

Best Practices for Machine Learning Applications

Brett Wujek, Patrick Hall, and Funda Güneş

SAS Institute Inc.

ABSTRACT

Building representative machine learning models that generalize well on future data requires careful consideration both of the data at hand and of assumptions about the various available training algorithms. Data are rarely in an ideal form that enables algorithms to train effectively. Some algorithms are designed to account for important considerations such as variable selection and handling of missing values, whereas other algorithms require additional preprocessing of the data or appropriate tweaking of the algorithm options. Ultimate evaluation of a model's quality requires appropriate selection and interpretation of an assessment criterion that is meaningful for the given problem. This paper discusses many of the most common issues faced by machine learning practitioners and provides guidance for using these powerful algorithms to build effective models.

INTRODUCTION

The study of machine learning algorithms often focuses on minimizing the error over an unknown data-generating distribution (a significant departure from classical statistics), quantifying the complexity of such algorithms, and establishing the bounds of their error (Vapnik 1996). This paper seeks to make no comment on the mathematical and statistical study of algorithms, but rather to guide the machine learning practitioner through the common steps of data preparation, model training, and model deployment.

When applied judiciously, machine learning solutions deliver significant value to a business by extracting previously hidden knowledge from stored or streaming data. Consider the data to be a stockpile of building material and supplies, and machine learning algorithms to be the powerful tools that can help construct a valuable structure from that stockpile. When you extend this analogy to incorporate skilled craftsmen operating these tools, the final product still depends heavily on the existence of a well-defined plan, adherence to sound building practices, and avoidance of mistakes at critical junctures in the process.

Using machine learning effectively and successfully boils down to a combination of knowledge, awareness, and ultimately taking a scientific approach to the overall process. Machine learning is a fundamental element of data science. In that regard, there exists an implied commitment to taking a scientific approach when establishing and executing a machine learning solution (Donoho 2015). Blindly supplying data to these powerful algorithms with little forethought given to the nature of the data, the strengths/weaknesses and training options of the algorithms, appropriate assessment, and deployment desires will likely result in models that do not adequately address your business problem. Quite simply, better modeling practices lead to better business decisions.

This paper examines best practices and highlights common mistakes and oversights that are often witnessed in the various phases of devising a machine learning application: data preparation, training, and deployment. Table 1 outlines some of the most common challenges faced during implementation of a machine learning application along with corresponding suggested best practices. The remainder of this paper provides detailed descriptions and explanations for each of these issues.

Topic	Common Challenges	Suggested Best Practice
Data Preparation		
Data collection	<ul style="list-style-type: none"> Biased data Incomplete data The curse of dimensionality Sparsity 	<ul style="list-style-type: none"> Take time to understand the business problem and its context Enrich the data Dimension-reduction techniques Change representation of data (e.g., COO)
"Untidy" data	<ul style="list-style-type: none"> Value ranges as columns Multiple variables in the same column Variables in both rows and columns 	Restructure the data to be "tidy" by using the melt and cast process
Outliers	<ul style="list-style-type: none"> Out-of-range numeric values and unknown categorical values in score data Undue influence on squared loss functions (e.g., regression, GBM, k-means) 	<ul style="list-style-type: none"> Robust methods (e.g., Huber loss function) Discretization (binning) Winsorizing
Sparse target variables	<ul style="list-style-type: none"> Low primary event occurrence rate Overwhelming preponderance of zero or missing values in target 	<ul style="list-style-type: none"> Proportional oversampling Inverse prior probabilities Mixture models
Variables of disparate magnitudes	<ul style="list-style-type: none"> Misleading variable importance Distance measure imbalance Gradient dominance 	Standardization
High-cardinality variables	<ul style="list-style-type: none"> Overfitting Unknown categorical values in holdout data 	<ul style="list-style-type: none"> Discretization (binning) Weight of evidence Leave-one-out event rate
Missing data	<ul style="list-style-type: none"> Information loss Bias 	<ul style="list-style-type: none"> Discretization (binning) Imputation Tree-based modeling techniques
Strong multicollinearity	Unstable parameter estimates	<ul style="list-style-type: none"> Regularization Dimension reduction
Training		
Overfitting	High-variance and low-bias models that fail to generalize well	<ul style="list-style-type: none"> Regularization Noise injection Partitioning or cross validation
Hyperparameter tuning	Combinatorial explosion of hyperparameters in conventional algorithms (e.g., deep neural networks, super learners)	<ul style="list-style-type: none"> Local search optimization, including genetic algorithms Grid search, random search
Ensemble models	<ul style="list-style-type: none"> Single models that fail to provide adequate accuracy High-variance and low-bias models that fail to generalize well 	<ul style="list-style-type: none"> Established ensemble methods (e.g., bagging, boosting, stacking) Custom or manual combinations of predictions
Model Interpretation	Large number of parameters, rules, or other complexity obscures model interpretation	<ul style="list-style-type: none"> Variable selection by regularization (e.g., L1) Surrogate models Partial dependency plots, variable importance measures
Computational resource exploitation	<ul style="list-style-type: none"> Single-threaded algorithm implementations Heavy reliance on interpreted languages 	<ul style="list-style-type: none"> Train many single-threaded models in parallel Hardware acceleration (e.g. SSD, GPU) Low-level, native libraries Distributed computing, when appropriate
Deployment		
Model deployment	Trained model logic must be transferred from a development environment to an operational computing system to assist in organizational decision-making processes	<ul style="list-style-type: none"> Portable scoring code or scoring executables In-database scoring Web service scoring
Model decay	<ul style="list-style-type: none"> Business problem or market conditions have changed since the model was created New observations fall outside domain of training data 	<ul style="list-style-type: none"> Monitor models for decreasing accuracy Update/retrain models regularly Champion-challenger tests Online updates

Table 1. Best Practices for Common Machine Learning Challenges

PREPARATION

Effective machine learning models are built on a foundation of well-prepared data. The importance of this phase cannot be overstated. In fact, it is commonly proclaimed that 80% of the time spent in devising a successful machine learning application is spent in data preparation (Dasu and Johnson 2003). Data preparation is not strictly about appropriately transforming and cleaning existing data; it also includes a good understanding of the features that need to be considered and ensuring that the data at hand are appropriate in the first place. Shortcuts in data preparation will shortchange your models. As they say, “garbage in, garbage out.” Take the time to cultivate your data, and be wary of the common challenges described in this section.

ENSURING SUFFICIENT AND APPROPRIATE DATA

Do You Have the Right Data?

Before you simply throw your data into a modeling algorithm, before you even start to perform transformations to clean and shape your data to a form more suitable for modeling, start by asking yourself “do I have the *right* data to answer the business question being asked?” Just because you have a lot of data doesn’t mean you have the *right* data. Ensure that the data are representative of the entire domain of interest—that the observations cover the anticipated range of values when this model is used in production. Beware of the perils of extrapolation, and understand that machine learning algorithms build models that are representative of the available training samples. The algorithms can be very inaccurate outside of that subspace, as shown in the example of various neural networks that are trained to the data shown in Figure 1 from Lohninger (1999). If you can collect more data to account for the domain of anticipated application of your model, your resulting model will probably be more effective. In addition to understanding the domain of the input variables, make certain that the target values you observe in the data set to be used for training include values representative of what you expect when you deploy the model. In particular, if your target is nominal, then the trained model will only be able to predict the specific values in your training set.

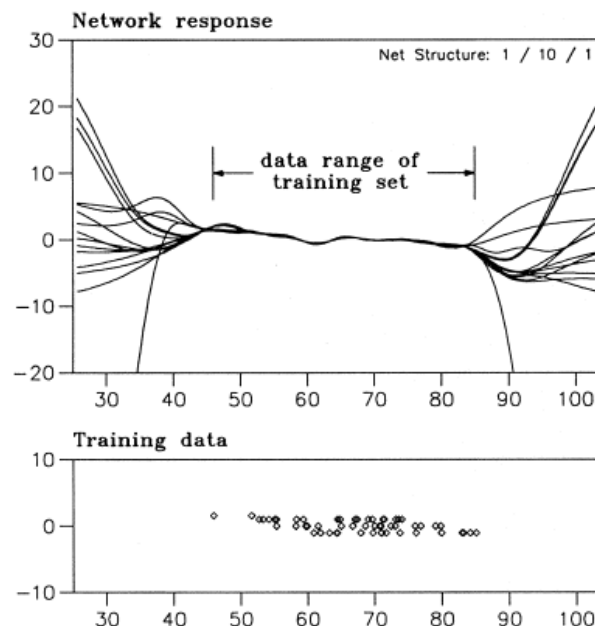


Figure 1. Highly Inaccurate Model Predictions from Extrapolation (Lohninger 1999)

Data Partitioning and Leakage

In typical machine learning tasks, data are divided into different sets (partitions): some data for training the model and some data for evaluating the model. It is critical that all transformations that are used to prepare the training data are applied to any validation, test, or other holdout data. It is also critically important that information from holdout data does not leak into the training data. Information leakage can occur in many ways and can potentially lead to **overfitting** or overly optimistic error measurements. For example, think of taking a mean or median across your data before partitioning and then later using this mean or median to impute missing values across all partitions of your data. In this case, your training data would be aware of information from your validation, test, or holdout partitions. To avoid this type of leakage, values for imputation and other basic transformations should be **generated from only the training data or within each partition independently**. Feature extraction can also lead to leakage. Consider principal component analysis (PCA), where features are created by decomposing a covariance or correlation matrix. If that covariance or correlation matrix is generated from all your data and then the derived principal component features or scores are used across all partitions of your data, your training data will be contaminated by information from other partitions. For more complex transformations such as feature extraction and binning, it is best to develop formulas or rules from the training data that can be applied to other partitions of data in order to generate the required features.

Accounting for Rare Events

It is also important to be mindful of your target of interest and understand whether it can be characterized as a rare event relative to your total number of samples. Applications such as detecting fraudulent activity must take special steps to ensure that the data used to train the model include a representative number of fraudulent samples in order to capture the event sufficiently (for example, 1 out of every 1,000 credit card transactions is fraudulent). Fitting a model to such data without accounting for the extreme imbalance in the occurrence of the event will provide you with a model that is extremely accurate at telling you absolutely nothing of value. Special sampling methods that modify an imbalanced data set are commonly used in order to provide a more balanced distribution when modeling rare events (He and Garcia 2009). These techniques include **oversampling** and **undersampling** methods.

In oversampling, the instances of the rare event class are increased by adding randomly selected instances from the existing rare events to achieve a more balanced overall distribution (usually close to 50%). This enables your model to more easily detect and express the relationship between features and the target of interest. Just realize that doing so **requires you to adjust your model predictions to account for the unnatural bias you have introduced to the rare event**, and that the predicted event probabilities and the false positive and false negative rates will not be accurate. In SAS® Enterprise Miner™ you can adjust your model predictions by defining a target profile and using weighting or offset methods that incorporate the probability of the event as observed in the complete population (the prior probabilities) versus the proportion of the event actually witnessed in the training set—that is, the posterior probabilities in the oversampled data set (Wielenga 2007). Oversampling has the additional benefit of enabling you to train on a more reasonably sized sample rather than requiring a very large sample simply to ensure that more rare event observations are included. A more reasonably sized sample results in **faster** training, allowing more time to experiment with different models. One problem with oversampling is that it can lead to **overfitting** because an excessive amount of replicated data for the rare class is used (Mease, Wyner, and Buja 2007).

In contrast to oversampling, **undersampling removes instances from the majority class in order to adjust the balance of the original data set**. For undersampling, the problem is relatively obvious—removing examples from the majority class might cause the classifier to miss important information from that class. To overcome this problem, you can use informed undersampling algorithms such as **EasyEnsemble** and **BalanceCascade**. Both of these techniques are very straightforward. The **EasyEnsemble method independently samples several subsets from the majority class and develops multiple classifiers that are trained on the combination of each subset with the minority class data**. In this way, EasyEnsemble can be considered to be an **unsupervised** learning algorithm that explores the majority class data by using independent random sampling with replacement. On the other hand, the **BalanceCascade algorithm takes a supervised learning approach that develops an ensemble of classifiers to systematically select which majority class examples to undersample** (Zhang and Mani 2003).

Another way to deal with rare events is to fit zero-inflated regression models, which are special cases of finite mixture models. Zero-inflated models view data as a mixture of a constant distribution (which always generates zero counts) and another distribution (which always generates nonzero counts), such as a Poisson, binomial, or multinomial distribution. Using zero-inflated models to model rare events enables you to generate a mixing probability that tells what percentage of data comes from a constant distribution and what percentage comes from the other distribution. You can also use these models for clustering rare events by assigning each observation to a component that has the highest posterior probability. In addition, handling rare events in the regression modeling framework is advantageous because it generates interpretable models.

TENDING TO THE DATA STRUCTURE AND FEATURE CONTENT

Assuming you have the *right* data, it is important to examine and understand the values of the features, both within each individual feature and across the set of features. After you ensure that the structure of your data is appropriate for your machine learning application, you should look for large discrepancies in magnitude among interval variables, high cardinality among nominal variables, and outliers and missing values within the values of each feature.

Data Structure

The original data set presented to you might be in a form that is ill-suited for applying machine learning algorithms to build models or identify patterns. If the data are unstructured or semi-structured (such as text from logs or information in XML format), you need to transform the data set in some way to produce a structured data set that is more suitable for modeling. But even if you have structured data, you should ensure that the rows represent what you consider to be observations and that the columns represent feature or target variables. This type of representation is referred to as “tidy data” in (Wickham 2014).

Take the time to transform your data set into a tidy data set so that you start the machine learning application with “a standardized way to link the structure of a dataset (its physical layout) with its semantics (its meaning)” (Wickham 2014). Wickham defines the tidying process to include the following techniques:

- **Melting:** When column headers contain values rather than true variables, melt (stack) those columns into two columns that are represented across multiple rows: one column for the original column header (which is really a value) and the other for the value from the original column (often a frequency). For example, if a data set has columns for income ranges such as <\$10k, \$10k–\$30k, and so on, with the values in those columns being the frequency of that income range for each observation (such as incomes within states), melt all these columns into two columns for Income Range and Frequency (such that you have multiple rows for each state).
- **String-splitting:** When a column actually contains multiple pieces of information, split it into multiple columns. For example, a column that contains M-20, F-35, M-42, and so on should be split into gender and age variables in two separate columns. Note that the melting process might result in such a column, so you might need to split after melting.
- **Casting:** When a column actually contains names of variables instead of values, cast (unstack) the column into multiple columns for each of the values (which are actually variables). Casting is the inverse of the melting process.

The ability of a machine learning algorithm to train an effective model for your business problem is highly dependent on the structure of your data set and on ensuring that features and observations are properly defined. Tidying your data set by melting, string-splitting, and casting can help you obtain more informative features and therefore increase model accuracy.

Standardization

Your data set most likely has features whose values are significantly different in magnitude and range. This disparity can degrade the performance of many machine learning algorithms, particularly those that distinguish observations and feature effectiveness based on a distance measurement (such as *k*-means clustering or nearest neighbor approaches), those that use numerical gradient information in their solution (such as neural networks and support vector machines), those that depend on a measure of the variance (such as principal component analysis), and those that penalize variables based on the size of their corresponding parameters (such as penalized regression techniques). The objective functions used for solving many of these machine learning algorithms can be dominated by the features that have large variance relative to other input features, preventing the model from being able to learn the relationship with the other features. For example, penalized regression techniques (which penalize the objective based on the magnitudes of the coefficients) disproportionately penalize the higher magnitude terms. However, some algorithms are scale invariant; for example, decision trees might bin inputs before splitting or make splitting decisions based on feature values independently of other features. But in general you need to account for the disparity in the magnitudes and ranges of values.

To mitigate the detrimental effect of widely varying magnitudes and ranges, you should **transform your interval feature values to be on a similar scale**, commonly standardizing to have mean 0 and variance 1 (z-scoring). Note that if your data set is sparse (that is, it contains a high percentage of 0 values), standardizing in this manner would destroy this sparseness and thus lose the ability to use methods that process sparse data very efficiently. In that case, a good alternative would be to simply scale the data based on the range or maximum absolute value of each feature. Also, if you have extreme outliers (in the far tails of the distribution) or nonnormally distributed features, z-scoring might not bring all values into scale. For extreme outliers, range standardization could compact all the other values into a very small percentage of the range and render them ineffectual in modeling; you might **need to address outliers before standardizing**. However you choose to scale your data, realize that you need to apply this same scaling process to new observations that are scored with any subsequently generated model in the future.

Standardization serves to put all variables on a level playing field, so to speak. However, it does not account for other issues within a set of individual variable values that must be considered. Some of the more common issues are covered in the following sections.

Managing Outliers

When you build models to predict future behavior or identify patterns, ideally you have a data set whose feature values are representative of typical observations. However, you will often find outliers in your data—observations that are very distinct from the others in one or more of the feature values. This can be true of interval variables that have values that occur in the extreme tails or as abnormal spikes in the distribution, and of nominal variables that have values that occur only in an extremely small percentage of the observations. Although outliers can certainly be very informative and can identify anomalies that deserve special attention, they can be quite detrimental to training an effective generalizable model.

Some algorithms are more robust at dealing with outliers, as described in the following list:

- **Supervised** learning algorithms that use a **squared-loss function** to determine the parameters that best fit the training data are heavily influenced by outliers. For example, gradient boosting algorithms add large weights to observations that are considered to be hard cases.
- If you are clustering your data, the **effect of outliers depends on the clustering method**. For example:
 - ***k*-means clustering can be quite sensitive to outliers** because it tries to minimize the sum of squared distances from cluster member points to the cluster means; a large deviation caused by an outlier receives a lot of weight.
 - Density-based clustering (such as **DBSCAN**) **tends to be less sensitive to outliers**, usually identifying them as individual outlying clusters.
 - **Hierarchical clustering simply assigns outliers to their own clusters.**

In general, outliers drag clusters artificially toward the outside of your feature space, either as individual clusters or by morphing clusters to incorporate the outliers.

Fortunately, outliers can usually be detected by some simple initial data exploration. You should make outlier management a standard part of your data exploration and preparation process before modeling. First, determine whether an outlying value is simply an invalid or erroneous entry that can be disregarded. If you have determined that an outlier provides no valuable information, it is acceptable to simply filter it out. However, if you determine that outliers might represent some real but rare relationship or if you think that the information from the other features in those observations is too valuable to discard, then include them in your model only after **dealing with them appropriately by using one of the following techniques**:

- **For categorical variables**, you can bin the values into an “Other” category. For more information, see the next section.
- **For interval variables**, you can **winsorize the values, setting them to the lowest or highest non-outlier value** (depending on which side of the distribution the outlier lies) or **forcing them to be no greater than three standard deviations** from the mean. Either of these approaches retains the observation, which might contain other valuable information from other features.
- **For algorithms that incorporate a loss function to direct the training process**, you can use a **Huber loss function** (Huber 1964), which greatly reduces the impact of outliers on the calculation of the loss.

Binning

Binning is the process of discretizing numerical variables into fewer categorical counterparts. For example, “age” variables are often binned into categories such as 20–39, 40–59, and 60–79. Building a model against each individual age probably does not provide any more information for a model than building it against age groups; **binning reduces the complexity of mapping the feature values to the response**. **Binning tends to generate a more effective predictive model and can make the model easier to interpret**. On the other hand, binning can also cause issues such as loss of power, because binning increases the number of model parameters to estimate. However, if you have big data, binning can be very beneficial, especially in difficult predictive modeling problems where many algorithms fail.

In particular, binning can simplify and improve accuracy of predictive models by doing the following:

- decreasing the impact of outliers
- enabling you to incorporate missing values
- managing high-cardinality variables (those with too many overall levels)
- reducing the noise or nonlinearity

Binning is known to work well to reduce noise (increase signal-to-noise ratio) and hence helps produce better predictive models for “messy” data that have many missing values, outliers, high-cardinality variables, nonlinearities between the input variables and the target, and skewed distributions of numeric input variables.

Missing Values

Missing values can be theoretically and practically problematic for many machine learning tasks, especially when missing values are present in the target variable. This section addresses only the more common scenario of missing values in input variables. When faced with missing values in input variables, practitioners must consider whether missing values are distributed randomly or whether missingness is somehow predictive of the target. If missing values appear at random in the input data, the input rows that contain missing values can be dropped from the analysis without introducing bias into the model. However, such a *complete case analysis* can remove a tremendous amount of information from the training data and reduce the predictive accuracy of the model. Missingness can actually be predictive: **retaining information that is associated with missing values, including the missing values themselves, can actually increase the predictive accuracy of a model**. The following list describes practices for accounting for missingness in training a machine learning model and describes how missing values must also be handled when scoring new data.

- **Naïve Bayes:** Naïve Bayes models elegantly handle missing values for training and scoring by computing the likelihood based on the observed features. Because of conditional independence between the features, naïve Bayes ignores a feature only when its value is missing. Thus, **you don't need to handle missing values before fitting a naïve Bayes model** unless you believe the missingness is not at random. For efficiency reasons, some implementations of naïve Bayes remove entire rows from the training process whenever a missing value is encountered. When missing is treated as a categorical level, infrequent missing values in new data can be problematic when they are not present in training data, because the missing level will have had no probability associated with it during training. You can solve this problem by ignoring the offending feature in the likelihood computation when scoring.
- **Decision trees:** In general, **imputation, missing markers, binning, and special scoring considerations are not required for missing values when you use a decision tree.** Decision trees allow for the elegant and direct use of missing values in two common ways:
 - When a splitting rule is determined, **missing can be considered to be a valid input value**, and missing values can either be placed on the side of the splitting rule that makes the best training prediction or be assigned to a separate branch in a split.
 - Surrogate rules can be defined to allow the tree to split on a surrogate variable when a missing value is encountered. For example, **a surrogate rule could be defined that allows a decision tree to split on the state variable when the zip code variable is missing.**
- **Missing markers:** Missing markers are binary variables that record whether the value of another variable is missing. They are used to preserve information about missingness so that missingness can be modeled. Missing markers can be used in a model to replace the original corresponding variable with missing values, or they can be used in a model alongside an imputed version of the original variable.
- **Imputation:** Imputation refers to replacing a missing value with information that is derived from nonmissing values in the training data. Simple imputation schemes include replacing a missing value in a particular input variable **with the mean or mode** of that variable's nonmissing values. For nonnormally distributed variables or variables that have a high proportion of missing values, simple mean or mode imputation can drastically alter a variable's distribution and negatively impact predictive accuracy. Even **when variables are normally distributed and contain a low proportion of missing values, creating missing markers and using them in the model alongside the new, imputed variables is a suggested practice.** Decision trees can also be used to derive imputed values. **A decision tree can be trained using a variable that has missing values as its target and all the other variables in the data set as inputs.** In this way, the decision tree can learn plausible replacement values for the missing values in the temporary target variable. This approach requires one decision tree for every input variable that has missing values, so it can become computationally expensive for large, dirty training sets. More sophisticated imputation approaches, including multiple imputation (MI), should be considered for small data sets (Rubin 1987).
- **Binning:** Interval input variables that have missing values can be discretized into a number of bins according to their original numeric values in order to create new categorical, nominal variables. Missing values in the original variable can simply be added to an additional bin in the new variable. Categorical input variables that have missing values can be assigned to new categorical nominal variables that have the same categorical levels as the corresponding original variables plus one new level for missing values. Because binning introduces additional nonlinearity into a predictive model and can be less damaging to an input variable's original distribution than imputation, **binning is generally considered acceptable**, if not beneficial, until the binning process begins to contribute to overfitting. However, you might not want to use binning if the ordering of the values in a particular input variable is important, because the ordering information is changed or erased by introducing a missing bin into the otherwise ordered values.

- **Scoring missing data:** If a decision tree or decision tree ensemble is used in training, missing values in new data will probably be scored automatically according to the splitting rules or the surrogate rules of the trained tree (or trees). If another type of algorithm was trained, then missing values in new data must be processed in the exact manner in which they were processed in the training data before the model was trained.

DIMENSIONALITY

Falling Prey to the Curse of Dimensionality

A common sentiment is that having more information will enable you to make better decisions. However, this belief is based on the assumption that you will be able to easily discern the meaningful information from the trivial and efficiently process the information. In reality, the more items of independent information you have, the more complex and costly your decision-making process becomes.

Although high dimensionality can result from the seemingly innocent desire to incorporate more features into your model, sometimes high dimensionality is simply the nature of the problem, such as in bioinformatics (DNA microarrays that are used to analyze tens of thousands of genes), multimedia data analysis with millions of pixels, and text documents that are represented by high-dimensional frequency count vectors.

The challenges that arise as you attempt to incorporate more features into your model are best described through a phenomenon known as the *curse of dimensionality* (Bellman 1957). Increasing the number of features increases the volume of the feature space exponentially; in turn, this higher-dimensional space requires exponentially more data points to sufficiently fill that space in order to ensure that combinations of feature values are accounted for. Figure 2 depicts the increased sparsity of the data with a very simple example of eight observations in one, two, and three dimensions. In one dimension, the observations sufficiently cover the domain, in two dimensions they still cover it fairly reasonably, but in three dimensions you start to see the sparsity of the data in the overall domain.

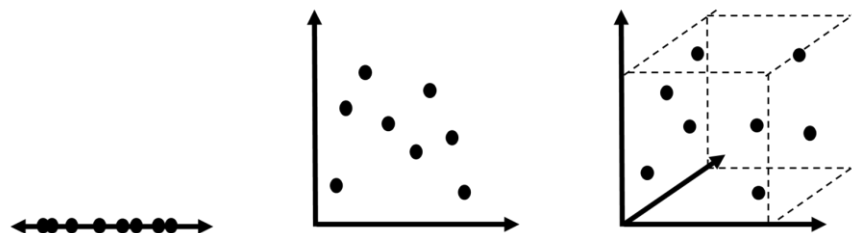


Figure 2. Increased Data Sparsity as Dimensions (Features) Are Added

It is difficult to imagine beyond three dimensions, but as the dimensionality further increases, a greater percentage of the available data reside in the “corners” of the feature space, which are much more difficult to classify than observations in the ever-shrinking “center” of the feature space. An illustrative example of this is provided in Spruyt (2014).

Consider *k*-means clustering as an example of the adverse effect of higher dimensionality. High dimensionality makes clustering difficult because having many dimensions means that all the observations are relatively “far away” from each other. It is difficult to know what a cluster is when all of the observations within a cluster are distant from one another. Generally, observations within a cluster are assumed to be close to one another; but in a high-dimensional space a cluster might be more like a dispersed cloud of loosely related points.

In general, higher dimensionality tends to stimulate overfitting, can lead to greater confounding among the feature effects, and renders visualization of the original problem domain impractical. Also, the amount of time and memory required to process additional dimensions generally diminishes the efficiency of the modeling algorithm.



Dimension Reduction

For the most efficient and effective predictive model, it is best to reduce the set of features (inputs to the model) to include those that are most relevant and have a nontrivial impact on the target. This is referred to as **feature selection**. The first course of action, if feasible, should be to directly inspect and evaluate the features to ensure that the values corresponding to that feature are reasonable, and to possibly identify an obvious subset of features for modeling, or at least determine features that can be excluded. If you have some domain knowledge, you might be able to use common sense to filter out some of the features in an ad-hoc manner.

Beyond initial manual filtering, the common approach to reducing dimensionality is to perform feature engineering, which can be categorized as either **feature selection** or **feature extraction**:

- **Feature selection** is the process of determining which features have the greatest impact on your model and downselecting to include only those features as inputs. This can be accomplished through one of the following types of techniques:
 - Filtering techniques, which assess each feature's impact through **mutual information criterion (MIC)**, **information gain**, **chi-square measure**, **odds ratio**, or **R-square measure**. These techniques neglect any interaction effects among features.
 - **Wrapper techniques** such as **LASSO** (Tibshirani 1996), **elastic net** (Zou and Hastie 2005), and **forward/backward/stepwise regression** techniques that iteratively determine which terms (including interactions) to include in the model.
 - **Embedded techniques**, which incorporate feature selection as an integral part of the training. One example is a **decision tree**, for which the splitting rules are based on determining the most influential variables to direct the splitting at each node (for example, the variables that best preserve the purity of the split).
- **Feature extraction** is the process of transforming the existing features into a lower-dimensional space, typically generating new features that are composites of the existing features. There are a number of techniques that reduce dimensionality through such a transformation process, including the following:
 - **Principal component analysis (PCA)** and **singular value decomposition (SVD)** are **unsupervised** feature transformations that strive to project the original features onto new orthogonal dimensions of maximum variance. The generated or extracted features that are the top contributors to variance can be preserved as a reduced set of features to be used as inputs for training the model. Standard PCA uses a **linear combination** of the existing features to achieve this; however, **kernel PCA** (Schölkopf, Smola, and Müller 1998) can be used to construct **nonlinear mappings** from the original features to new orthogonal dimensions of maximum variance. Although PCA can be performed via eigendecomposition of the square and symmetrical covariance matrix, it is often advantageous to compute the SVD of the rectangular data matrix instead. It is then trivial to obtain the principal components from the SVD, and you avoid the intermediate step of computing the covariance matrix. In addition, greater interpretability of the final feature set can be achieved using **sparse PCA** (Zou, Hastie, and Tibshirani 2005), which enforces sparsity constraints to achieve the linear combinations in only a few input variables, although orthogonality is lost.
 - **Nonnegative matrix factorization (NMF)** (Lee and Seung 2000) is another important **unsupervised** feature transformation. As in PCA (and SVD if you perform some algebraic manipulation), the original data matrix is decomposed into the product of two factor matrices. The distinctive feature of NMF is that all the entries of the factor matrices are constrained to be nonnegative. This constraint aids interpretability of the factors in many settings (for example, text analysis) where the factors can be thought of as probabilities.
 - **Autoencoding neural networks** extract a highly representative set of nonlinear features from the bottleneck layer of a specialized network.

An obvious drawback to feature extraction is that the actual inputs to the model are no longer meaningful with respect to the business problem. However, you can simply consider this another transformation of the original inputs to be provided to the model, something that must be accounted for as part of the scoring process when the model is deployed.

An excellent overview of feature engineering strategies is provided in (Cunningham 2007).

Whether you perform feature selection or feature extraction, your ultimate goal is to include the subset of features that describe most, but not all, of the variance and to reduce the signal-to-noise ratio in your data. Although intuition would tell you that elimination of features equates to a loss of information, in the end this loss is compensated for by the ability of the model to more accurately map the remaining features to the target in a lower-dimensional space. The result is simpler models, shorter training times, improved generalization, and a greater ability to visualize the feature space. As a side note, it is good practice to perform feature engineering before and after any imputation you might implement and compare the results; the effect of providing artificial values for values that were missing might cause different features to be selected.

Some high-dimensional data sets require special attention to perform feature extraction efficiently; one example is a data set of user ratings for items (such as movies) in which each column represents an item and each row is a user (or vice versa). Data sets such as this are very sparse and can be reduced by converting to a **sparse data representation** such as coordinate list (COO) format, in which only nonzero items (or items that actually have values) are preserved in a data set that has three columns for row, column, and value (user, item, and rating in this example). Model training time can be reduced exponentially simply by realizing that you have sparse data, converting the data set to a format such as COO, and using feature extraction techniques that are optimized for sparse data representations.

UNDERSTAND THE EXPECTATIONS OF THE MODEL CONSUMER

At some point, the model you ultimately generate is going to be used in some way to make predictions or other types of business decisions. A key consideration you must establish from the outset is whether the recommendations from the model will need to be justified by interpreting or explaining the logic behind how the model arrived at those conclusions; regulated environments might also have certain documentation requirements. Machine learning algorithms are usually formulated in a manner that emphasizes accuracy over interpretability. What makes these models accurate is literally what makes them difficult to understand: they are very complex. This is a fundamental tradeoff. If you know in advance that you will need to provide a business executive with some sort of reasoning behind a model's results other than "because that's what the model tells us," you might decide to limit your model candidates to those that can be more easily explained, such as regression and decision trees. It's much easier to describe the if-then hierarchical splitting logic of a decision tree than to describe the hyperplane that maximizes the margin between classes in the kernel-transformed feature space of a support vector machine. On the other hand, if you are simply looking for the most accurate generalizable model you can generate, then a more complex machine learning approach is appropriate. Either way, it's a question you should answer up front before you spend significant time and effort building models. Some guidance on answering this question is provided in the next section.

TRAINING

Now that you have ensured that you have sufficient and appropriate data, massaged the data into a form suitable for modeling, identified key features to include in your model, and established how the model is to be used, you are ready to make use of powerful machine learning algorithms to build predictive models or discover patterns in your data. This is really the phase where you should allow yourself more freedom to experiment with different approaches to identify the algorithms (and configuration of options for those algorithms) that produce the best model for your specific application. Still, as with proper data preparation, the process of training models cannot be entered into carelessly; an understanding of the main algorithm concepts and key concerns to heed will help ensure that you are using the appropriate algorithms and applying them judiciously.

OVERFITTING

A discussion of the primary issues and best practices for training models must start with the concept of overfitting. Recall that the goal is to build models that can be used to score future observations to enable you to make business decisions, such as to accept or deny credit application, flag fraudulent activity, identify tissue as cancerous or not, predict potential revenue, and so on. Machine learning algorithms are very effective at learning a mapping between the features and known target values in your existing data; if left unattended, they can often create a 100% accurate mapping, as shown in Figure 3a.

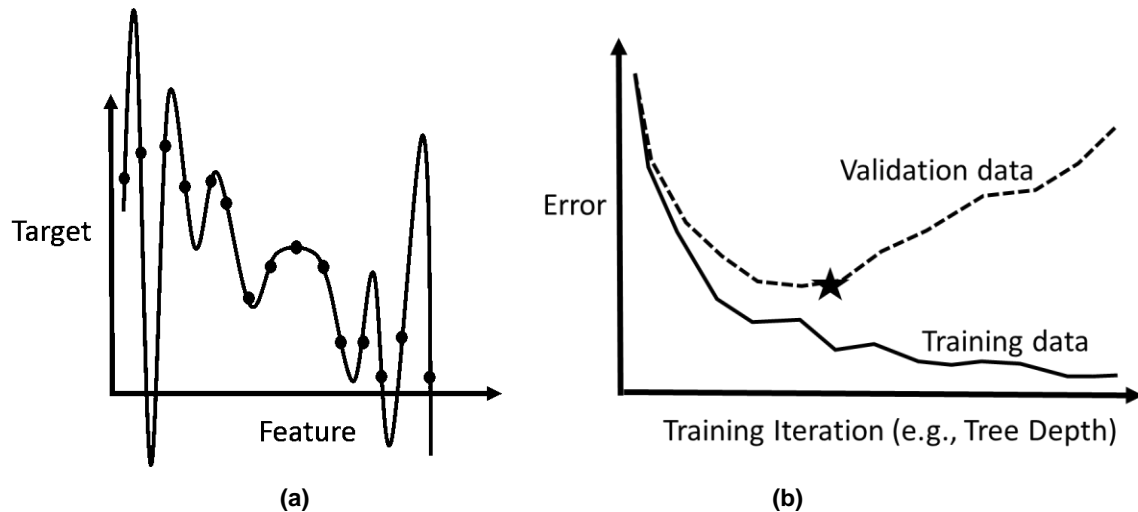


Figure 3. (a) Example of Overfitting, (b) Training and Validation Error Compromise

Clearly, a model that is complex enough to perfectly fit the existing data will not generalize well when used to score new observations. It might provide accurate answers for some observations by chance, but in general it does not represent the trend of the data. This is referred to as overfitting. A decision tree is another prime example of an algorithm that can easily overfit the data. If the tree is allowed to continue to split the data all the way down to each observation being in its own leaf, it will be 100% accurate for every observation in the training data. But after a certain depth, the tree is not providing any information that can be applied in general.

Honest Model Assessment

Certainly you are striving to achieve low training error, but it is just as important (or more important) to achieve low generalization error. The training process needs to account for this compromise and make an honest assessment of the accuracy of the model. Machine learning typically involves training a succession of candidate configurations toward selecting a final model, and the error of each candidate must be assessed at every iteration of the training process. Assessing a candidate model on the data that are used to train the model would direct the algorithm to overfit to that training data.

Honest assessment, which is highly related to the bias-variance tradeoff, involves calculating error metrics from scoring the model on data that were not used in any way during the training process. It is very important to understand the distinctions between validation and testing, and to incorporate them as part of your model training, assessment, and selection process.

- **Validation** data are holdout data that are used to assess the model *during* training for the purpose of selecting variables and adjusting the model parameters or hyperparameters in order to generate a more accurate, generalizable model. Validation data sets are instrumental in evaluating the bias-variance tradeoff and preventing overfitting. As shown in Figure 3b, the error evaluated on the validation set will actually start to increase at some point, indicating that attempting to fit the training data any more accurately will diminish the generalization capability of the model. In lieu of a separate holdout set (which might not be feasible for smaller data sets), **k-fold cross validation** can be performed. In this technique, the training data set is split into k subsets and each subset is held out and used for validation on a model that is trained by using the other $k-1$ subsets; the error is taken to

be the average across all of the models. Whether through a validation set or through cross validation, ensure that the training process assesses the error on data that are not used to train the model in order to avoid overfitting to the training data.

- **Test** data are holdout data that are used *at the end* of model fitting to obtain a final, honest assessment of how well the trained model generalizes to new data. The reason for using test data (instead of validation data) for an unbiased assessment is that validation data play a role in the model training process and hence would yield a biased assessment similar to assessment on training data. For this reason, a test data set should be used only at the end of the analysis and should not play a role in the model training process.

JUDICIOUS ALGORITHM SELECTION AND TUNING

Ultimately, the success of your machine learning application comes down to the effectiveness of the actual model you build. The popular “no free lunch” theorem (Wolpert 1996) states that no one model works best for every problem—selecting the appropriate algorithm and configuring the hyperparameters to tune that algorithm for maximum effectiveness are critical steps in the process.

Algorithm Selection

Selecting the modeling algorithm for your machine learning application can sometimes be the most difficult part. The decision of which algorithm to use can be guided by answering a few key questions:

- What is the size and nature of your data?

If you expect a fairly linear relationship between your features and your target, linear or logistic regression or a linear kernel support vector machine might be sufficient. Linear models are also a good choice for large data sets due to their training efficiency and due to the curse of dimensionality. As the number of features increase, the distance between points grows and observations are more likely to be linearly separable. To an extent, nonlinearity and interaction effects can be captured by adding higher-order polynomial and interaction terms in a regression model. However, as illustrated in Figure 4, more complex relationships can be modeled through the power of the more sophisticated machine learning algorithms such as decision trees, random forests, neural networks, and nonlinear kernel support vector machines. Of course, these more sophisticated algorithms can require more training time and might be unsuitable for very large data sets.

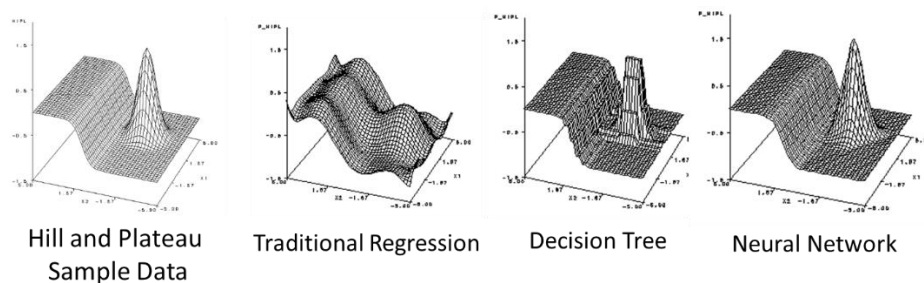



Figure 4. Various Models of a Complex Target Profile

- What are you trying to achieve with your model?

Are you creating a model to classify observations, predict a value for an interval target, detect patterns or anomalies, or provide recommendations? Answering this question will direct you to a subset of machine learning algorithms that specialize in the particular type of problem.

- How accurate does your model need to be?

Although you always want your model to be as accurate as possible when  applied to new data, it is still always good to strive for simplicity. Simpler models train faster and are easier to understand, making it easier to explain how and why the results were achieved. Simpler models are also easier to deploy. Start with a regression model as a benchmark, and then train a more complex model such as a neural net, random forest, or gradient boosted model. If your regression model is much less accurate than the more complex model, you have probably missed some important predictor or interaction of predictors. An additional benefit of a simpler model is that it will be less prone to overfitting the training data.

- How much time do you have to train your model?

This question goes hand-in-hand with the question of how accurate your model needs to be. If you need to train a model in a short amount of time, linear or logistic regression and decision trees are probably your best options. If training time is not an issue, take advantage of the powerful algorithms (neural networks, support vector machines, gradient boosting, and so on) that iteratively refine the model to better represent complex relationships between features and the target of interest.

- How interpretable or understandable does your model need to be?

It is very important to establish the expectations of your model consumer in regards to how explainable your model must be. If an uninterpretable prediction is acceptable, you should use as sophisticated an algorithm as you can afford in terms of time and computational resources. Train a neural network, a support vector machine, or any flavor of ensemble model to achieve a highly accurate and generalizable model. If interpretability or explainable documentation is important, use decision trees or a regression technique, and consider using penalized regression techniques, generalized additive models, quantile regression, or model averaging to refine your model.

If you need to ensure high accuracy but still need to explain the model results, a common approach is to train a complex model, use that model to generate predicted target values for all training observations, and then use these predicted values to train a decision tree. This decision tree is then essentially a surrogate model that acts as a proxy to the complex logic of the other algorithm. New observations are scored on the complex model for more accurate evaluation, but the surrogate model is used to explain the logic.

Table 1 in the Appendix provides a reference guide for the usage of the most common machine learning algorithms.

Even if you use these questions as guidance, selecting the single most effective algorithm to model your business problem can be very challenging. Experiment as much as you can afford to, and consider using an ensemble of multiple techniques, as described in the section “Ensemble Modeling” on page 16.

Regularization and Hyperparameter Tuning

The objective of a learning algorithm is to find the model parameters that minimize the loss function over the independent samples. For example, these parameters could be regression weights for a linear model, or they could be the adaptive weights on defining the connections within a neural network. As the complexity of your model increases, its predictive abilities often decrease after a certain point due to overfitting and multicollinearity issues. Hence, the resulting models often do not generalize well to new data, and they yield unstable parameter estimates.

Regularization methods help deal with overfitting models and multicollinearity problems by placing one or more penalties on the objective function that controls the size of the model weights. This penalty on the model weights decreases the variance of the model while increasing its bias. The total error of a model is the sum of its variance and bias. If the amount of penalty on the model weights is selected carefully, the

decrease in variance is less than the gain in bias, and hence the total error of the model decreases. This gives you a better model that has improved predictive abilities and more stable model parameters, and is known as the **bias-variance tradeoff**. Figure 5 illustrates the bias-variance tradeoff as it relates to the total error (Hastie, Tibshirani, and Friedman 2001).

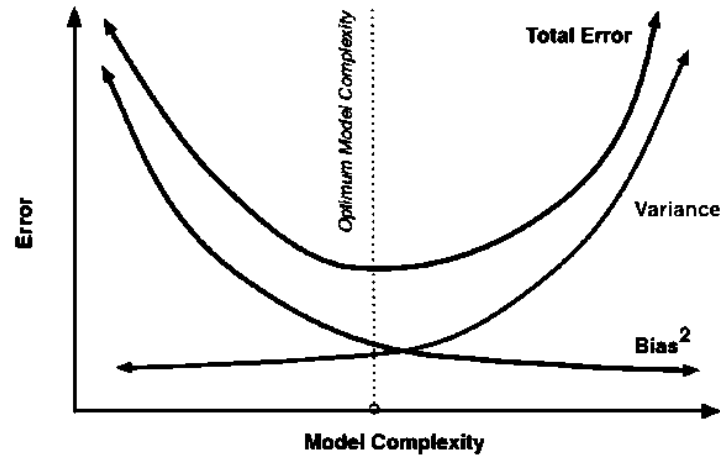


Figure 5: Bias-Variance Tradeoff

As a simple example, consider a linear regression model for which the penalty is placed on the squared error loss function as

$$\arg \min_{\beta} \{ \|Y - X\beta\|^2 + \lambda p(\beta) \}$$

where $\|Y - X\beta\|^2$ is the squared error loss function, β represents the vector of regression weights, λ is a hyperparameter (tuning or regularization parameter), and $p(\beta)$ defines the form of the penalty. The form of the penalty is defined by L1 regularization (sum of the absolute value of the regression coefficients) for LASSO regression (Tibshirani 1996) and by L2 regularization (sum of the square of the regression coefficients) for ridge regression. Both L1 and L2 regularization shrink model weights toward 0. L1 regularization performs feature selection by setting some of the weights exactly to 0, whereas L2 regularization never sets them to 0. Therefore, L2 regularization is good at dealing with multicollinearity issues by shrinking the correlated regression weights toward each other. Another advantage of L2 regularization is that it can be used with any type of learning algorithm; L1 regularization is more difficult to implement with some learning algorithms, in particular those that need to calculate gradient information.

Regularization methods are very useful techniques for reducing model overfitting, but they require you to set the hyperparameters to certain values. Hyperparameters do not necessarily need to be regularization parameters. For example, the number of hidden units for a neural network or the depth and number of leaves in a decision tree are also hyperparameters.

Optimal hyperparameter settings are extremely data-dependent; therefore, it is difficult to offer a general rule about how to identify a subset of important hyperparameters for a learning algorithm or how to find optimal values of each hyperparameter that would work for all data sets. Controlling hyperparameters of a learning algorithm is very important because proper control can increase accuracy and prevent overfitting.

Hyperparameter tuning is an optimization task, and each proposed hyperparameter setting requires the model training process to derive a model for the data set and evaluate results on the holdout or cross validation data sets. After evaluating a number of hyperparameter settings, the hyperparameter tuner provides the setting that yields the best performing model. The last step is to train a new model on the entire data set (which includes both training and validation data) under the best hyperparameter setting.

Strategies for hyperparameter tuning include the following:

- **Grid search:** In grid search, the sets of possible hyperparameter values are formed by assembling every possible combination of hyperparameter values. If the number of tuning parameters is not very large (two or three parameters), the grid search approach is feasible as long as the learning algorithms are computationally efficient. Grid search is also easy to implement, and **parallelization is straightforward**. However, **grid search suffers from the curse of dimensionality**; that is, as the number of hyperparameters increases, the number of value combinations grows exponentially.
- **Random search:** Bergstra and Bengio (2012) showed that random search can often perform as well as grid search in a much more computationally efficient way. This might seem surprising, but a simple probability example makes it easy to understand. Suppose you choose only 60 random points over the entire hyperparameter space. Now imagine a 5% region around the point where the best set of hyperparameter values lies. Each random draw then has a 5% chance of lying in that region. Thus, the probability of missing this space for all 60 sample points would be $(1 - 0.05)^{60} = 0.046$, which is a very small probability (less than 5%). If you assume that the optimal region of the hyperparameters occupies at least 5% of the hyperparameter space, then a random search that uses only 60 sets of hyperparameter values will find that region with a success probability of at least 0.95 ($1 - 0.046 = 0.954$). The simplicity and yet surprisingly reasonable performance of random search makes it a very effective method of hyperparameter tuning. Moreover, parallelization is trivial for random search (as it is for grid search), and random search is computationally much more efficient.
- **Experimental design:** Another popular method of hyperparameter tuning is choosing the trials according to an experimental design instead of choosing them randomly. These methods are referred to as low-discrepancy point sets because they attempt to ensure that points are approximately equidistant from one another to fill the space efficiently. Some examples of these methods include the Sobol sequences (Antonov and Saleev 1979) and Latin hypercube sampling (McKay 1992).
- **Smart tuning methods:** Instead of searching through all possible candidate hyperparameter setting combinations or randomly choosing them, smart tuning methods use intelligent optimization routines (such as genetic algorithms), which **start with a small set of points and use logic and information from those points to determine how to search through the space**. The search is **based on an objective of minimizing the model validation error**, so each “evaluation” from the optimization algorithm’s perspective is a full cycle of model training and validation. These methods are designed to make intelligent use of fewer evaluations and thus save on the overall computation time. However, unlike a grid search or experimental design approach, **tuning methods that are based on optimization routines are only partially parallelizable, depending on the optimization algorithm that is used**. For example, a genetic algorithm can evaluate each candidate set of hyperparameter values in a population in parallel, but it must carry out crossover and mutation steps to determine the next population to be evaluated. Ultimately, **a smart tuning method should find a more effective set of hyperparameter values than a random search or a discrete approach such as a grid search or experimental design finds**. A benchmark study of different tuners is provided in Konen et al. (2011).


Regardless of the approach you take, understand that significant predictive power can be gained by intelligently tuning the hyperparameters for the selected algorithm. Don’t settle for the defaults.


Ensemble Modeling

Even with an understanding of some of the basic guidelines for selecting an algorithm and incorporating hyperparameter tuning, determining the single most effective machine learning algorithm (and its tuning parameters) to use for a particular problem domain and data set is daunting—even for experts. Ensemble modeling can take some of that weight off your shoulders and can give you peace of mind that the

predictions are the result of a collaborative effort, or consensus, among multiple models that are trained either from different algorithms that approach the problem from different perspectives, or from the same algorithm applied to different samples or using different tuning parameter settings, or both.

In general, ensemble modeling is all about using many models in collaboration to combine their strengths, compensate for their weaknesses, and make the resulting model generalize better for future data. If you rely on building a single model to represent the relationships and behavior in whatever your application is (even if you try to tweak and optimize that model based on the data you currently have), by focusing on one modeling method alone you're most likely sacrificing accuracy and robustness when making decisions and predictions on future data.

Many popular and effective algorithms have been formulated specifically around this concept of "consensus" prediction. These algorithms generally fall into one of the following categories: 

- **Bagging:** Bagging (an abbreviation for bootstrap aggregating) is an approach in which multiple base learners (often decision trees) are trained on different sample sets of the data, which are randomly drawn with replacement, and their predictions are aggregated through a function such as majority voting or averaging. Bagging particularly focuses on combining the predictions of base learners to reduce variance and avoid overfitting; it is an efficient and ideal way to handle the bias-variance tradeoff. The **random forest** method developed by Breiman (2001) is a very effective and popular bagging algorithm that combines the predictions of multiple decision trees that are trained on different samples by using random subsets of the variables for splitting. Bagging algorithms are highly parallelizable.
- **Boosting:** In boosting (Shapire et al. 1998), a weight is applied to each observation in the training set and used as a probability distribution for sampling with replacement, a base learner (again often a decision tree) is trained, and misclassified points from the training set are provided higher weights for the next iteration of sampling and training. In this way, the algorithm is encouraged at each iteration to focus more on the points that it previously had difficulty classifying. A specific implementation of boosting called **gradient boosting** (Friedman 2001) adjusts the model on each iteration by using a random bootstrap sample to train a model to predict the residuals of previous models (which is theoretically equivalent to the stochastic gradient descent approach in optimization). Unlike bagging, boosting is a very sequential process. 
- **Stacking:** Stacking involves training multiple models by using a diverse set of strong learners and then applying a higher-level "combiner" algorithm to generate a model that includes the predictions of the member models as inputs. The output of this combiner algorithm is the final prediction as a consensus among the member models. For example, a regression model (or other modeling algorithm of your choice) can be fit to weighted predictions from the base learners. The **super learner** (van der Laan, Polley, and Hubbard 2007) is a specific implementation of stacking that has proven to be quite effective.
- **Custom combinations:** In general, any number of diverse models can be trained on the data set and aggregated in some fashion to combine their strengths and compensate for their weaknesses. Each algorithm for training a model assumes some form of relationship between the inputs and the target. Combining predictions from multiple different algorithms might produce a relationship of a different form than any one of the algorithm assumes. If two models specify different relationships and fit the data well, it's likely that their average will generalize to new data much better. If not, then maybe an individual model is adequate. In practice, the best way to know is to combine some models and compare the results.

Note that if you do choose to employ an ensemble model in your application, the business decisions made using that model in the future will be difficult to explain because you cannot explicitly interpret the logic behind combining the results of multiple models. However, as previously suggested, you can alleviate this concern by using a more explainable surrogate model (such as a decision tree) that is built from the predictions of the ensemble or by using a small ensemble of interpretable models.

COMPUTATIONAL RESOURCE EXPLOITATION

Most machine learning tasks are computationally intensive. Computational resources should be used prudently to ensure efficient data preprocessing, model training, and scoring of new data. Data preparation and modeling techniques must often be refined repeatedly by the human operator to obtain the best results. Waiting for days while data is preprocessed or a model is trained not only is frustrating, but also can lead to inferior results. Most work in any field is conducted under time constraints; machine learning is no different. **Preprocessing more data and training more models faster typically allows for more human iteration on the problem at hand and leads to better results.**

Many researchers and practitioners still rely on older, but trusted and revered, single-threaded algorithm implementations. During prototyping, multithreading or distributing new algorithms can also be neglected. Interpreted languages are also heavily relied on in research and practice. Single-threaded algorithms and interpreted languages fail to sufficiently exploit computational resources. Researchers and practitioners should consider taking advantage of computational resources in the following ways:

- **Concurrent execution of single-threaded algorithms:** Single-threaded algorithms are necessary in certain cases. However, contemporary computers enable the execution of multiple threads, and the **typical machine learning exercise requires the evaluation of multiple feature sets, tuning parameters, and optimization routines to find the best results.** Instead of running these trials in serial, run them in parallel on different threads.
- **Hardware acceleration:** For I/O intensive tasks, such as using databases or SAS® software for data preparation, use solid state hard drives (SSDs). For computationally intensive, but parallelizable, tasks such as matrix algebra, use graphical processing units (GPUs). Numerous libraries are available that allow the transparent use of GPUs from high-level languages.
- **Low-level languages and libraries:** Low-level languages such as C and Fortran, though often seen as more difficult to use, are much faster than interpreted languages. Java can also offer significant performance increases over interpreted languages. When you are faced with designing a computationally intensive machine learning application that will be used repeatedly and over a longer period of time, carefully consider the tradeoff between perceived development difficulties with lower-level languages and the performance drawbacks of interpreted languages. An easier approach to increase performance might be to exploit the multiple packages and libraries that enable compiled lower-level binaries to be called from interpreted languages. **A newer generation of tensor manipulation libraries even enables users of interpreted languages to transparently compile and then execute optimized low-level code on CPUs and GPUs.**
- **Distributed computing:** Designing algorithms for distributed computing environments, where data and tasks are split across many connected computers, can greatly reduce execution times. However, not all distributed environments are well suited for machine learning and not all machine learning algorithms are well suited for distributed computing. Generally, shared-nothing environments can present insurmountable implementation problems for sophisticated machine learning tasks, and algorithms or implementations that require the movement of large amounts of data across the network of the distributed environment can easily negate any potential computational timing gains.

DEPLOYMENT

METHOD OF DEPLOYMENT

Given that machine learning models tend to be difficult to interpret, their primary use is to create predictions that create value (monetary or otherwise) for an organization or other entity. The actual mechanism by which machine learning models will create their predictions requires thought and attention.

For example, making predictions on an individual's laptop is a good idea only for a limited time in most cases. If a model is really useful, it needs to be used by an organization in an operational manner to make decisions quickly, if not automatically. Keep in mind that some level of data preparation has likely been applied to the data set in its original, raw form, and this must be accounted for when making predictions on new observations. Moving the logic that defines all the necessary data preparation and mathematical expressions of a sophisticated predictive model from a development environment such as a personal laptop into an operational database is one of the most difficult and tedious aspects of machine learning. Mature, successful organizations are masters of this process—called “model deployment,” “deployment,” or “model production.”

To better understand model deployment, consider a credit card company. These companies often use logistic regression models to automatically authorize each transaction. This logistic regression model probably starts out as Python, R, or SAS code on an individual's laptop or workstation. However, because this model must be used millions of times a day in a massive number of simultaneous authorization decisions that are guaranteed to be made in milliseconds, the model simply cannot be run on an individual's laptop or workstation. Moreover, interpreted languages are probably too slow to guarantee millisecond response times. **The model probably needs to be ported into a compiled language, such as C or Java.** Model deployment is the process of moving the model from an individual's development environment to a large, powerful, and secure database or server where it can be used simultaneously by many mission-critical processes. Deployment as a web service that is programmatically accessible from custom applications and websites is another powerful and popular approach. Although deployment can require additional technical skills and knowledge beyond the analytics domain, many commercial machine learning software vendors provide this capability in a convenient, automated manner.

MONITORING AND UPDATING

Even after a model has been deployed, **it must be monitored.** Because models are often trained on static snapshots of data, their predictions typically become less accurate over time as the environment shifts away from the conditions that were captured in the training data. Consider a model for car insurance rates, which just 10 years ago did not account for behaviors such as texting while driving. Or consider a movie recommendation model that must adapt as viewers grow and mature through stages of life. **After a certain period of time, the error rate on new data surpasses a predefined threshold, and models must be retrained or replaced.** Champion-challenger testing is another common model deployment practice, in which a new, challenger model is compared against a currently deployed model at regular time intervals. When a challenger model outperforms a currently deployed model, the deployed model is replaced by the challenger, and the champion-challenger process is repeated. Yet another approach to refreshing a trained model is through online updates. Online updates continuously change the value of model parameters or rules based on the values of new, streaming data. **It is prudent to assess the trustworthiness of real-time data streams before implementing an online modeling system.**

CONCLUSION

The field of machine learning offers an array of powerful techniques for generating flexible and highly accurate models that empower businesses to effectively make use of vast amounts of data to make decisions. However, these techniques should not be considered to be push-button, black-box solutions that can be employed in an unattended manner. From data exploration and preparation, through model training, to ultimate deployment, these techniques must be used as part of a scientific approach to developing your overall machine learning application. The common issues and associated best practices described in this paper, although not necessarily comprehensive, offer some guidance on key points to consider in formulating a successful solution.

REFERENCES

- Antonov, I. A., and Saleev, V. M. (1979). "An Economic Method of Computing LP_T-Sequences." *USSR Computational Mathematics and Mathematical Physics* 19:252–256.
- Bellman, R. E. (1957). *Dynamic Programming*. Princeton, NJ: Princeton University Press.
- Bergstra, J., and Bengio, Y. (2012). "Random Search for Hyper-parameter Optimization." *Journal of Machine Learning Research* 13:281–305.
- Breiman, L. (2001). "Random Forests." *Machine Learning* 45:5–32.
- Cunningham, P. (2007). *Dimension Reduction*. Technical Report UCD-CSI-2007-7, University College Dublin.
- Dasu, T., and Johnson, T. (2003). *Exploratory Data Mining and Data Cleaning*. Hoboken, NJ: John Wiley & Sons.
- Donoho, D. (2015). "50 Years of Data Science." Available at <http://courses.csail.mit.edu/18.337/2015/docs/50YearsDataScience.pdf>.
- Friedman, J. H. (2001). "Greedy Function Approximation: A Gradient Boosting Machine." *Annals of Statistics* 29:1189–1232.
- Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York: Springer-Verlag.
- He, H., and Garcia, E. A. (2009). "Learning from Imbalanced Data." *IEEE Transactions on Knowledge and Data Engineering* 21:1263–1284.
- Huber, P. J. (1964). "Robust Estimation of a Location Parameter." *Annals of Mathematical Statistics* 35:73–101.
- Konen, W., Koch, P., Flasch, O., Bartz-Beielstein, T., Frieze, M., and Naujoks, B. (2011). "Tuned Data Mining: A Benchmark Study on Different Tuners." In *Proceedings of the Thirteenth Annual Conference on Genetic and Evolutionary Computation (GECCO-2011)*. New York: SIGEVO/ACM.
- Lee, D. D., and Seung, H. S. (2000). "Algorithms for Non-negative Matrix Factorization." In *Advances in Neural Information Processing Systems 13: Proceedings of the 2000 Conference*, edited by T. K. Leen, T. G. Dietterich, and V. Tresp, 556–562. Cambridge, MA: MIT Press.
- Lohninger, H. (1999). *Teach/Me Data Analysis*. Berlin: Springer-Verlag.
- Marcus, G. F. (1998). "Rethinking Eliminative Connectionism." *Cognitive Psychology* 37:243–282.
- McKay, M. D. (1992). "Latin Hypercube Sampling as a Tool in Uncertainty Analysis of Computer Models." In *Proceedings of the Twenty-Fourth Conference on Winter Simulation (WSC '92)*, edited by J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson, 557–564. New York: ACM.
- Mease, D., Wyner, A. J., and Buja, A. (2007). "Boosted Classification Trees and Class Probability/Quantile Estimation." *Journal of Machine Learning Research* 8:409–439.
- Rubin, D. (1987). *Multiple Imputation for Nonresponse in Surveys*. New York: John Wiley & Sons. Classic edition, published in 2004.

Schapire, R. E., Freund, Y., Bartlett, P., and Lee, W. S. (1998). "Boosting the Margin: A New Explanation for the Effectiveness of Voting Methods." *Annals of Statistics* 26:1651–1686.

Schölkopf, B., Smola, A., and Müller, K.-R. (1998). "Nonlinear Component Analysis as a Kernel Eigenvalue Problem." *Neural Computation* 10:1299–1319.

Scott, A. J., and Wild, C. J. (1986). "Fitting Logistic Models under Case-Control or Choice Based Sampling." *Journal of the Royal Statistical Society B* 48:170–182.

Spruyt, V. (2014). "The Curse of Dimensionality in Classification." *Computer Vision for Dummies* (blog). <http://www.visiondummy.com/2014/04/curse-dimensionality-affect-classification/>.

Tibshirani, R. (1996). "Regression Shrinkage and Selection via the Lasso." *Journal of the Royal Statistical Society B* 58:267–288.

Van der Laan, M. J., Polley, E. C., and Hubbard, A. E. (2007). "Super Learner." *UC Berkeley Division of Biostatistics Working Paper Series*, No. 222. <http://biostats.bepress.com/ucbbiostat/paper222>.

Vapnik, V.M. (1996). *The Nature of Statistical Learning Theory*. New York: Springer-Verlag.

Wickham, H. (2014). "Tidy Data." *Journal of Statistical Software* 59:1–23.

Wielenga, D. (2007). "Identifying and Overcoming Common Data Mining Mistakes." In *Proceedings of the SAS Global Forum 2007 Conference*. Cary, NC: SAS Institute Inc. <http://www2.sas.com/proceedings/forum2007/073-2007.pdf>.

Wolpert, D. H. (1996). "The Lack of A Priori Distinctions between Learning Algorithms." *Neural Computation* 8:1341–1390.

Zhang, J., and Mani, I. (2003). "kNN Approach to Unbalanced Data Distributions: A Case Study Involving Information Extraction." In *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003)*, Workshop on Learning from Imbalanced Data Sets II. Palo Alto, CA: AAAI Press.

Zou, H., and Hastie, T. (2005). "Regularization and Variable Selection via the Elastic Net." *Journal of the Royal Statistical Society B* 67:301–320.

Zou, H., Hastie, T., and Tibshirani, R. (2006). "Sparse Principal Component Analysis." *Journal of Computational and Graphical Statistics* 15:265–286.

ACKNOWLEDGMENTS

The authors would like to thank Jorge Silva and Anne Baxter for their contributions to this paper.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors:

Brett Wujek
SAS Institute Inc.
brett.wujek@sas.com

Patrick Hall
SAS Institute Inc.
patrick.hall@sas.com

Funda Güneş
SAS Institute Inc.
funda.gunes@sas.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

MACHINE LEARNING QUICK REFERENCE: ALGORITHMS - 1

Algorithm Type	Common Usage	Suggested Usage	Suggested Scale	Interpretability	Common Concerns
Penalized Regression	<ul style="list-style-type: none"> Supervised regression Supervised classification 	<ul style="list-style-type: none"> Modeling linear or linearly separable phenomena Manually specifying nonlinear and explicit interaction terms Well suited for $N \ll p$ 	Small to large data sets	High	<ul style="list-style-type: none"> Missing values Outliers Standardization Parameter tuning
Naïve Bayes	Supervised classification	<ul style="list-style-type: none"> Modeling linearly separable phenomena in large data sets Well-suited for extremely large data sets where complex methods are intractable 	Small to extremely large data sets	Moderate	<ul style="list-style-type: none"> Strong linear independence assumption Infrequent categorical levels
Decision Trees	<ul style="list-style-type: none"> Supervised regression Supervised classification 	<ul style="list-style-type: none"> Modeling nonlinear and nonlinearly separable phenomena in large, dirty data Interactions considered automatically, but implicitly Missing values and outliers in input variables handled automatically in many implementations Decision tree ensembles (e.g., random forests and gradient boosting) can increase prediction accuracy and decrease overfitting, but also decrease scalability and interpretability 	Medium to large data sets	Moderate	<ul style="list-style-type: none"> Instability with small training data sets Gradient boosting can be unstable with noise or outliers Overfitting Parameter tuning
k-Nearest Neighbors (kNN)	<ul style="list-style-type: none"> Supervised regression Supervised classification 	<ul style="list-style-type: none"> Modeling nonlinearly separable phenomena Can be used to match the accuracy of more sophisticated techniques, but with fewer tuning parameters 	Small to medium data sets	Low	<ul style="list-style-type: none"> Missing values Overfitting Outliers Standardization Curse of dimensionality
Support Vector Machines (SVM)	<ul style="list-style-type: none"> Supervised regression Supervised classification Anomaly detection 	<ul style="list-style-type: none"> Modeling linear or linearly separable phenomena by using linear kernels Modeling nonlinear or nonlinearly separable phenomena by using nonlinear kernels Anomaly detection with one-class SVM (OSVM) 	<ul style="list-style-type: none"> Small to large data sets for linear kernels Small to medium data sets for nonlinear kernels 	Low	<ul style="list-style-type: none"> Missing values Overfitting Outliers Standardization Parameter tuning Accuracy versus deep neural networks depends on choice of nonlinear kernel; Gaussian and polynomial often less accurate
Artificial Neural Networks (ANN)	<ul style="list-style-type: none"> Supervised regression Supervised classification Unsupervised clustering Unsupervised feature extraction Anomaly detection 	<ul style="list-style-type: none"> Modeling nonlinear and nonlinearly separable phenomena Deep neural networks (e.g., deep learning) are well-suited for state-of-the-art pattern recognition in images, videos, and sound All interactions considered in fully connected, multilayer topologies Nonlinear feature extraction with autoencoder and restricted Boltzmann machine (RBM) networks Anomaly detection with autoencoder networks Clustering and visualization with self-organizing maps (SOMs) 	<ul style="list-style-type: none"> Usually small to medium data sets Stochastic gradient descent (SGD) optimization drastically increases scalability 	Low	<ul style="list-style-type: none"> Missing values Overfitting Outliers Standardization Parameter tuning

MACHINE LEARNING QUICK REFERENCE: ALGORITHMS - 2

Algorithm Type	Common Usage	Suggested Usage	Suggested Scale	Interpretability	Common Concerns
Association Rules	<ul style="list-style-type: none"> Supervised rule building Unsupervised rule building 	Building sets of complex rules by using the co-occurrence of items or events in transactional data sets	Medium to large transactional data sets	Moderate	<ul style="list-style-type: none"> Instability with small training data Overfitting Parameter tuning
k-Means	Unsupervised clustering	<ul style="list-style-type: none"> Creating a known a priori number of spherical, disjoint, equally sized clusters k-modes method can be used for categorical data k-prototypes method can be used for mixed data 	Small to large data sets	Moderate	<ul style="list-style-type: none"> Missing values Outliers Standardization Correct number of clusters is often unknown Highly sensitive to initialization Curse of dimensionality
Hierarchical Clustering	Unsupervised clustering	Creating a known a priori number of nonspherical, disjoint, or overlapping clusters of different sizes	Small data sets	Moderate	<ul style="list-style-type: none"> Missing values Standardization Correct number of clusters is often unknown Curse of dimensionality
Spectral Clustering	Unsupervised clustering	Creating a data-dependent number of arbitrarily shaped, disjoint, or overlapping clusters of different sizes	Small data sets	Moderate	<ul style="list-style-type: none"> Missing values Standardization Parameter tuning Curse of dimensionality
Principal Components Analysis (PCA)	Unsupervised feature extraction	<ul style="list-style-type: none"> Extracting a data-dependent number of linear, orthogonal features, where $N \gg p$ Extracted features can be rotated to increase interpretability, but orthogonality is usually lost Singular value decomposition (SVD) is often used instead of PCA on wide or sparse data Sparse PCA can be used to create more interpretable features, but orthogonality is lost Kernel PCA can be used to extract nonlinear features 	<ul style="list-style-type: none"> Small to large data sets for traditional PCA and SVD Small to medium data sets for sparse PCA and kernel PCA 	Generally low, but higher for sparse PCA or rotated solutions	<ul style="list-style-type: none"> Missing values Outliers
Nonnegative Matrix Factorization (NMF)	Unsupervised feature extraction	Extracting a known a priori number of interpretable, linear, oblique, nonnegative features	Small to large data sets	High	<ul style="list-style-type: none"> Missing values Outliers Standardization Correct number of features is often unknown Presence of negative values
Random Projections	Unsupervised feature extraction	Extracting a data-dependent number of linear, uninterpretable, randomly-oriented features of equal importance	Medium to extremely large data sets	Low	Missing values
Factorization Machines	<ul style="list-style-type: none"> Supervised regression and classification Unsupervised feature extraction 	<ul style="list-style-type: none"> Extracting a known a priori number of uninterpretable, oblique features from sparse or transactional data sets Can automatically account for variable interactions Creating models from a large number of sparse features, can outperform SVM for sparse data 	Medium to extremely large sparse or transactional data sets	Moderate	<ul style="list-style-type: none"> Missing values Outliers Standardization Correct number of features is often unknown Less well suited for dense data