DOI:10.1145/2347736.2347755

Tapping into the "folk knowledge" needed to advance machine learning applications.

BY PEDRO DOMINGOS

A Few Useful Things to Know About Machine Learning

MACHINE LEARNING SYSTEMS automatically learn programs from data. This is often a very attractive alternative to manually constructing them, and in the last decade the use of machine learning has spread rapidly throughout computer science and beyond. Machine learning is used in Web search, spam filters, recommender systems, ad placement, credit scoring, fraud detection, stock trading, drug design, and many other applications. A recent report from the McKinsey Global Institute asserts that machine learning (a.k.a. data mining or predictive analytics) will be the driver of the next big wave of innovation.[15] Several fine textbooks are available to interested practitioners and researchers (for example, Mitchell[16] and Witten et al.[24]). However, much of the "folk knowledge" that is needed to successfully develop machine learning applications is not readily available in them. As a result, many machine learning projects take much longer than necessary or wind up producing less-than-ideal results. Yet much of this folk knowledge is fairly easy to communicate. This is the purpose of this article.

» key insights

- Machine learning algorithms can figure out how to perform important tasks by generalizing from examples. This is often feasible and cost-effective where manual programming is not. As more data becomes available, more ambitious problems can be tackled.
- Machine learning is widely used in computer science and other fields. However, developing successful machine learning applications requires a substantial amount of "black art" that is difficult to find in textbooks.
- This article summarizes 12 key lessons that machine learning researchers and practitioners have learned. These include pitfalls to avoid, important issues to focus on, and answers to common questions.

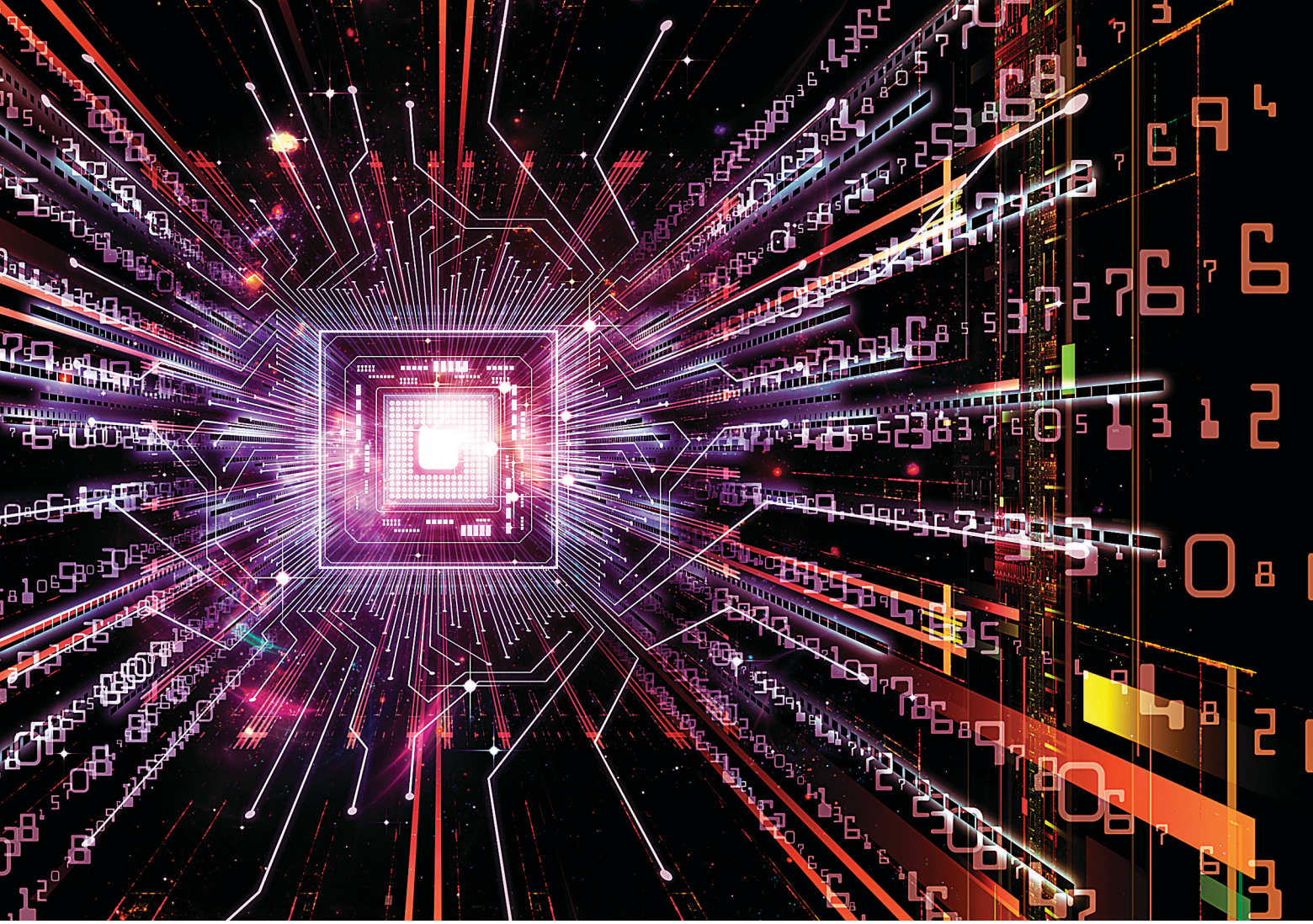


IMAGE BY AGSANDREW/SHUTTERSTOCK.COM

Many different types of machine learning exist, but for illustration purposes I will focus on the most mature and widely used one: classification. Nevertheless, the issues I will discuss apply across all of machine learning. A *classifier* is a system that inputs (typically) a vector of discrete and/or continuous *feature values* and outputs a single discrete value, the *class*. For example, a spam filter classifies email messages into "spam" or "not spam," and its input may be a Boolean vector $\mathbf{x} = (x_1,\ldots,x_j,\ldots,x_d)$, where $x_j = 1$ if the j^{th} word in the dictionary appears in the email and $x_j = 0$ otherwise. A *learner* inputs a training set of examples $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i = (x_{i,1}, \ldots, x_{i,d})$ is an observed input and y_i is the corresponding output, and outputs a classifier. The test of the learner is whether this classifier produces the correct output y_t for future examples $\mathbf{x}_t$ (for example, whether the spam filter correctly classifies previously unseen email messages as spam or not spam).

Learning = Representation + Evaluation + Optimization

Suppose you have an application that you think machine learning might be good for. The first problem facing you is the bewildering variety of learning algorithms available. Which one to use? There are literally thousands available, and hundreds more are published each year. The key to not getting lost in this huge space is to realize that it consists of combinations of just three components. The components are:

▸ **Representation.** A classifier must be represented in some formal language that the computer can handle. Conversely, choosing a representation for a learner is tantamount to choosing the set of classifiers that it can possibly learn. This set is called the *hypothesis space* of the learner. If a classifier is not in the hypothesis space, it cannot be learned. A related question, that I address later, is how to represent the input, in other words, what features to use.

▸ **Evaluation.** An evaluation function (also called *objective function* or *scoring function*) is needed to distinguish good classifiers from bad ones. The evaluation function used internally by the algorithm may differ from the external one that we want the classifier to optimize, for ease of optimization and due to the issues I will discuss.

▸ **Optimization.** Finally, we need a method to search among the classifiers in the language for the highest-scoring one. The choice of optimization technique is key to the efficiency of the learner, and also helps determine the classifier produced if the evaluation function has more than one optimum. It is common for new learners to start out using off-the-shelf optimizers, which are later replaced by custom-designed ones.

The accompanying table shows common examples of each of these three components. For example, *k*-nearest neighbor classifies a test example by finding the *k* most similar training examples and predicting the majority class among them. Hyperplane-based methods form a linear

Table 1. The three components of learning algorithms.

Representation	Evaluation	Optimization
Instances	Accuracy/Error rate	Combinatorial optimization
K-nearest neighbor	Precision and recall	Greedy search
Support vector machines	Squared error	Beam search
Hyperplanes	Likelihood	Branch-and-bound
Naive Bayes	Posterior probability	Continuous optimization
Logistic regression	Information gain	Unconstrained
Decision trees	K-L divergence	Gradient descent
Sets of rules	Cost/Utility	Conjugate gradient
Propositional rules	Margin	Quasi-Newton methods
Logic programs		Constrained
Neural networks		Linear programming
Graphical models		Quadratic programming
Bayesian networks		
Conditional random fields		

Algorithm 1. Decision tree induction.

LearnDT (*TrainSet*)

if all examples in *TrainSet* have the same class y_* **then**
  return MakeLeaf(y_*)
if no feature x_j has InfoGain(x_j, y) > 0 **then**
  $y_* \leftarrow$ Most frequent class in *TrainSet*
  return MakeLeaf(y_*)
$x_* \leftarrow \text{argmax}_{x_j}$ InfoGain(x_j, y)
$TS_0 \leftarrow$ Examples in *TrainSet* with $x_* = 0$
$TS_1 \leftarrow$ Examples in *TrainSet* with $x_* = 1$
return MakeNode(x_*, LearnDT(TS_0), LearnDT(TS_1))

combination of the features per class and predict the class with the highest-valued combination. Decision trees test one feature at each internal node, with one branch for each feature value, and have class predictions at the leaves. Algorithm 1 (above) shows a bare-bones decision tree learner for Boolean domains, using information gain and greedy search.[20] InfoGain(x_j, y) is the mutual information between feature x_j and the class y. MakeNode(x,c_0,c_1) returns a node that tests feature x and has c_0 as the child for $x = 0$ and c_1 as the child for $x = 1$.

Of course, not all combinations of one component from each column of the table make equal sense. For example, discrete representations naturally go with combinatorial optimization, and continuous ones with continuous optimization. Nevertheless, many learners have both discrete and continuous components, and in fact the day may not be far when every single possible combination has appeared in some learner!

Most textbooks are organized by representation, and it is easy to overlook the fact that the other components are equally important. There is no simple recipe for choosing each component, but I will touch on some of the key issues here. As we will see, some choices in a machine learning project may be even more important than the choice of learner.

It's Generalization that Counts

The fundamental goal of machine learning is to generalize beyond the examples in the training set. This is because, no matter how much data we have, it is very unlikely that we will see those exact examples again at test time. (Notice that, if there are 100,000 words in the dictionary, the spam filter described above has $2^{100,000}$ possible different inputs.) Doing well on the training set is easy (just memorize the examples). The most common mistake among machine learning beginners is to test on the training data and have the illusion of success. If the chosen classifier is then tested on new data, it is often no better than random guessing. So, if you hire someone to build a classifier, be sure to keep some of the data to yourself and test the classifier they give you on it. Conversely, if you have been hired to build a classifier, set some of the data aside from the beginning, and only use it to test your chosen classifier at the very end, followed by learning your final classifier on the whole data.

Contamination of your classifier by test data can occur in insidious ways, for example, if you use test data to tune parameters and do a lot of tuning. (Machine learning algorithms have lots of knobs, and success often comes from twiddling them a lot, so this is a real concern.) Of course, holding out data reduces the amount available for training. This can be mitigated by doing cross-validation: randomly dividing your training data into (say) 10 subsets, holding out each one while training on the rest, testing each learned classifier on the examples it did not see, and averaging the results to see how well the particular parameter setting does.

In the early days of machine learning, the need to keep training and test data separate was not widely appreciated. This was partly because, if the learner has a very limited representation (for example, hyperplanes), the difference between training and test error may not be large. But with very flexible classifiers (for example, decision trees), or even with linear classifiers with a lot of features, strict separation is mandatory.

Notice that generalization being the goal has an interesting consequence for machine learning. Unlike in most other optimization problems, we do not have access to the function we want to optimize! We have to use training error as a surrogate for test error, and this is fraught with danger. (How to deal with it is addressed later.) On the positive side, since the objective function is only a proxy for the true goal, we may not need to fully

optimize it; in fact, a local optimum returned by simple greedy search may be better than the global optimum.

Data Alone Is Not Enough

Generalization being the goal has another major consequence: Data alone is not enough, no matter how much of it you have. Consider learning a Boolean function of (say) 100 variables from a million examples. There are $2^{100} - 10^6$ examples whose classes you do not know. How do you figure out what those classes are? In the absence of further information, there is just no way to do this that beats flipping a coin. This observation was first made (in somewhat different form) by the philosopher David Hume over 200 years ago, but even today many mistakes in machine learning stem from failing to appreciate it. Every learner must embody some knowledge or assumptions beyond the data it is given in order to generalize beyond it. This notion was formalized by Wolpert in his famous "no free lunch" theorems, according to which no learner can beat random guessing over all possible functions to be learned.[25]

This seems like rather depressing news. How then can we ever hope to learn anything? Luckily, the functions we want to learn in the real world are *not* drawn uniformly from the set of all mathematically possible functions! In fact, very general assumptions—like smoothness, similar examples having similar classes, limited dependences, or limited complexity—are often enough to do very well, and this is a large part of why machine learning has been so successful. Like deduction, induction (what learners do) is a knowledge lever: it turns a small amount of input knowledge into a large amount of output knowledge. Induction is a vastly more powerful lever than deduction, requiring much less input knowledge to produce useful results, but it still needs more than zero input knowledge to work. And, as with any lever, the more we put in, the more we can get out.

A corollary of this is that one of the key criteria for choosing a representation is which kinds of knowledge are easily expressed in it. For example, if we have a lot of knowledge about what makes examples similar in our domain, instance-based methods may be a good choice. If we have knowledge about probabilistic dependencies, graphical models are a good fit. And if we have knowledge about what kinds of preconditions are required by each class, "IF . . . THEN . . ." rules may be the best option. The most useful learners in this regard are those that do not just have assumptions hardwired into them, but allow us to state them explicitly, vary them widely, and incorporate them automatically into the learning (for example, using first-order logic[21] or grammars[6]).

In retrospect, the need for knowledge in learning should not be surprising. Machine learning is not magic; it cannot get something from nothing. What it does is get more from less. Programming, like all engineering, is a lot of work: we have to build everything from scratch. Learning is more like farming, which lets nature do most of the work. Farmers combine seeds with nutrients to grow crops. Learners combine knowledge with data to grow programs.

Overfitting Has Many Faces

What if the knowledge and data we have are not sufficient to completely determine the correct classifier? Then we run the risk of just hallucinating a classifier (or parts of it) that is not grounded in reality, and is simply encoding random quirks in the data. This problem is called *overfitting*, and is the bugbear of machine learning. When your learner outputs a classifier that is 100% accurate on the training data but only 50% accurate on test data, when in fact it could have output one that is 75% accurate on both, it has overfit.

Everyone in machine learning knows about overfitting, but it comes in many forms that are not immediately obvious. One way to understand overfitting is by decomposing generalization error into *bias* and *variance*.[9] Bias is a learner's tendency to consistently learn the same wrong thing. Variance is the tendency to learn random things irrespective of the real signal. Figure 1 illustrates this by an analogy with throwing darts at a board. A linear learner has high bias, because when the frontier between two classes is not a hyperplane the learner is unable to induce it. Decision trees do not have this problem because they can represent any Boolean function, but on the other hand they can suffer from high variance: decision trees learned on different training sets generated by the same phenomenon are often very different, when in fact they should be

Figure 1. Bias and variance in dart-throwing.

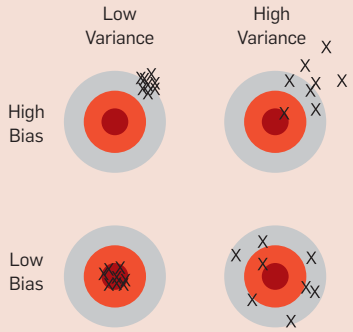
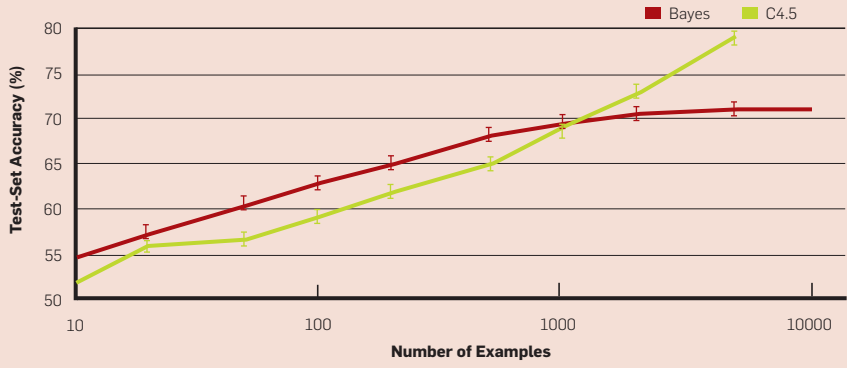


Figure 2. Naïve Bayes can outperform a state-of-the-art rule learner (C4.5rules) even when the true classifier is a set of rules.



the same. Similar reasoning applies to the choice of optimization method: beam search has lower bias than greedy search, but higher variance, because it tries more hypotheses. Thus, contrary to intuition, a more powerful learner is not necessarily better than a less powerful one.

Figure 2 illustrates this.[a] Even though the true classifier is a set of rules, with up to 1,000 examples naive Bayes is more accurate than a rule learner. This happens despite naive Bayes's false assumption that the frontier is linear! Situations like this are common in machine learning: strong false assumptions can be better than weak true ones, because a learner with the latter needs more data to avoid overfitting.

Cross-validation can help to combat overfitting, for example by using it to choose the best size of decision tree to learn. But it is no panacea, since if we use it to make too many parameter choices it can itself start to overfit.[17]

Besides cross-validation, there are many methods to combat overfitting. The most popular one is adding a *regularization term* to the evaluation function. This can, for example, penalize classifiers with more structure, thereby favoring smaller ones with less room to overfit. Another option is to perform a statistical significance test like chi-square before adding new structure, to decide whether the distribution of the class really is different with and without this structure. These techniques are particularly useful when data is very scarce. Nevertheless, you should be skeptical of claims that a particular technique "solves" the overfitting problem. It is easy to avoid overfitting (variance) by falling into the opposite error of underfitting (bias). Simultaneously avoiding both requires learning a perfect classifier, and short of knowing it in advance there is no single technique that will always do best (no free lunch).

A common misconception about overfitting is that it is caused by noise, like training examples labeled with the wrong class. This can indeed aggravate overfitting, by making the learner draw a capricious frontier to keep those examples on what it thinks is the right side. But severe overfitting can occur even in the absence of noise. For instance, suppose we learn a Boolean classifier that is just the disjunction of the examples labeled "true" in the training set. (In other words, the classifier is a Boolean formula in disjunctive normal form, where each term is the conjunction of the feature values of one specific training example.) This classifier gets all the training examples right and every positive test example wrong, regardless of whether the training data is noisy or not.

The problem of *multiple testing*[13] is closely related to overfitting. Standard statistical tests assume that only one hypothesis is being tested, but modern learners can easily test millions before they are done. As a result what looks significant may in fact not be. For example, a mutual fund that beats the market 10 years in a row looks very impressive, until you realize that, if there are 1,000 funds and each has a 50% chance of beating the market on any given year, it is quite likely that one will succeed all 10 times just by luck. This problem can be combatted by correcting the significance tests to take the number of hypotheses into account, but this can also lead to underfitting. A better approach is to control the fraction of falsely accepted non-null hypotheses, known as the *false discovery rate*.[3]

Intuition Fails in High Dimensions

After overfitting, the biggest problem in machine learning is the *curse of dimensionality*. This expression was coined by Bellman in 1961 to refer to the fact that many algorithms that work fine in low dimensions become intractable when the input is high-dimensional. But in machine learning it refers to much more. Generalizing correctly becomes exponentially harder as the dimensionality (number of features) of the examples grows, because a fixed-size training set covers a dwindling fraction of the input space. Even with a moderate dimension of 100 and a huge training set of a trillion examples, the latter covers only a fraction of about 10^{-18} of the input space. This is what makes machine learning both necessary and hard.

More seriously, the similarity-based reasoning that machine learning algorithms depend on (explicitly or implicitly) breaks down in high dimensions. Consider a nearest neighbor classifier with Hamming distance as the similarity measure, and suppose the class is just $x_1 \wedge x_2$. If there are no other features, this is an easy problem. But if there are 98 irrelevant features $x_3,\ldots, x_{100}$, the noise from them completely swamps the signal in x_1 and x_2, and nearest neighbor effectively makes random predictions.

Even more disturbing is that nearest neighbor still has a problem even if all 100 features are relevant! This is because in high dimensions all examples look alike. Suppose, for instance, that examples are laid out on a regular grid, and consider a test example x_t. If the grid is d-dimensional, x_t's $2d$ nearest examples are all at the same distance from it. So as the dimensionality increases, more and more examples become nearest neighbors of x_t, until the choice of nearest neighbor (and therefore of class) is effectively random.

This is only one instance of a more general problem with high dimensions: our intuitions, which come from a three-dimensional world, often do not apply in high-dimensional ones. In high dimensions, most of the mass of a multivariate Gaussian distribution is not near the mean, but in an increasingly distant "shell" around it; and most of the volume of a high-dimensional orange is in the skin, not the pulp. If a constant number of examples is distributed uniformly in a high-dimensional hypercube, beyond some dimensionality most examples are closer to a face of the hypercube than to their nearest neighbor. And if we approximate a hypersphere by inscribing it in a hypercube, in high dimensions almost all the volume of the hypercube is outside the hypersphere. This is bad news for machine learning, where shapes of one type are often approximated by shapes of another.

Building a classifier in two or three dimensions is easy; we can find a reasonable frontier between examples of different classes just by visual in-

a Training examples consist of 64 Boolean features and a Boolean class computed from them according to a set of "IF . . . THEN . . ." rules. The curves are the average of 100 runs with different randomly generated sets of rules. Error bars are two standard deviations. See Domingos and Pazzani[10] for details.

spection. (It has even been said that if people could see in high dimensions machine learning would not be necessary.) But in high dimensions it is difficult to understand what is happening. This in turn makes it difficult to design a good classifier. Naively, one might think that gathering more features never hurts, since at worst they provide no new information about the class. But in fact their benefits may be outweighed by the curse of dimensionality.

Fortunately, there is an effect that partly counteracts the curse, which might be called the "blessing of nonuniformity." In most applications examples are not spread uniformly throughout the instance space, but are concentrated on or near a lower-dimensional manifold. For example, *k*-nearest neighbor works quite well for handwritten digit recognition even though images of digits have one dimension per pixel, because the space of digit images is much smaller than the space of all possible images. Learners can implicitly take advantage of this lower effective dimension, or algorithms for explicitly reducing the dimensionality can be used (for example, Tenenbaum[22]).

Theoretical Guarantees Are Not What They Seem

Machine learning papers are full of theoretical guarantees. The most common type is a bound on the number of examples needed to ensure good generalization. What should you make of these guarantees? First of all, it is remarkable that they are even possible. Induction is traditionally contrasted with deduction: in deduction you can guarantee that the conclusions are correct; in induction all bets are off. Or such was the conventional wisdom for many centuries. One of the major developments of recent decades has been the realization that in fact we can have guarantees on the results of induction, particularly if we are willing to settle for probabilistic guarantees.

The basic argument is remarkably simple.[5] Let's say a classifier is bad if its true error rate is greater than ϵ. Then the probability that a bad classifier is consistent with n random, independent training examples is less than $(1 - \epsilon)^n$. Let b be the number of bad classifiers in the learner's hypothesis space H. The probability that at least one of them is consistent is less than $b(1 - \epsilon)^n$, by the union bound. Assuming the learner always returns a consistent classifier, the probability that this classifier is bad is then less than $|H|(1 - \epsilon)^n$, where we have used the fact that $b \leq |H|$. So if we want this probability to be less than δ, it suffices to make $n > \ln(\delta/|H|)/\ln(1 - \epsilon) \geq 1/\epsilon\,(\ln |H| + \ln 1/\delta)$.

One of the major developments of recent decades has been the realization that we can have guarantees on the results of induction, particularly if we are willing to settle for probabilistic guarantees.

Unfortunately, guarantees of this type have to be taken with a large grain of salt. This is because the bounds obtained in this way are usually extremely loose. The wonderful feature of the bound above is that the required number of examples only grows logarithmically with $|H|$ and $1/\delta$. Unfortunately, most interesting hypothesis spaces are *doubly* exponential in the number of features d, which still leaves us needing a number of examples exponential in d. For example, consider the space of Boolean functions of d Boolean variables. If there are e possible different examples, there are 2^e possible different functions, so since there are 2^d possible examples, the total number of functions is 2^{2^d}. And even for hypothesis spaces that are "merely" exponential, the bound is still very loose, because the union bound is very pessimistic. For example, if there are 100 Boolean features and the hypothesis space is decision trees with up to 10 levels, to guarantee $\delta = \epsilon = 1\%$ in the bound above we need half a million examples. But in practice a small fraction of this suffices for accurate learning.

Further, we have to be careful about what a bound like this means. For instance, it does not say that, if your learner returned a hypothesis consistent with a particular training set, then this hypothesis probably generalizes well. What it says is that, given a large enough training set, with high probability your learner will either return a hypothesis that generalizes well or be unable to find a consistent hypothesis. The bound also says nothing about how to select a good hypothesis space. It only tells us that, if the hypothesis space contains the true classifier, then the probability that the learner outputs a bad classifier decreases with training set size.

If we shrink the hypothesis space, the bound improves, but the chances that it contains the true classifier shrink also. (There are bounds for the case where the true classifier is not in the hypothesis space, but similar considerations apply to them.)

Another common type of theoretical guarantee is asymptotic: given infinite data, the learner is guaranteed to output the correct classifier. This is reassuring, but it would be rash to choose one learner over another because of its asymptotic guarantees. In practice, we are seldom in the asymptotic regime (also known as "asymptopia"). And, because of the bias-variance trade-off I discussed earlier, if learner A is better than learner B given infinite data, B is often better than A given finite data.

The main role of theoretical guarantees in machine learning is not as a criterion for practical decisions, but as a source of understanding and driving force for algorithm design. In this capacity, they are quite useful; indeed, the close interplay of theory and practice is one of the main reasons machine learning has made so much progress over the years. But caveat emptor: learning is a complex phenomenon, and just because a learner has a theoretical justification and works in practice does not mean the former is the reason for the latter.

Feature Engineering Is The Key

At the end of the day, some machine learning projects succeed and some fail. What makes the difference? Easily the most important factor is the features used. Learning is easy if you have many independent features that each correlate well with the class. On the other hand, if the class is a very complex function of the features, you may not be able to learn it. Often, the raw data is not in a form that is amenable to learning, but you can construct features from it that are. This is typically where most of the effort in a machine learning project goes. It is often also one of the most interesting parts, where intuition, creativity and "black art" are as important as the technical stuff.

First-timers are often surprised by how little time in a machine learning project is spent actually doing machine learning. But it makes sense if you consider how time-consuming it is to gather data, integrate it, clean it and preprocess it, and how much trial and error can go into feature design. Also, machine learning is not a one-shot process of building a dataset and running a learner, but rather an iterative process of running the learner, analyzing the results, modifying the data and/or the learner, and repeating. Learning is often the quickest part of this, but that is because we have already mastered it pretty well! Feature engineering is more difficult because it is domain-specific, while learners can be largely general purpose. However, there is no sharp frontier between the two, and this is another reason the most useful learners are those that facilitate incorporating knowledge.

A dumb algorithm with lots and lots of data beats a clever one with modest amounts of it.

Of course, one of the holy grails of machine learning is to automate more and more of the feature engineering process. One way this is often done today is by automatically generating large numbers of candidate features and selecting the best by (say) their information gain with respect to the class. But bear in mind that features that look irrelevant in isolation may be relevant in combination. For example, if the class is an XOR of k input features, each of them by itself carries no information about the class. (If you want to annoy machine learners, bring up XOR.) On the other hand, running a learner with a very large number of features to find out which ones are useful in combination may be too time-consuming, or cause overfitting. So there is ultimately no replacement for the smarts you put into feature engineering.

More Data Beats a Cleverer Algorithm

Suppose you have constructed the best set of features you can, but the classifiers you receive are still not accurate enough. What can you do now? There are two main choices: design a better learning algorithm, or gather more data (more examples, and possibly more raw features, subject to the curse of dimensionality). Machine learning researchers are mainly concerned with the former, but pragmatically the quickest path to success is

often to just get more data. As a rule of thumb, a dumb algorithm with lots and lots of data beats a clever one with modest amounts of it. (After all, machine learning is all about letting data do the heavy lifting.)

This does bring up another problem, however: scalability. In most of computer science, the two main limited resources are time and memory. In machine learning, there is a third one: training data. Which one is the bottleneck has changed from decade to decade. In the 1980s it tended to be data. Today it is often time. Enormous mountains of data are available, but there is not enough time to process it, so it goes unused. This leads to a paradox: even though in principle more data means that more complex classifiers can be learned, in practice simpler classifiers wind up being used, because complex ones take too long to learn. Part of the answer is to come up with fast ways to learn complex classifiers, and indeed there has been remarkable progress in this direction (for example, Hulten and Domingos[11]).

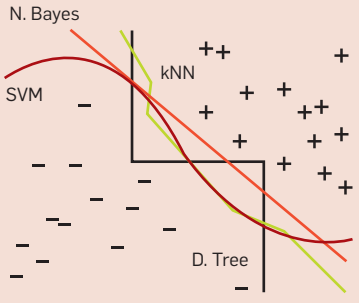
Part of the reason using cleverer algorithms has a smaller payoff than you might expect is that, to a first approximation, they all do the same. This is surprising when you consider representations as different as, say, sets of rules and neural networks. But in fact propositional rules are readily encoded as neural networks, and similar relationships hold between other representations. All learners essentially work by grouping nearby examples into the same class; the key difference is in the meaning of "nearby." With nonuniformly distributed data, learners can produce widely different frontiers while still making the same predictions in the regions that matter (those with a substantial number of training examples, and therefore also where most test examples are likely to appear). This also helps explain why powerful learners can be unstable but still accurate. Figure 3 illustrates this in 2D; the effect is much stronger in high dimensions.

As a rule, it pays to try the simplest learners first (for example, naïve Bayes before logistic regression, *k*-nearest neighbor before support vector machines). More sophisticated learners are seductive, but they are usually harder to use, because they have more knobs you need to turn to get good results, and because their internals are more opaque.

Learners can be divided into two major types: those whose representation has a fixed size, like linear classifiers, and those whose representation can grow with the data, like decision trees. (The latter are sometimes called nonparametric learners, but this is somewhat unfortunate, since they usually wind up learning many more parameters than parametric ones.) Fixed-size learners can only take advantage of so much data. (Notice how the accuracy of naive Bayes asymptotes at around 70% in Figure 2.) Variable-size learners can in principle learn any function given sufficient data, but in practice they may not, because of limitations of the algorithm (for example, greedy search falls into local optima) or computational cost. Also, because of the curse of dimensionality, no existing amount of data may be enough. For these reasons, clever algorithms—those that make the most of the data and computing resources available—often pay off in the end, provided you are willing to put in the effort. There is no sharp frontier between designing learners and learning classifiers; rather, any given piece of knowledge could be encoded in the learner or learned from data. So machine learning projects often wind up having a significant component of learner design, and practitioners need to have some expertise in it.[12]

In the end, the biggest bottleneck is not data or CPU cycles, but human

Figure 3. Very different frontiers can yield similar predictions. (+ and – are training examples of two classes.)



cycles. In research papers, learners are typically compared on measures of accuracy and computational cost. But human effort saved and insight gained, although harder to measure, are often more important. This favors learners that produce human-understandable output (for example, rule sets). And the organizations that make the most of machine learning are those that have in place an infrastructure that makes experimenting with many different learners, data sources, and learning problems easy and efficient, and where there is a close collaboration between machine learning experts and application domain ones.

Learn Many Models, Not Just One

In the early days of machine learning, everyone had a favorite learner, together with some a priori reasons to believe in its superiority. Most effort went into trying many variations of it and selecting the best one. Then systematic empirical comparisons showed that the best learner varies from application to application, and systems containing many different learners started to appear. Effort now went into trying many variations of many learners, and still selecting just the best one. But then researchers noticed that, if instead of selecting the best variation found, we combine many variations, the results are better—often much better—and at little extra effort for the user.

Creating such *model ensembles* is now standard.[1] In the simplest technique, called *bagging*, we simply generate random variations of the training set by resampling, learn a classifier on each, and combine the results by voting. This works because it greatly reduces variance while only slightly increasing bias. In *boosting*, training examples have weights, and these are varied so that each new classifier focuses on the examples the previous ones tended to get wrong. In *stacking*, the outputs of individual classifiers become the inputs of a "higher-level" learner that figures out how best to combine them.

Many other techniques exist, and the trend is toward larger and larger ensembles. In the Netflix prize, teams from all over the world competed to build the best video recommender

system (http://netflixprize.com). As the competition progressed, teams found they obtained the best results by combining their learners with other teams', and merged into larger and larger teams. The winner and runner-up were both stacked ensembles of over 100 learners, and combining the two ensembles further improved the results. Doubtless we will see even larger ones in the future.

Model ensembles should not be confused with Bayesian model averaging (BMA)—the theoretically optimal approach to learning.[4] In BMA, predictions on new examples are made by averaging the individual predictions of *all* classifiers in the hypothesis space, weighted by how well the classifiers explain the training data and how much we believe in them a priori. Despite their superficial similarities, ensembles and BMA are very different. Ensembles change the hypothesis space (for example, from single decision trees to linear combinations of them), and can take a wide variety of forms. BMA assigns weights to the hypotheses in the original space according to a fixed formula. BMA weights are extremely different from those produced by (say) bagging or boosting: the latter are fairly even, while the former are extremely skewed, to the point where the single highest-weight classifier usually dominates, making BMA effectively equivalent to just selecting it.[8] A practical consequence of this is that, while model ensembles are a key part of the machine learning toolkit, BMA is seldom worth the trouble.

Simplicity Does Not Imply Accuracy

Occam's razor famously states that entities should not be multiplied beyond necessity. In machine learning, this is often taken to mean that, given two classifiers with the same training error, the simpler of the two will likely have the lowest test error. Purported proofs of this claim appear regularly in the literature, but in fact there are many counterexamples to it, and the "no free lunch" theorems imply it cannot be true.

We saw one counterexample previously: model ensembles. The generalization error of a boosted ensemble continues to improve by adding classifiers even after the training error has reached zero. Another counterexample is support vector machines, which can effectively have an infinite number of parameters without overfitting. Conversely, the function sign(sin(ax)) can discriminate an arbitrarily large, arbitrarily labeled set of points on the x axis, even though it has only one parameter.[23] Thus, contrary to intuition, there is no necessary connection between the number of parameters of a model and its tendency to overfit.

Just because a function can be represented does not mean it can be learned.

A more sophisticated view instead equates complexity with the size of the hypothesis space, on the basis that smaller spaces allow hypotheses to be represented by shorter codes. Bounds like the one in the section on theoretical guarantees might then be viewed as implying that shorter hypotheses generalize better. This can be further refined by assigning shorter codes to the hypotheses in the space we have some a priori preference for. But viewing this as "proof" of a trade-off between accuracy and simplicity is circular reasoning: we made the hypotheses we prefer simpler by design, and if they are accurate it is because our preferences are accurate, not because the hypotheses are "simple" in the representation we chose.

A further complication arises from the fact that few learners search their hypothesis space exhaustively. A learner with a larger hypothesis space that tries fewer hypotheses from it is less likely to overfit than one that tries more hypotheses from a smaller space. As Pearl[18] points out, the size of the hypothesis space is only a rough guide to what really matters for relating training and test error: the procedure by which a hypothesis is chosen.

Domingos[7] surveys the main arguments and evidence on the issue of Occam's razor in machine learning. The conclusion is that simpler hypotheses should be preferred because simplicity is a virtue in its own right, not because of a hypothetical connection with accuracy. This is probably what Occam meant in the first place.

Representable Does Not Imply Learnable

Essentially all representations used in variable-size learners have associated

theorems of the form "Every function can be represented, or approximated arbitrarily closely, using this representation." Reassured by this, fans of the representation often proceed to ignore all others. However, just because a function can be represented does not mean it can be learned. For example, standard decision tree learners cannot learn trees with more leaves than there are training examples. In continuous spaces, representing even simple functions using a fixed set of primitives often requires an infinite number of components. Further, if the hypothesis space has many local optima of the evaluation function, as is often the case, the learner may not find the true function even if it is representable. Given finite data, time and memory, standard learners can learn only a tiny subset of all possible functions, and these subsets are different for learners with different representations. Therefore the key question is not "Can it be represented?" to which the answer is often trivial, but "Can it be learned?" And it pays to try different learners (and possibly combine them).

Some representations are exponentially more compact than others for some functions. As a result, they may also require exponentially less data to learn those functions. Many learners work by forming linear combinations of simple basis functions. For example, support vector machines form combinations of kernels centered at some of the training examples (the support vectors). Representing parity of n bits in this way requires 2^n basis functions. But using a representation with more layers (that is, more steps between input and output), parity can be encoded in a linear-size classifier. Finding methods to learn these deeper representations is one of the major research frontiers in machine learning.[2]

Correlation Does Not Imply Causation

The point that correlation does not imply causation is made so often that it is perhaps not worth belaboring. But, even though learners of the kind we have been discussing can only learn correlations, their results are often treated as representing causal relations. Isn't this wrong? If so, then why do people do it?

More often than not, the goal of learning predictive models is to use them as guides to action. If we find that beer and diapers are often bought together at the supermarket, then perhaps putting beer next to the diaper section will increase sales. (This is a famous example in the world of data mining.) But short of actually doing the experiment it is difficult to tell. Machine learning is usually applied to *observational* data, where the predictive variables are not under the control of the learner, as opposed to *experimental* data, where they are. Some learning algorithms can potentially extract causal information from observational data, but their applicability is rather restricted.[19] On the other hand, correlation is a sign of a potential causal connection, and we can use it as a guide to further investigation (for example, trying to understand what the causal chain might be).

Many researchers believe that causality is only a convenient fiction. For example, there is no notion of causality in physical laws. Whether or not causality really exists is a deep philosophical question with no definitive answer in sight, but there are two practical points for machine learners. First, whether or not we call them "causal," we would like to predict the effects of our actions, not just correlations between observable variables. Second, if you can obtain experimental data (for example by randomly assigning visitors to different versions of a Web site), then by all means do so.[14]

Conclusion

Like any discipline, machine learning has a lot of "folk wisdom" that can be difficult to come by, but is crucial for success. This article summarized some of the most salient items. Of course, it is only a complement to the more conventional study of machine learning. Check out http://www.cs.washington.edu/homes/pedrod/class for a complete online machine learning course that combines formal and informal aspects. There is also a treasure trove of machine learning lectures at http://www.videolectures.net. A good open source machine learning toolkit is Weka.[24]

Happy learning!

References

1. Bauer, E. and Kohavi, R. An empirical comparison of voting classification algorithms: Bagging, boosting and variants. *Machine Learning 36* (1999), 105–142.
2. Bengio, Y. Learning deep architectures for AI. *Foundations and Trends in Machine Learning 2*, 1 (2009), 1–127.
3. Benjamini, Y. and Hochberg, Y. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society, Series B, 57* (1995), 289–300.
4. Bernardo, J.M. and Smith, A.F.M. *Bayesian Theory.* Wiley, NY, 1994.
5. Blumer, A., Ehrenfeucht, A., Haussler, D. and Warmuth, M.K. Occam's razor. *Information Processing Letters 24* (1987), 377–380.
6. Cohen, W.W. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence 68* (1994), 303–366.
7. Domingos, P. The role of Occam's razor in knowledge discovery. *Data Mining and Knowledge Discovery 3* (1999), 409–425.
8. Domingos, P. Bayesian averaging of classifiers and the overfitting problem. In *Proceedings of the 17th International Conference on Machine Learning* (Stanford, CA, 2000), Morgan Kaufmann, San Mateo, CA, 223–230.
9. Domingos, P. A unified bias-variance decomposition and its applications. In *Proceedings of the 17th International Conference on Machine Learning* (Stanford, CA, 2000), Morgan Kaufmann, San Mateo, CA, 231–238.
10. Domingos, P. and Pazzani, M. On the optimality of the simple Bayesian classifier under zero-one loss. *Machine Learning 29* (1997), 103–130.
11. Hulten, G. and Domingos, P. Mining complex models from arbitrarily large databases in constant time. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Edmonton, Canada, 2002). ACM Press, NY, 525–531.
12. Kibler, D. and Langley, P. Machine learning as an experimental science. In *Proceedings of the 3rd European Working Session on Learning* (London, UK, 1988). Pitman.
13. Klockars, A.J. and Sax, G. *Multiple Comparisons.* Sage, Beverly Hills, CA, 1986.
14. Kohavi, R., Longbotham, R., Sommerfield, D. and Henne, R. Controlled experiments on the Web: Survey and practical guide. *Data Mining and Knowledge Discovery 18* (2009), 140–181.
15. Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C. and Byers, A. Big data: The next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute, 2011.
16. Mitchell, T.M. *Machine Learning.* McGraw-Hill, NY, 1997.
17. Ng, A.Y. Preventing "overfitting" of cross-validation data. In *Proceedings of the 14th International Conference on Machine Learning* (Nashville, TN, 1997). Morgan Kaufmann, San Mateo, CA, 245–253.
18. Pearl, J. On the connection between the complexity and credibility of inferred models. *International Journal of General Systems 4* (1978), 255–264.
19. Pearl, J. *Causality: Models, Reasoning, and Inference.* Cambridge University Press, Cambridge, UK, 2000.
20. Quinlan, J.R. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.
21. Richardson, M. and P. Domingos. Markov logic networks. *Machine Learning 62* (2006), 107–136.
22. Tenenbaum, J., Silva, V. and Langford, J. A global geometric framework for nonlinear dimensionality reduction. *Science 290* (2000), 2319–2323.
23. Vapnik, V.N. *The Nature of Statistical Learning Theory.* Springer, NY, 1995.
24. Witten, I., Frank, E. and Hall, M. *Data Mining: Practical Machine Learning Tools and Techniques, 3rd Edition.* Morgan Kaufmann, San Mateo, CA, 2011.
25. Wolpert, D. The lack of a priori distinctions between learning algorithms. *Neural Computation 8* (1996), 1341–1390.

Pedro Domingos (pedrod@cs.washington.edu) is a professor in the Department of Computer Science and Engineering at the University of Washington, Seattle.