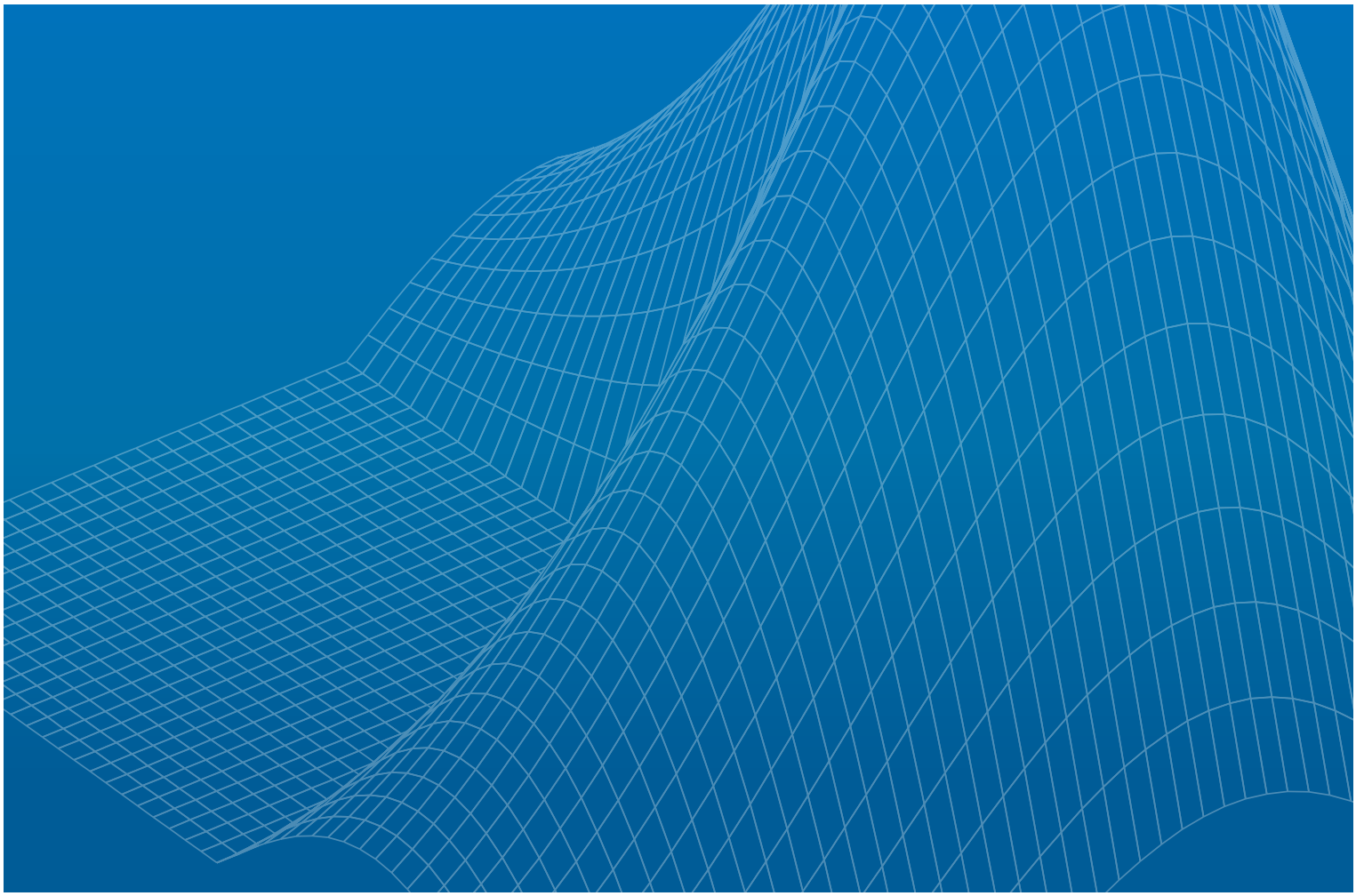


Machine Learning Challenges: Choosing the Best Model and Avoiding Overfitting

Modeling with machine learning is a challenging but valuable skill for any user working with data. No matter what you use machine learning for, chances are you have encountered a classification or overfitting concern along the way. This paper shows you how to mitigate the effects of these challenges using MATLAB®.

WHITE PAPER

MathWorks®

Classification Models

Choosing the Right Classification Model

What is a data classification model?

Classification models are used to assign items to a discrete group or class based on a specific set of features.

Why is it so hard to get right?

Each model has its own strengths and weaknesses in a given scenario. There is no cut-and-dried flowchart that can be used to determine which model you should use without grossly oversimplifying the considerations. Choosing a data classification model is also closely tied to the business case and a solid understanding of what you are trying to accomplish.

What can you do to choose the right model?

To begin with, make sure you can answer the following questions:

- How much data do you have and is it continuous?
- What type of data is it?
- What are you trying to accomplish?
- How important is it to visualize the process?
- How much detail do you need?
- Is storage a limiting factor?

When you're confident you understand the type of data you're going to be working with and what it will be used for, you can start looking at the strengths of various models. There are some generic rules of thumb to help you choose the best classification model, but these are just starting points. If you are working with a large amount of data (where a small variance in performance or accuracy can have a large effect), then choosing the right approach often requires trial and error to achieve the right balance of complexity, performance, and accuracy. The following sections describe some of the common models you should be aware of.

Naïve Bayes

If the data is not complex and your task is relatively simple, try a Naïve Bayes algorithm. It's a high bias/low variance classifier, which has advantages over logistic regression and nearest neighbor algorithms when working with a limited amount of data available to train a model.

Naïve Bayes is also a good choice when CPU and memory resources are a limiting factor. Because Naïve Bayes is very simple, it doesn't tend to overfit data, and can be trained very quickly. It also does well with continuous new data used to update the classifier.

If the data grows in size and variance and you need a more complex model, other classifiers will probably work better. Also, its simple analysis is not a good basis for complex hypotheses.

Naïve Bayes is often the first algorithm scientists try when working with text (think spam filters and sentiment analysis). It's a good idea to try this algorithm before ruling it out.

k-Nearest Neighbor

Categorizing data points based on their distance to other points in a training dataset can be a simple yet effective way of classifying data. k-nearest neighbor (*k*NN) is the "guilty by association" algorithm.

*k*NN is an instance-based lazy learner, which means there's no real training phase. You load the training data into the model and let it sit until you actually want to start using the classifier. When you have a new query instance, the *k*NN model looks for the specified *k* number of nearest neighbors; so if *k* is 5, then you find the class of 5 nearest neighbors. If you are looking to apply a label or class, the model takes a vote to see where it should be classed. If you're performing a regression problem and want to find a continuous number, take the mean of *f* values of *k* nearest neighbors.

Although the training time of *k*NN is short, actual query time (and storage space) might be longer than that of other models. This is especially true as the number of data points grows because you're keeping all the training data, not just an algorithm.

The greatest drawback to this method is that it can be fooled by irrelevant attributes that obscure important attributes. Other models such as decision trees are better able to ignore these distractions. There are ways to correct for this issue, such as applying weights to your data, so you'll need to use your judgment when deciding which model to use.

Decision Trees

To see how a decision tree predicts a response, follow the decisions in the tree from the root (beginning) node down to a leaf node which contains the response. Classification trees give responses that are nominal, such as true or false. Regression trees give numeric responses.

Decision trees are relatively easy to follow: You can see a full representation of the path taken from root to leaf. This is especially useful if you need to share the results with people interested in how a conclusion was reached. They are also relatively quick.

The main disadvantage of decision trees is that they tend to overfit, but there are ensemble methods to counteract this. Toshi Takeuchi has written a good example (for a Kaggle competition) that uses a ***bagged decision tree*** to determine how likely someone is to survive the Titanic disaster.

Support Vector Machine

You might use a support vector machine (SVM) when your data has exactly two classes. An SVM classifies data by finding the best hyperplane that separates all data points of one class from those of the other class (the best hyperplane for an SVM is the one with the largest margin between the two classes). You can use an SVM with more than two classes, in which case the model will create a set of binary classification subproblems (with one SVM learner for each subproblem).

There are a couple strong advantages of using an SVM: First, it is extremely accurate and tends not to overfit data. Second, once trained, it is a fast option because it's just deciding between one of two classes. Because SVM models are very fast, once your model has been trained you can discard the

training data if you have limited memory available. It also tends to handle complex, nonlinear classification very well.

However, SVMs need to be trained and tuned up front, so you need to invest time in the model before you can begin to use it. Also, its speed is heavily impacted if you are using the model with more than two classes.

Neural Networks

An artificial neural network (ANN) can learn, and therefore be trained to find solutions, recognize patterns, classify data, and forecast future events. People often use ANNs to solve more complex problems, such as character recognition, stock market prediction, and image compression.

The behavior of a neural network is defined by the way its individual computing elements are connected and by the strengths of those connections, or weights. The weights are automatically adjusted by training the network according to a specified learning rule until it performs the desired task correctly.

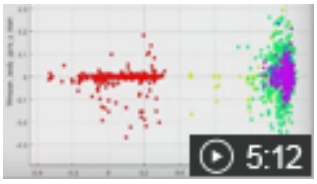
For experienced users, ANNs are great at modeling nonlinear data with a high number of input features. When used correctly, ANNs can solve problems that are too difficult to address with a straightforward algorithm. However, neural networks are computationally expensive, it is difficult to understand how an ANN has reached a solution (and therefore infer an algorithm), and fine-tuning an ANN is often not practical—all you can do is change the inputs of your training set and retrain.

Classification Cross-Validation

Cross-validation is a model assessment technique used to evaluate a machine learning algorithm's performance when making predictions on new datasets it has not been trained on. This is done by partitioning a dataset and using a subset to train the algorithm and the remaining data for testing. This technique is discussed in more detail in the Avoiding Overfitting section.

The Classification Learner App

MATLAB removes a lot of the hassle of figuring out which model works best with its Classification Learner app. You can also use MATLAB to judge how well your model is performing, and verify your results.



Classify Data Using the Classification Learner App

The Classification Learner app lets you train models and classify data using supervised machine learning.

Using Classification Learner, you can perform common machine learning tasks such as interactively exploring your data, selecting features, specifying validation schemes, training models, and assessing results. Choose from several classification types including decision trees, support vector machines, and k-nearest neighbors, and select from ensemble methods such as bagging, boosting, and random subspace.

Classification Learner helps you choose the best model for your data by enabling you to assess and compare models using confusion matrices and receiver operating characteristic (ROC) curves. You can export classification models to the MATLAB workspace to generate predictions on new data, or generate MATLAB code to integrate models into applications such as computer vision, signal processing, and data analytics.

The Classification Learner app supports:

- Decision trees: Deep tree, medium tree, and shallow tree
- Support vector machines: Linear SVM, fine Gaussian SVM, medium Gaussian SVM, coarse Gaussian SVM, quadratic SVM, and cubic SVM
- Nearest neighbor classifiers: Fine *k*NN, medium *k*NN, coarse *k*NN, cosine *k*NN, cubic *k*NN, and weighted *k*NN
- Ensemble classifiers: Boosted trees (AdaBoost, RUSBoost), bagged trees, subspace *k*NN, and subspace discriminant

With the app you can:

- Assess classifier performance using confusion matrices, ROC curves, or scatter plots
- Compare model accuracy using the misclassification rate on the validation set
- Improve model accuracy with advanced options and feature selection
- Export the best model to the workspace to make predictions on new data
- Generate MATLAB code to train classifiers on new data

Overfitting

Avoiding Overfitting

What is overfitting?

Overfitting means that your model is so closely aligned to training data sets that it does not know how to respond to new situations.

Why is overfitting difficult to avoid?

One of the reasons overfitting is difficult to avoid is that it is often the result of insufficient training data. The person responsible for the model may not be the same person responsible for gathering the data, who may not even realize how much data is needed to supply training and testing phases.

An overfit model returns very few errors which makes it look attractive at first glance. Unfortunately, there are too many parameters in the model in relation to the underlying system. The training algorithm tunes these parameters to minimize the loss function, but this results in the model being overfit to the training data, rather than the desired behavior of generalizing the underlying trends. When large amounts of new data are introduced to the network, the algorithm cannot cope and large errors begin to surface.

Ideally, your machine learning model should be as simple as possible and accurate enough to produce meaningful results. The more complex your model, the more prone it will be to overfitting.

How do you avoid overfitting?

The best way to avoid overfitting is by making sure you are using enough training data. Conventional wisdom says that you need a minimum of 10,000 data points to test and train a model, but that number depends greatly on the type of task you're performing (Naïve Bayes and *k*-nearest neighbor both require far more sample points). That data also needs to accurately reflect the complexity and diversity of the data the model will be expected to work with.

Using Regularization

If you've already started working and think your model is overfitting the data, you can try correcting it using ***regularization***. Regularization penalizes large parameters to help keep the model from relying too heavily on individual data points and becoming too rigid. The objective function changes so that it becomes $Error+\lambda f(\theta)$, where $f(\theta)$ grows larger as the components of (θ) grow larger and λ represents the strength of the regularization.

The value you pick for λ decides how much you want to protect against overfitting. If $\lambda=0$, you aren't looking to correct for overfitting at all. On the other hand, if the value for λ is too large, then your model will keep θ as small as possible (over having a model that performs well on your training set). Finding the best value for λ can take some time to get right.

Cross-Validation

An important step when working with machine learning is checking the performance of your model. One method of assessing a machine learning algorithm's performance is cross-validation. This technique has the algorithm make predictions using data not used during the training stage. Cross-validation partitions a dataset and uses a subset to train the algorithm and the remaining data for testing. Because cross-validation does not use all of the data to build a model, it is a commonly used method to prevent overfitting during training.

Each round of cross-validation involves randomly partitioning the original dataset into a training set and a testing set. The training set is then used to train a ***supervised learning*** algorithm, and the testing set is used to evaluate its performance. This process is repeated several times and the average cross-validation error is used as a performance indicator.

Common cross-validation techniques include:

- **k-fold:** Partitions data into k randomly chosen subsets (or folds) of roughly equal size. One subset is used to validate the model trained using the remaining subsets. This process is repeated k times, such that each subset is used exactly once for validation.
- **Holdout:** Partitions data into exactly two subsets (or folds) of specified ratio for training and validation.
- **Leaveout:** Partitions data using the k-fold approach, where k is equal to the total number of observations in the data. Also known as leave-one-out cross-validation.
- **Repeated random subsampling:** Performs *Monte Carlo* repetitions of randomly partitioning data and aggregates results over all the runs.
- **Stratify:** Partitions data such that both training and test sets have roughly the same class proportions in the response or target.
- **Resubstitution:** Does not partition the data; uses the training data for validation. Often produces overly optimistic estimates for performance and must be avoided if there is sufficient data.

MATLAB Functions That Check for Overfitting

Regularization techniques are used to prevent statistical overfitting in a predictive model. By introducing additional information into the model, regularization algorithms can handle multicollinearity and redundant predictors by making the model more parsimonious and accurate. These algorithms typically work by applying a penalty for complexity, such as adding the coefficients of the model into the minimization or including a roughness penalty.

Regularization for neural networks can be performed simply in Neural Network Toolbox™ by using the `trainbr` function. `trainbr` is a network training function that updates the weight and bias values according to Levenberg-Marquardt optimization. It minimizes a combination of squared errors and parameters, and then determines the correct combination to produce a network that generalizes well. The process is called Bayesian regularization.

The default method for improving generalization is called *early stopping*. This technique is automatically provided for all of the supervised network creation functions, including backpropagation network creation functions such as `feedforwardnet`.

With the early stopping technique, the available data is divided into three subsets. The first subset is the training set, which is used for computing the gradient and updating the network parameters and biases. The second subset is the validation set. The error on the validation set is monitored during the training process. The validation error normally decreases during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set typically begins to rise. When the validation error increases for a specified number of iterations (`net.trainParam.max_fail`), the training is stopped, and the parameters and biases at the minimum of the validation error are returned. The third subset is the test set, which is used on the fully trained classifier after the training and validation phases to compare the different models.

Early stopping and regularization can ensure network generalization when you apply them properly. For early stopping, you must be careful not to use an algorithm that converges too rapidly. If you are

using a fast algorithm (such as `trainlm`), set the training parameters so that the convergence is relatively slow. For example, set mu to a relatively large value, such as 1, and set `mu _ dec` and `mu _ inc` to values close to 1, such as 0.8 and 1.5, respectively. The training functions **`trainscg`** and `trainbr` usually work well with early stopping.

With early stopping, the choice of the validation set is also important. The validation set should be representative of all points in the training set.

Cross-Validation in MATLAB

Statistics and Machine Learning Toolbox™ has two functions that are particularly useful when performing cross-validation: `cvpartition` and `crossval`.

You use the `cvpartition` function to create a cross-validation partition for data. Using a k-fold cross-validation example, the syntax would be:

`c = cvpartition(n,'KFold',k)` constructs an object `c` of the `cvpartition` class defining a random partition for k-fold cross-validation on n observations.

`c = cvpartition(group,'KFold',k)` creates a random partition for a stratified k-fold cross-validation.

Use `crossval` to perform a loss estimate using cross-validation. An example of the syntax for this is:

`vals = crossval(fun,X)` performs 10-fold cross-validation for the function fun, applied to the data in X.

`fun` is a function handle to a function with two inputs, the training subset of `X, XTRAIN`, and the test subset of `X, XTEST`, as follows:

```
testval = fun(XTRAIN,XTEST)
```

For more information, see ***Improve Neural Network Generalization and Avoid Overfitting*** and ***Cross-Validation in MATLAB***.

Conclusion

Machine learning proficiency requires a combination of diverse skills, but the apps, functions, and training examples found in MATLAB can make mastering this technique an achievable goal. This paper outlines just some of the challenges that are common to machine learning practitioners. To learn more useful techniques, watch the ***Machine Learning Made Easy*** webinar, which covers several examples of typical workflows for both supervised learning (classification) and unsupervised learning (clustering).

92974v00 03/16