A BRIEF PRIMER FOR BUILDING AND EVALUATING

ML MODELS

THE ARTISTS OF **DATA SCIENCE**

CURATED BY **HARPREET SAHOTA**

A PHILOSOPHY

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

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. Overfit models tend to perform extremely well on training data, but much less accurately on independent test data.

ALL MODELS ARE WRONG, BUT SOME MODELS ARE USEFUL. - GEORGE BOX

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.

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 t the training data as tightly as possible, were they allowed the freedom to do so.

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.

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.

The Artists of Data Science
with Harpreet Sahota

ABOUT THE CURATOR

Hi, I'm Harpreet!

I'm the host of T**he Artists of Data Science Podcast** - available wherever you listen to your favorite podcast! I'm also Head of Mentorship at **Data Science Dream Job** (check out this FREE training if you've tried everything you can but STILL can't land a job in data science).

I host two completely FREE open office hours every week, check them out!

Sunday's at 11am CST: http://bit.ly.comet-ml-oh

Friday's at 4:30pm CST: http://bit.ly/adsoh

Check me out on the Super Data Science Podcast!

Cheers!

TYPES OF MODELS

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.

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.

TYPES OF MODELS

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.

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.

TYPES OF MODELS

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. 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.

Data science is about science, and things that happen for understandable reasons. Models which ignore this are doomed to fail embarrassingly in certain circumstances.

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 test and evaluate your models.

TYPES OF MODELS

STOCHASTIC VS DETERMINISTIC MODELS

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 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 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. 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

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.

TYPES OF MODELS

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:

- The general state of the economy
- The company's balance sheet
- 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.

THE IMPORTANCE OF A BASELINE MODEL

A wise man once observed that **a broken clock is right twice a day**.

As machine learning practitioners 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 prediction 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.

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. **Only after you decisively beat your baselines can your models really be deemed effective.**

BASELINE MODELS FOR CLASSIFICATION

There are two common tasks for data science models: classification and regression.

In classification tasks, we are given a small set of possible labels for any given item: spam or not spam, man or woman, and 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. 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 $x_i \geq t$, and class 2 if otherwise. To find the best threshold t, we can test all n possible thresholds of the form $t_i = x_i + e$ 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**.

- **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.

BASELINE MODELS FOR REGRESSION

In regression problems, we are given a collection of feature-value pairs (xi; yi) to use to train a function F such that F(xi) = yi. 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: This powerful but simple-to use technique builds the best possible linear function. 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(tn; x) at time tn given feature set x and the observed values f0(ti) for 1 ≤ i ≤ n. But today's weather is a good guess for whether it will rain tomorrow. Similarly, the value of the previous observed value f0(tn1) is a reasonable forecast for time f(tn). 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.

EVALUATING MODELS

Congratulations! You have built a predictive model for classification or regression. 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.

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.

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.

EVALUATING CLASSIFERS

Evaluating a classifier means measuring how accurately our predicted labels match the ground truth in the evaluation set.

For the common case of 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 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**:

- **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.

EVALUATING CLASSIFIERS:

ACCURACY AND PRECISION

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.

Accuracy

The first statistic measures the accuracy of classifier, the ratio of the number of correct predictions over total predictions. **Accuracy = (TP + TN) / (TP + TN + FP + FN)**

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.

Precision

So 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: **TP / (TP + FP).**

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 instances, so the classifier achieves low true positives.

EVALUATING CLASSIFIERS:

RECALL AND F-SCORE

Recall

Recall measures how often you prove right on all positive instances: **TP/ (TP + FN).**

A **high recall implies that the classifier has few false negatives**. The easiest way to achieve this declares that everyone has is predicted as the positive class, as done by a sharp always answering yes.

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.

F1-score

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: 2* (precision * recall)/ (precision + 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 disproportionate 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.

EVALUATING CLASSIFIERS:

KEY TAKEAWAYS

- **Accuracy is a misleading statistic when the class sizes are substantially different.**

- **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**

- **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.

EVALUATING CLASSIFIERS:

AREA UNDER ROC CURVE

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

The AUC is one way to summarize the ROC curve into a single number, so that it can be compared easily and automatically. A good ROC curve has a lot of space under it (because the true positive rate shoots up to 100% very quickly). A bad ROC curve covers very little area. So high AUC is good, and low AUC is not so good.

Each point on this curve represents a particular classifier threshold, defined by its false positive and false negative rates.

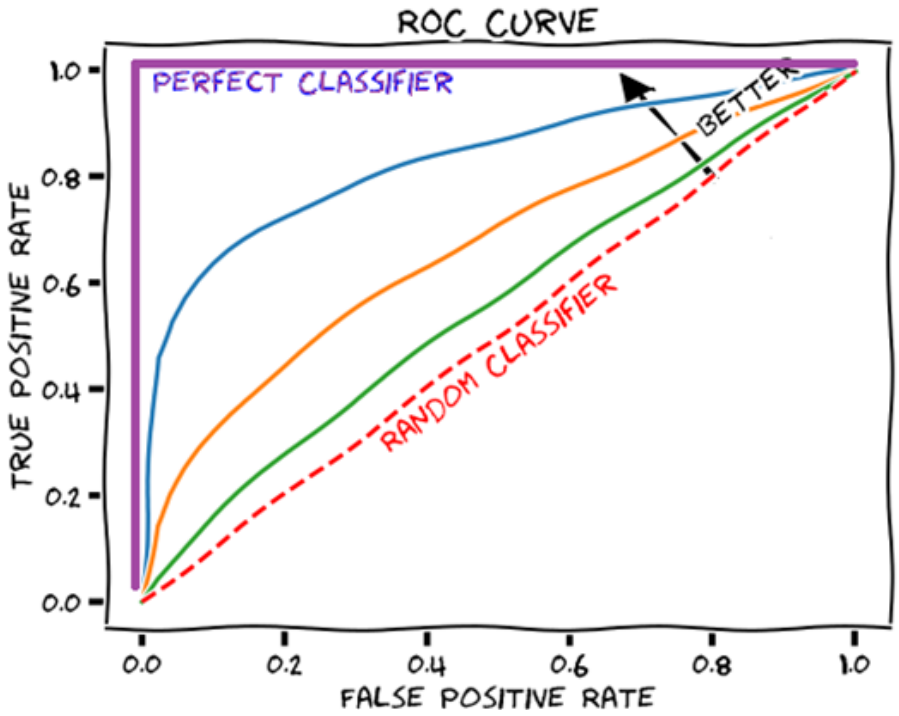
These rates are in turn defined by the count of errors divided by the total number of positives in the evaluation data, and perhaps multiplied by one hundred to turn into percentages

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 1, while the monkey's triangle (ie, left up to chance) has an area of 1/2.

The closer the area is to 1, the better our classification function is.

Note: I can't remember where I stole the below picture from. But it's nice.



EVALUATING CLASSIFIERS:

MULTICLASS PROBLEMS

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 $k = 1$, 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.

EVALUATING REGRESSORS

For numerical values, **error** is a function of the difference between a forecast y0 = f(x) and the actual result y.

Measuring the performance of a regression model involves two decisions: (1) fixing the specifc **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 difference between the predicted and actual value has the virtue of being simple and symmetric, so the sign can distinguish the case where predicted value > actual value from actual value > predicted value. 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 as a percentage: **relative error = (actual value - predicted value)/actual value.** Absolute error weighs instances with larger values of the ground truth as more important than smaller ones, a bias corrected when computing relative errors.

- **Squared error:** The value (predicted value - actual values)^2 is always positive, and hence these values can be meaningfully summed. Large errors values contribute disproportionately to the total when squaring. Thus outliers can easily come to dominate the error statistic in a large ensemble.

It's 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 oset to all forecasts will improve the consensus results.

EVALUATING REGRESSORS

We need a summary statistic reducing such error distributions to a single number, in order to compare the performance of different regression 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.

EVALUATION SYSTEM

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.
- 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?

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 conform 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.

I**t 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.

CROSS-VALIDATION

Hold-Out

Hold-out validation is simple. Assuming that all data points are i.i.d. (independently and identically distributed), we simply randomly hold out part of the data for validation. We train the model on the larger portion of the data and evaluate validation metrics on the smaller hold-out set.

Computationally speaking, hold-out validation is simple to program and fast to run. The downside is that it is less powerful statistically. The validation results are derived from a small subset of the data, hence its estimate of the generalization error is less reliable.

It is also difficult to compute any variance information or confidence intervals on a single dataset. Use hold-out validation when there is enough data such that a subset can be held out, and this subset is big enough to ensure reliable statistical estimates.

Cross-Validation

Cross-validation is another validation technique. It is not the only validation technique, and it is not the same as hyperparameter tuning.

So be careful not to get the three (the concept of model validation, cross-validation, and hyperparameter tuning) confused with each other. Cross-validation is simply a way of generating
training and validation sets for the process of hyperparameter tuning. Holdout validation, another validation technique, is also valid for hyperparameter tuning, and is in fact computationally much cheaper.

There are many variants of cross-validation. The most commonly used is k-fold cross-validation. In this procedure, we first divide the training dataset into k folds.

For a given hyperparameter setting, each of the k folds takes turns being the hold-out validation set; a model is trained on the rest of the k – 1 folds and measured on the held-out fold. The overall performance is taken to be the average of the performance on all k folds. Repeat this procedure for all of the hyperparameter settings that need to be evaluated, then pick the hyperparameters that resulted in the highest k-fold average.

CROSS-VALIDATION

Another variant of cross- validation is leave-one-out cross-validation. 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 model 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 model trained on a particular subset of the data will differ slightly from its peers.

The test data for each model 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 make 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 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 metric. 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.

CROSS-VALIDATION

Bootstrap and Jacknife

Bootstrap is a resampling technique. It generates multiple datasets by sampling from a single, original dataset. Each of the "new" datasets can be used to estimate a quantity of interest.

Since there are multiple datasets and therefore multiple estimates, one can also calculate things like the variance or a confidence interval for the estimate.

Bootstrap is closely related to cross-validation. It was inspired by another resampling technique called the jackknife, which is essentially leave-one-out cross-validation.

One can think of the act of dividing the data into k folds as a (very rigid) way of resampling the data without replacement; i.e., once a data point is selected for one fold, it cannot be selected again for another fold.

Bootstrap, on the other hand, resamples the data with replacement. Given a dataset containing N data points, bootstrap picks a data point uniformly at random, adds it to the bootstrapped set, puts
that data point back into the dataset, and repeats.

Why put the data point back?

A real sample would be drawn from the real distribution of the data. But we don't have the real distribution of the data.

All we have is one dataset that is supposed to represent the underlying distribution. This gives us an empirical distribution of data. Bootstrap simulates new samples by drawing from the empirical distribution. The data point must be put back, because otherwise the empirical distribution would change after each draw.

One way to use the bootstrapped dataset for validation is to train the model on the unique instances of the bootstrapped dataset and validate results on the rest of the unselected data.
The effects are very similar to what one would get from cross-validation.

Cross-validation is not the same as hyperparameter tuning. Cross-validation is a mechanism for generating training and validation splits. Hyperparameter tuning is the mechanism by which we select the best hyperparameters for a model; it might use cross-validation to evaluate the model.

HYPERPARAMETERS VS MODEL PARAMETERS

Cross-validation is not the same as hyperparameter tuning. Cross-validation is a mechanism for generating training and validation splits. Hyperparameter tuning is the mechanism by which we
select the best hyperparameters for a model; it might use cross-validation to evaluate the model.

First, let's define what a hyperparameter is, and how it is different from a normal nonhyper model parameter.

Machine learning models are basically mathematical functions that represent the relationship between different aspects of data. For instance, a linear regression model uses a line to
represent the relationship between "features" and "target."

The formula looks like this: $$w^T x = y$$

where x is a vector that represents features of the data and y is a scalar variable that represents the target (some numeric quantity that we wish to learn to predict).

This model assumes that the relationship between x and y is linear. The variable w is a weight vector that represents the normal vector for the line; it specifies the slope of the line. This is
what's known as a model parameter, which is learned during the training phase.

"Training a model" involves using an optimization procedure to determine the best model parameter that "fits" the data.

These are values that must be specified outside of the training procedure. Vanilla linear regression doesn't have any hyperparameters. But variants of linear regression do. Ridge regression and
lasso both add a regularization term to linear regression; the weight for the regularization term is called the regularization parameter.

Decision trees have hyperparameters such as the desired depth and number of leaves in the tree.

Support vector machines (SVMs) require setting a misclassification penalty term. Kernelized SVMs require setting kernel parameters like the width for radial basis function (RBF) kernels. The list goes on.

WHAT DO HYPERPARAMETERS DO?

A regularization hyperparameter controls the capacity of the model, i.e., how flexible the model is, how many degrees of freedom it has in fitting the data.

Proper control of model capacity can prevent overfitting, which happens when the model is too flexible, and the training process adapts too much to the training data, thereby losing
predictive accuracy on new test data. So a proper setting of the hyperparameters is important.

Another type of hyperparameter comes from the training process itself. Training a machine learning model often involves optimizing a loss function (the training metric).

A number of mathematical optimization techniques may be employed, some of them having parameters of their own.

For instance, stochastic gradient descent optimization requires a learning rate or a learning schedule. Some optimization methods require a convergence threshold.

Random forests and boosted decision trees require knowing the number of total trees (though this could also be classified as a type of regularization hyperparameter). These also need to be set to
reasonable values in order for the training process to find a good model.

Hyperparameter settings could have a big impact on the prediction accuracy of the trained model. Optimal hyperparameter settings often differ for different datasets. Therefore they should be tuned for each dataset.

Since the training process doesn't set the hyperparameters, there needs to be a meta process that tunes the hyperparameters.

This is what we mean by hyperparameter tuning.