *Ghoul pool.*
- *Future gold/oil price.*

Evaluation Environments

7-6. *[3]* Suppose you build a classifier that answers *yes* on every possible input. What precision and recall will this classifier achieve?

7-7. *[3]* Explain what precision and recall are. How do they relate to the ROC curve?

7-8. *[5]* Is it better to have too many false positives, or too many false negatives? Explain.

7-9. *[5]* Explain what overfitting is, and how you would control for it.

7-10. *[5]* Suppose $f \leq 1/2$ is the fraction of positive elements in a classification. What is the probability p that the monkey should guess positive, as a function of f, in order to maximize the specific evaluation metric below? Report both p and the expected evaluation score the monkey achieves.

(a) Accuracy.

(b) Precision.

(c) Recall.

(d) F-score.

7-11. *[5]* What is cross-validation? How might we pick the right value of k for k-fold cross validation?

7-12. *[8]* How might we know whether we have collected enough data to train a model?

7-13. *[5]* Explain why we have training, test, and validation data sets and how they are used effectively?

7-14. *[5]* Suppose we want to train a binary classifier where one class is very rare. Give an example of such a problem. How should we train this model? What metrics should we use to measure performance?

7-15. *[5]* Propose baseline models for one or more of the following *The Quant Shop* challenges:

- *Miss Universe.*
- *Movie gross.*
- *Baby weight.*
- *Art auction price.*
- *Snow on Christmas.*
- *Super Bowl/college champion.*
- *Ghoul pool.*
- *Future gold/oil price.*

Implementation Projects

7-16. *[5]* Build a model to forecast the outcomes of one of the following types of betable events, and rigorously analyze it through back testing:

(a) Sports like football, basketball, and horse racing.

(b) Pooled bets involving multiple events, like soccer pools or the NCAA basketball tournament.

(c) Games of chance like particular lotteries, fantasy sports, and poker.

(d) Election forecasts for local and congressional elections.

(e) Stock or commodity price prediction/trading.

Rigorous testing will probably confirm that your models are not strong enough for profitable wagering, and this is 100% ok. Be honest: make sure that you are using fresh enough prices/odds to reflect betting opportunities which would still be available at the time you place your simulated bet. To convince me that your model is in fact genuinely profitable, send me a cut of the money and then I will believe you.

7-17. *[5]* Build a general model evaluation system in your favorite programming language, and set it up with the right data to assess models for a particular problem. Your environment should report performance statistics, error distributions and/or confusion matrices as appropriate.

Interview Questions

7-18. *[3]* Estimate prior probabilities for the following events:

(a) The sun will come up tomorrow.

(b) A major war involving your country will start over the next year.

(c) A newborn kid will live to be 100 years old.

(d) Today you will meet the person whom you will marry.

(e) The Chicago Cubs will win the World Series this year.

7-19. *[5]* What do we mean when we talk about the bias–variance trade-off?

7-20. *[5]* A test has a true positive rate of 100% and false positive rate of 5%. In this population 1 out of 1000 people have the condition the test identifies. Given a positive test, what is the probability this person actually has the condition?

7-21. *[5]* Which is better: having good data or good models? And how do you define good?

7-22. *[3]* What do you think about the idea of injecting noise into your data set to test the sensitivity of your models?

7-23. *[5]* How would you define and measure the predictive power of a metric?

Kaggle Challenges

7-24. Will a particular grant application be funded?
`https://www.kaggle.com/c/unimelb`

7-25. Who will win the NCAA basketball tournament?
`https://www.kaggle.com/c/march-machine-learning-mania-2016`

7-26. Predict the annual sales in a given restaurant.
`https://www.kaggle.com/c/restaurant-revenue-prediction`