Mastering Machine Learning

A Step-by-Step Guide with MATLAB

MathWorks®

Introduction

This ebook builds on *Machine Learning with MATLAB*, which reviewed machine learning basics and introduced supervised and unsupervised techniques.

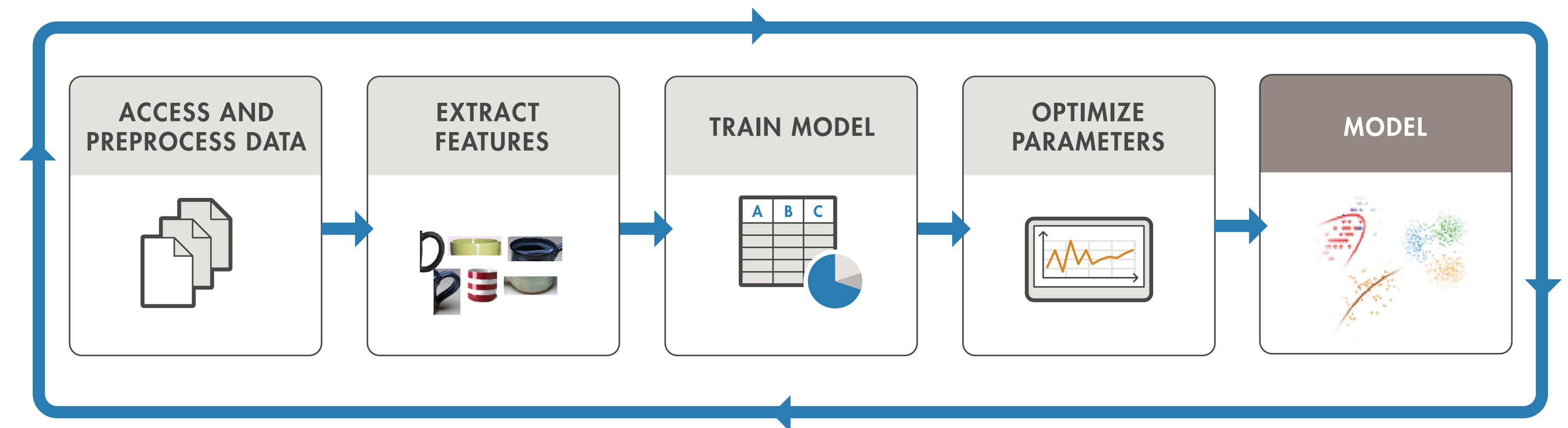
Using a heart sounds classifier as an example, we take you through the complete workflow for developing a real-world machine learning application, from loading data to deploying a trained model. For each training phase, we demonstrate the techniques that are critical to achieving accurate models, and help you master the more challenging training tasks, including selecting algorithms, optimizing model parameters, and avoiding overfitting.

In this ebook you'll also learn how to turn a model into a predictive tool by training it on new data, extracting features, and generating code for deployment on an embedded device.

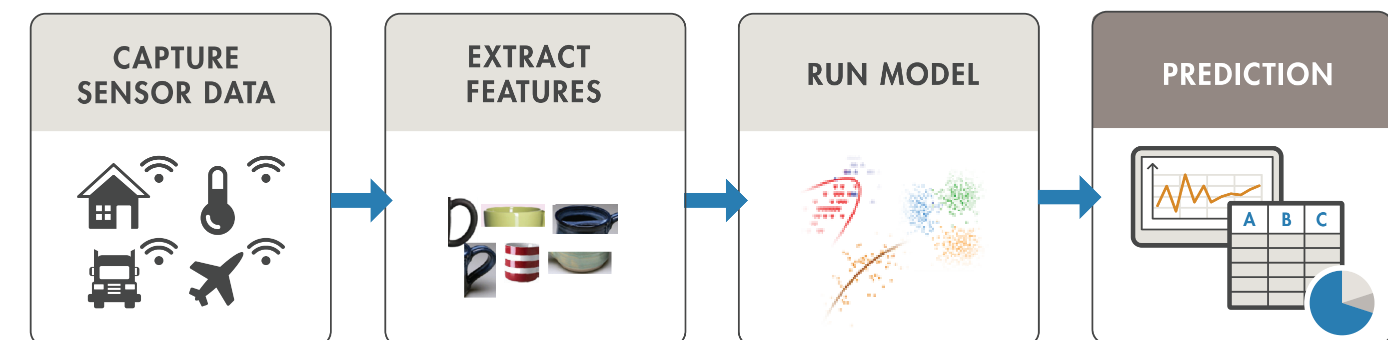
Review the Basics

Machine Learning Overview 3:02
Machine Learning with MATLAB

TRAIN: Iterate until you achieve satisfactory performance.



PREDICT: Integrate trained models into applications.



MathWorks

Building a Heart Sounds Classification Application with MATLAB

Heart sounds are a rich source of information for early diagnosis of cardiac pathologies. Distinguishing normal from abnormal heart sounds requires a specially trained clinician. Our goal is to develop a machine learning application that can identify abnormal heart sounds. Using a heart sounds monitoring application, regular medical staff could screen for heart conditions when no specialist is available, and patients could monitor themselves.

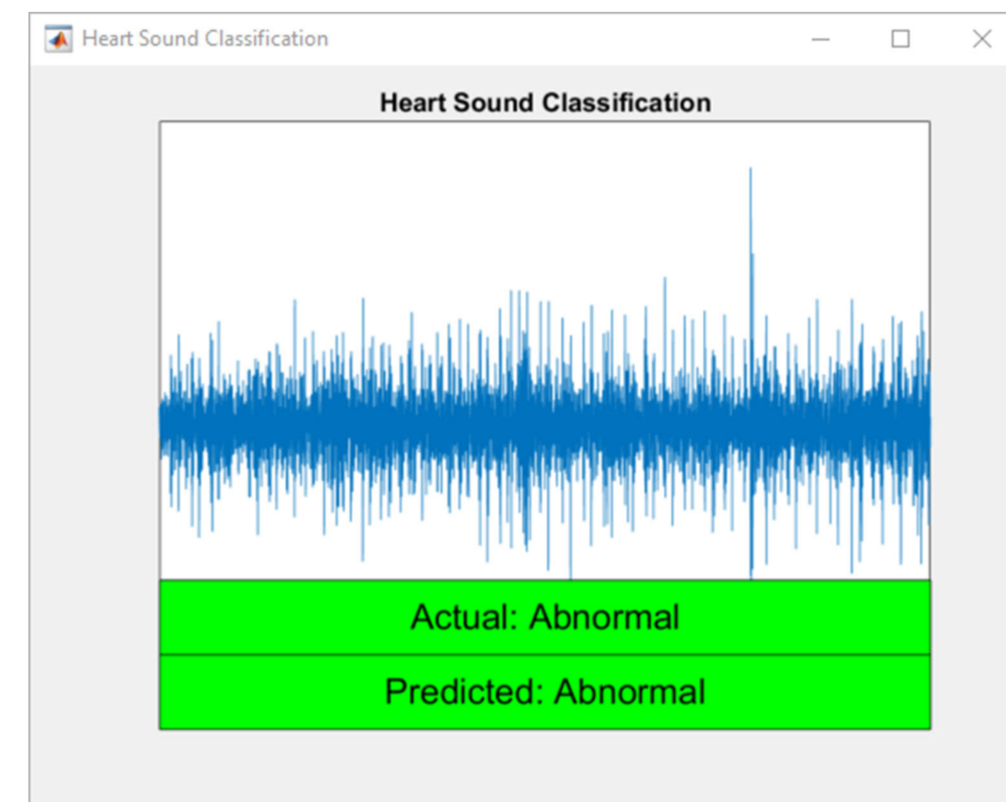
In developing this application, we'll follow these steps:

1. Access and explore the data.
2. Preprocess the data and extract features.
3. Develop predictive models.
4. Optimize the model.
5. Deploy analytics to a production system.

We encourage you to work through the example yourself. Simply download the MATLAB® code and follow the "Hands-On Exercise" callouts throughout the ebook.

Tools You'll Need

Download a *free 30-day trial of MATLAB for Machine Learning.*
Download *MATLAB code for the heart sounds classification application.*



Schematic heart sounds classifier and prototype app.

MathWorks®

STEP 1. Access and Explore the Data

Our example uses the dataset from the 2016 PhysioNet and Computing in Cardiology challenge, which consists of thousands of recorded heart sounds ranging in length from 5 seconds to 120 seconds.

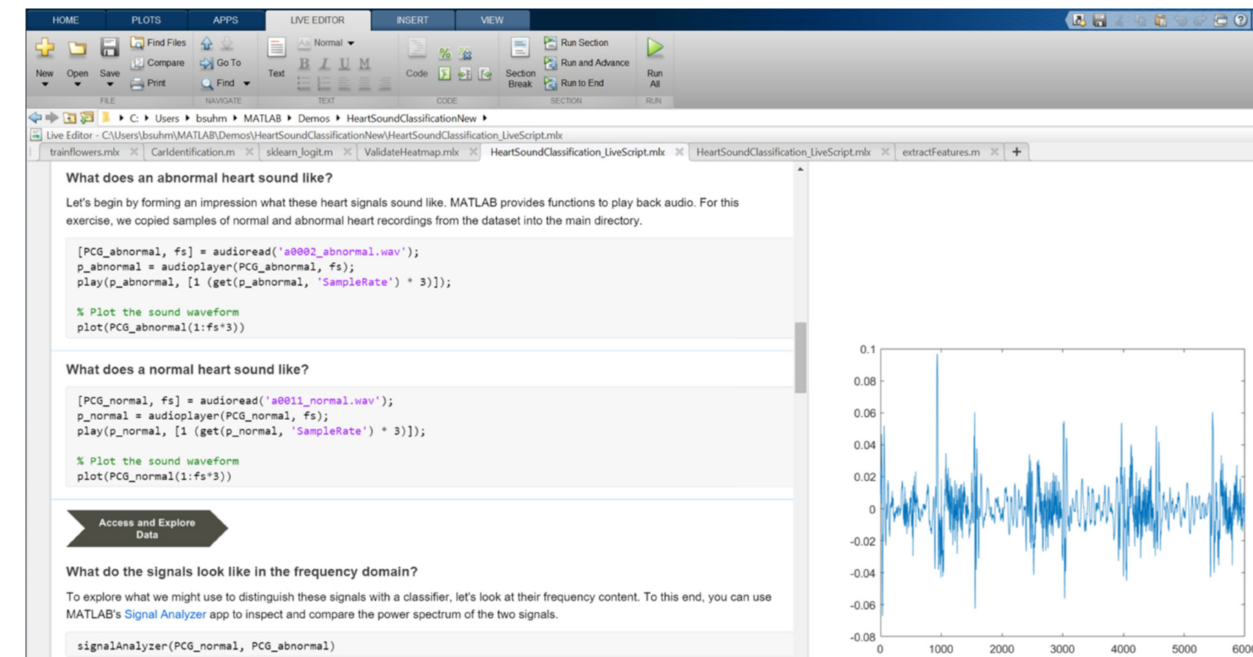
> **Hands-On Exercise**
>
> Execute the first section of the Live Editor script. The script will download the heart sounds dataset from the 2016 PhysioNet and Computing in Cardiology challenge to your local workspace.

The dataset includes 3240 recordings for model training and 301 recordings for model validation. After downloading the data, we store the training and validation sets in separate folders—a standard procedure in machine learning.

Exploring the Data

The first step in any machine learning project is understanding what kind of data you are working with. Common ways to explore data include inspecting some examples, creating visualizations, and (more advanced) applying signal processing or clustering techniques to identify patterns.

To understand what is involved in distinguishing normal from abnormal heart sounds, let's begin by listening to some examples.



Plot of an abnormal heart sound, generated in the MATLAB Live Editor.

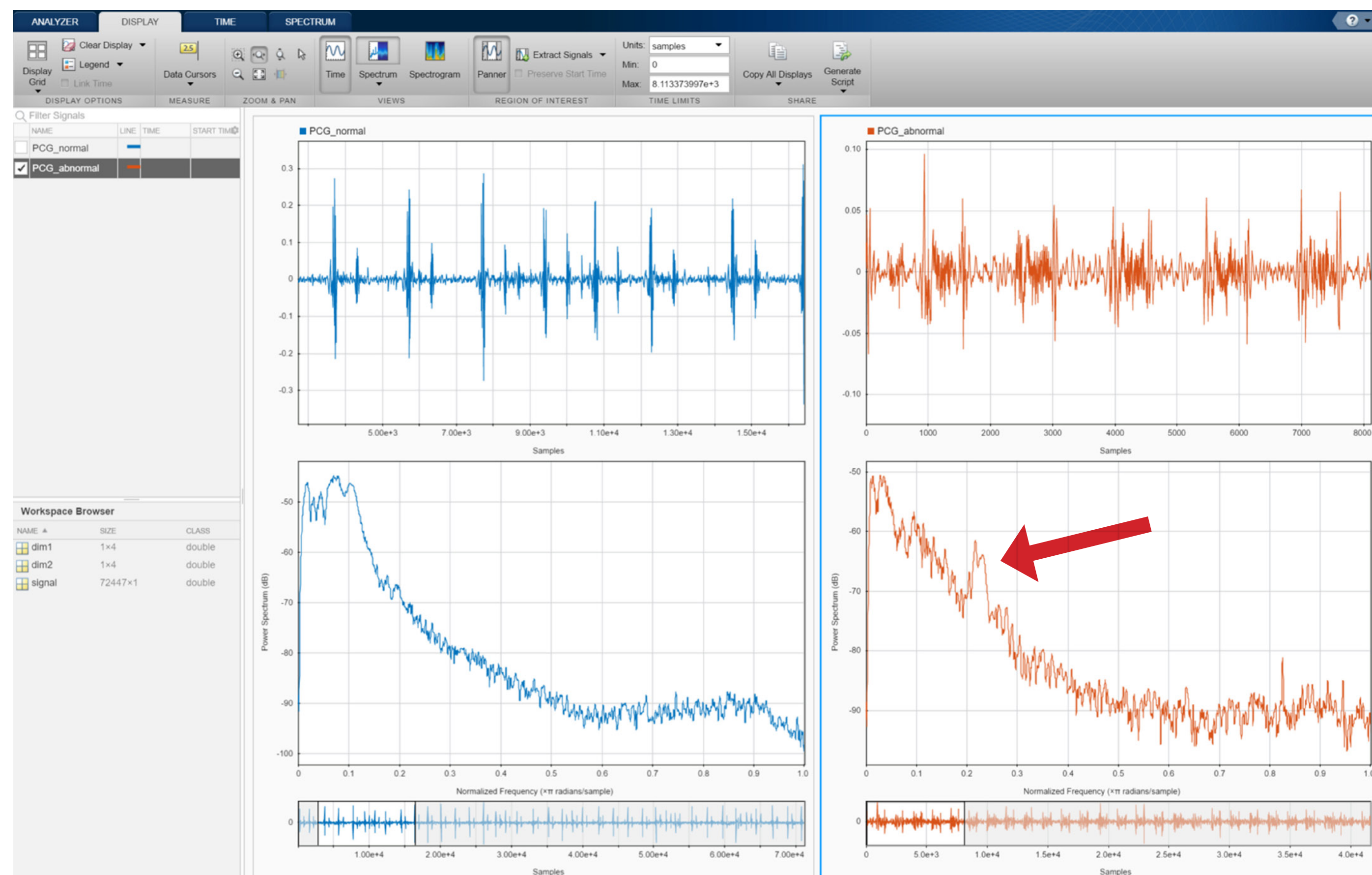
> **Hands-On Exercise**
>
> Listen to the examples of abnormal and normal heart sounds provided in the main directory of the code example. MATLAB provides **Audio Read** and **Audio Player** for working with audio files. These files are used in the "What does an (ab)normal heart sound like?" sections of the example scripts.

You might notice that the abnormal heart sound has higher frequencies, with noise between beats. The normal heart sound is more regular, with silence between beats.

MathWorks®

Analyzing the Signals

We can get a deeper understanding of the differences between these signals—without writing any code—by using the Signal Analyzer app in *Signal Processing Toolbox™* to view the signals side-by-side in the frequency domain.



Normal heart sound (left) and abnormal (right), shown in the Signal Analyzer app.
The red arrow indicates a spike in frequency content at around 200 Hz for the abnormal heart sound.

After this initial exploration of the data and the classification task, we load all the data into memory for preprocessing. MATLAB makes it easy to load large datasets that are distributed across multiple directories: it creates a handle on the complete dataset, which can then be read into memory in one or multiple chunks. For large datasets, MATLAB can distribute execution across multiple compute resources.

Each recording is an audio file labeled either abnormal or normal. Because we know the true category ("ground truth") of each file in the dataset, we can apply *supervised machine learning*, which takes a known set of input data and known responses to the data (output) and trains a model to generate reasonable predictions for the response to new data.

Hands-On Exercise

To load the training data and corresponding true category into memory, execute the sections "Prepare to read the data into memory" and "Create a table with filenames and labels" in the example.

MathWorks®

STEP 2. Preprocess the Data and Extract Features

Most datasets require some preprocessing before feature extraction—typical tasks include removing outliers and trends, imputing missing data, and normalizing the data. These tasks are not required for our dataset because it has already been preprocessed by the organizers of the PhysioNet challenge.

Extracting and selecting features helps improve a machine learning algorithm by focusing on the data that's most likely to produce accurate results.

Extracting Features

Feature extraction is one of the most important parts of machine learning because it turns raw data into information that's suitable for machine learning algorithms. Feature extraction eliminates the redundancy present in many types of measured data, facilitating generalization during the learning phase. Generalization is critical to avoiding overfitting the model to specific examples.

Learn More

The ebook *Machine Learning with MATLAB* describes common feature extraction techniques for sensor, image, video, and transactional data.

Selecting Features

While feature extraction is the first step, we need to avoid using too many features. A model with a larger number of features requires more computational resources during the training stage, and using too many features leads to overfitting.

The challenge is to find the minimum number of features that will capture the essential patterns in the data.

Feature selection is the process of selecting those features that are most relevant for your specific modeling problem and removing unneeded or redundant features. Common approaches to feature selection include stepwise regression, sequential feature selection, and regularization.

Feature extraction can be time-consuming if you have a large dataset and many features. To speed up this process, you can distribute computation across available cores (or scale to a cluster) using the **`parfor`** loop construct in *Parallel Computing Toolbox™*.

In the heart sounds example, based on our knowledge of signal classification, we extract the following types of features:

- Summary statistics: mean, median, and standard deviation
- Frequency domain: dominant frequency, spectrum entropy, and Mel Frequency Cepstral Coefficients (MFCC)

Extracting these types of features listed above yields 26 features from the audio signal.

Hands-On Exercise

The "Preprocess Data" section of the example script generates these features from the heart sound files, but since that process takes several minutes, by default pregenerated features are simply read from the file `FeatureTable.mat`. To perform the feature extraction, remove (or rename) this file before running the section.

For developing the model, the features must be captured in either table or matrix format. In our example, we built a table where each column represents a feature. The figure below shows some of the Mel Frequency Cepstras and the sound label (the true category of 'normal' or 'abnormal' heart sounds).

CC8	MFCC9	MFCC10	MFCC11	MFCC12	MFCC13	class
11315	-0.28488	1.6218	-0.53338	-1.6926	-2.0239	'Abnormal'
2.394	0.10001	2.9168	-1.3413	-0.90557	-1.4914	'Abnormal'
.1322	-0.42672	2.3943	1.5946	-2.0933	-1.3693	'Abnormal'
.8257	0.865	2.4926	-0.91656	-0.55254	-2.2298	'Abnormal'
.5196	-0.64708	3.923	-0.5634	-1.7582	-0.4827	'Abnormal'

(Partial) feature table.

MathWorks®

STEP 3. Develop Predictive Models

Developing a predictive model is an iterative process involving these steps:

i. Select the training and validation data.
ii. Select a classification algorithm.
iii. Iteratively train and evaluate classification models.

i. Selecting the Training and Validation Data

Before training actual classifiers, we need to divide the data into a training and a validation set. The validation set is used to measure accuracy during model development. For large datasets such as the heart sounds data, holding out a certain percentage of the data is appropriate; cross-validation is recommended for smaller datasets because it maximizes how much data is used for model training, and typically results in a model that generalizes better.

When you choose "No validation," the model is trained and evaluated on the entire dataset—no data is held out for validation. Retraining the model on the entire dataset can significantly affect its performance, especially if you have a very limited dataset. And knowing how accurately the model performs on the training set gives you guidance on which approaches to take to further improve the model.

MathWorks®

ii. Selecting a Classification Algorithm

No single machine learning algorithm works for every problem, and identifying the right algorithm is often a process of trial and error. However, being aware of key characteristics of various algorithms empowers you to choose which ones to try first, and to understand the tradeoffs you are making. The following table lists characteristics of popular classification algorithms.

For the heart sounds example, we use the *Classification Learner app* to quickly compare classifiers.

Hands-On Exercise

You can launch the Classification Learner app from the Apps tab or programmatically by typing `ClassificationLearner` in the command window. Once you start a new session, select `feature_table` as the data to work with, use all 26 features extracted in the previous step (for now), and select "Holdout Validation" with 25% of data held out as the validation method.

Algorithm	Prediction Speed	Training Speed	Memory Usage	Required Tuning	General Assessment
Logistic Regression (and Linear SVM)	Fast	Fast	Small	Minimal	Good for small problems with linear decision boundaries
Decision Trees	Fast	Fast	Small	Some	Good generalist, but prone to overfitting
(Nonlinear) SVM (and Logistic Regression)	Slow	Slow	Medium	Some	Good for many binary problems, and handles high-dimensional data well
Nearest Neighbor	Moderate	Minimal	Medium	Minimal	Lower accuracy, but easy to use and interpret
Naïve Bayes	Fast	Fast	Medium	Some	Widely used for text, including spam filtering
Ensembles	Moderate	Slow	Varies	Some	High accuracy and good performance for small- to medium-sized datasets
Neural Network	Moderate	Slow	Medium to Large	Lots	Popular for classification, compression, recognition, and forecasting

MathWorks®

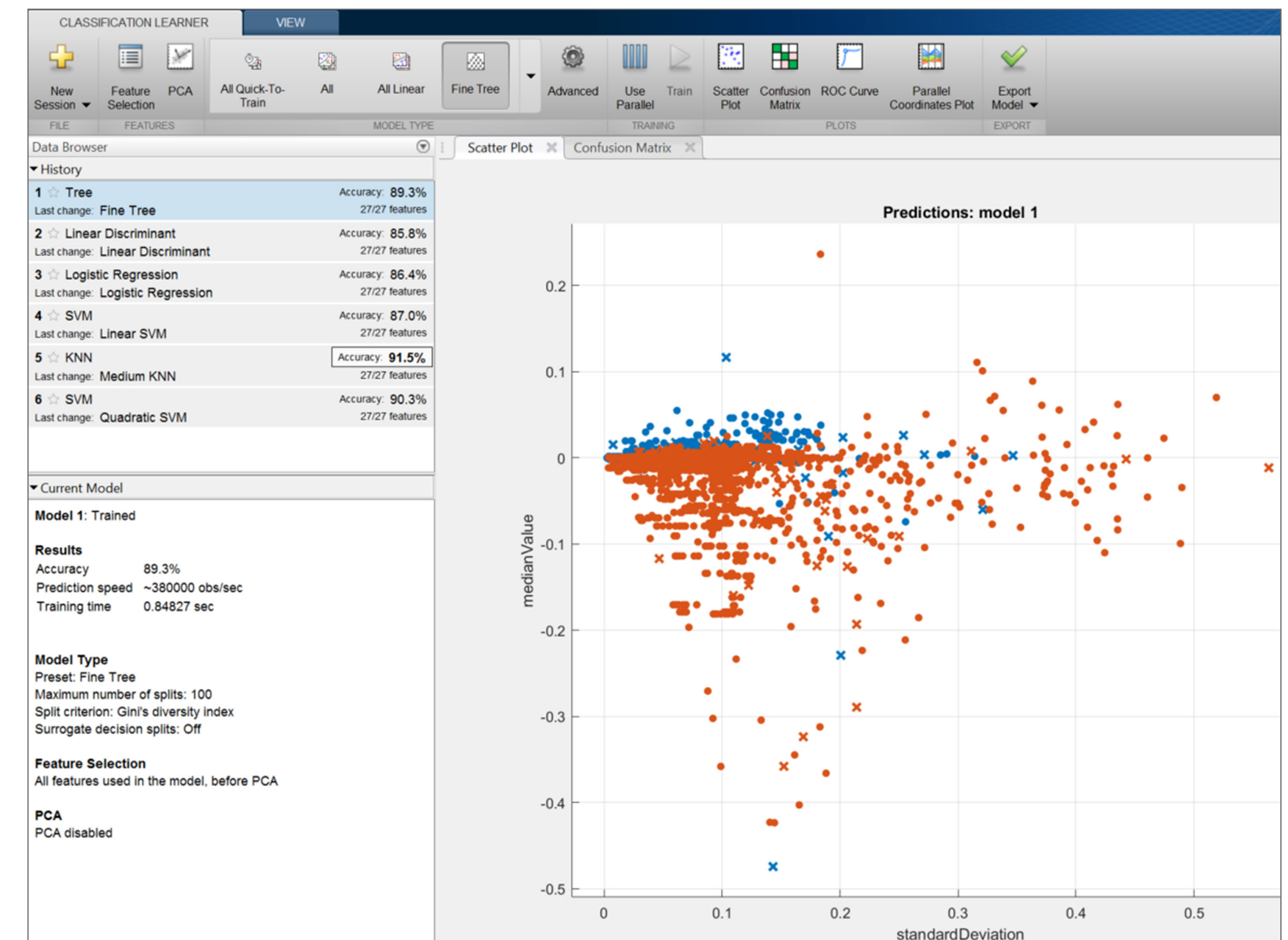
iii. Iteratively Training and Evaluating Classification Models

We are now ready to begin the iterative training and evaluation of models. We can either follow the brute force approach and run all the algorithms (this is easy to do with the Classification Learner app), or start with the algorithms that seem to best fit the specific classification task.

You can select an individual classifier or multiple classifiers simultaneously, such as "All support vector machines." You can then train classifiers in parallel and compare their performance on your data. For each classifier that you train, the accuracy is estimated either on the held-out validation data or using cross validation, depending on which data you selected for validation in the previous step.

For our heart sounds application, the initial results suggest that the ("Fine") K-nearest neighbors (KNN) classifier performs well, followed by the quadratic support vector machine (SVM) and (Fine) decison tree.

Our initial heart sounds classifier achieves an accuracy rate above 90%. This sounds good, but it is still insufficient for a heart-screening application. How can we further improve the model's performance?



Initial comparison of multiple classification algorithms.

Learn More

For an overview of the training workflow, watch
Classify Data Using the Classification Learner App 5:12.

MathWorks

STEP 4. Optimize the Model

To improve model performance, aside from trying other (more complex) algorithms, we need to make changes to the process. Common approaches focus on one of the following aspects of machine learning model development:

- **Tuning model parameters.** We can almost always improve performance by changing key model parameters from their default settings.
- **Adding to or modifying training data.** Adding training data helps until we reach the point of overfitting (when the error rate starts to increase). Additional preprocessing can remedy any issues with the data itself that we may have overlooked, such as corrupted data, outliers, or missing values.
- **Transforming or extracting features.** If our current feature set doesn't capture all the variation inherent in the data, extracting additional features may help. By contrast, if we see signs of overfitting, we can try further reducing the feature set by applying a reduction technique such as principal component analysis (PCA), linear discriminant analysis (LDA), or singular value decomposition (SVD). If the features vary widely in scale, feature transformations such as normalization may help.
- **Making task-specific tradeoffs.** If some misclassifications are less desirable than others, we can apply a cost matrix to assign different weights to specific prediction classes (for example, misclassifying an actual heart condition as normal).

Since our model achieved almost the same level of accuracy on the test data as it did on the training data, adding more training data is unlikely to improve it. To further improve the accuracy of our heart sound classifier, we will first try a more complex algorithm and then introduce bias and tune the model parameters.

Hands-On Exercise

Run the "Split data into training and testing sets" section in the example script so that you can evaluate the impact of the various optimization techniques on the held-out test data.

Trying a More Complex Classifier

Using an individual classification tree tends to overfit the training data. To overcome this tendency, we'll try ensembles of classification trees (commonly known as "random forests"). With the heart sounds data, a bagged decision tree ensemble achieves 93% accuracy—in this case no improvement over the best individual classifier.

Introducing Bias

So far, we have assumed that all classification errors are equally undesirable, but that is not always the case. In the heart sounds application, for example, a *false negative* (failing to detect an actual heart condition) has much more severe consequences than a *false positive* (erroneously classifying a normal heart sound as abnormal).

To explore the tradeoff between different kinds of misclassification, we'll use a confusion matrix. (We can easily obtain a confusion matrix by switching from the scatter plot view in the Classification Learner app.) The confusion matrix for the heart sound classifier shows that our preliminary models misclassify only 5% of normal sounds as abnormal while classifying 12% of abnormal sounds as normal. In other words, while we mistakenly flag only 5% of healthy hearts, we fail to detect more than 10% of actual heart conditions, a situation that is clearly unacceptable in medical practice.

This example demonstrates how analyzing performance based solely on overall accuracy—almost 94% for this model—can be misleading. This situation occurs when the data is unbalanced, as in this case, where our dataset contains roughly four times as many normal as abnormal heart sounds.

To improve task-specific performance, we will bias our classifier to minimize misclassifications of actual heart conditions. The tradeoff, that the model will misclassifiy more normal heart sounds as abnormal, is acceptable.

Assessing bias in classifiers with the Classification Learner app.

MathWorks®

The standard way to bias a classifier is to introduce a cost function that assigns higher penalties to undesired misclassifications.

The following code introduces a cost function that penalizes misclassifications of abnormal sounds by a factor of 20.

Develop Predictive Models

Train the classifier with misclassification cost

To compensate for fewer 'Abnormal' observations in the data, and to bias the classifier towards fewer misclassifications of abnormal sounds, we use a cost function that assigns higher misclassification cost to the 'Abnormal' class. At the same time, we perform hyperparameter tuning by using Bayesian Optimization to find optimal values for model parameters.

Since the ensemble of trees outperformed the SVM classifier in the Classification Learner, we continue with the ensemble.

```
% Assign higher cost for misclassification of abnormal heart sounds
C = [0, 20; 1, 0];

% Create a random sub sample (to speed up training) from the training set
%subsample = randi([1 height(training_set)], round(height(training_set)/4), 1);
subsample = [1:height(training_set)];

rng(1);

% Create a 5-fold cross-validation set from training data
cvp = cvpartition(length(subsample),'KFold',5);

if ~exist('TrainedEnsembleModel.mat')
    % perform training only if we don't find a saved model

    % train ensemble of decision trees (random forest)
    disp("Training Ensemble classifier...")

    % bayesian optimization parameters (stop after 15 iterations)
    opts = struct('Optimizer','bayesopt','ShowPlots',true,'CVPartition',cvp,...
          'AcquisitionFunctionName','expected-improvement-plus','MaxObjectiveEvaluations',15);
    trained_model = fitcensemble(training_set(subsample,:),'class','Cost',C,...
        'OptimizeHyperparameters',{'Method','NumLearningCycles','LearnRate'},...
        'HyperparameterOptimizationOptions',opts)

    save('TrainedEnsembleModel2', 'trained_model');
else
    % load previously saved model
    load('TrainedEnsembleModel.mat')
end

% Predict class labels for the validation set using trained model
% NOTE: if training ensemble without optimization, need to use trained_model.Trained{idx} to predict
predicted_class = predict(trained_model, testing_set);
```

```
Estimated objective function value = 0.077005
Function evaluation time = 467.9228

Best estimated feasible point (according to models):
      Method      NumLearningCycles    LearnRate
    __________    _________________    _________

    AdaBoostM1    468                  0.95583

Estimated objective function value = 0.077005
Estimated function evaluation time = 468.8481

trained_model =
  classreg.learning.classif.ClassificationEnsemble
                      PredictorNames: {1×27 cell}
                        ResponseName: 'class'
               CategoricalPredictors: []
                          ClassNames: {'Abnormal'  'Normal'}
                      ScoreTransform: 'none'
                     NumObservations: 9111
    HyperparameterOptimizationResults: [1×1 BayesianOptimization]
                          NumTrained: 468
                              Method: 'AdaBoostM1'
                        LearnerNames: {'Tree'}
                ReasonForTermination: 'Terminated normally after completin
                             FitInfo: [468×1 double]
                  FitInfoDescription: {2×1 cell}

  Properties, Methods
```

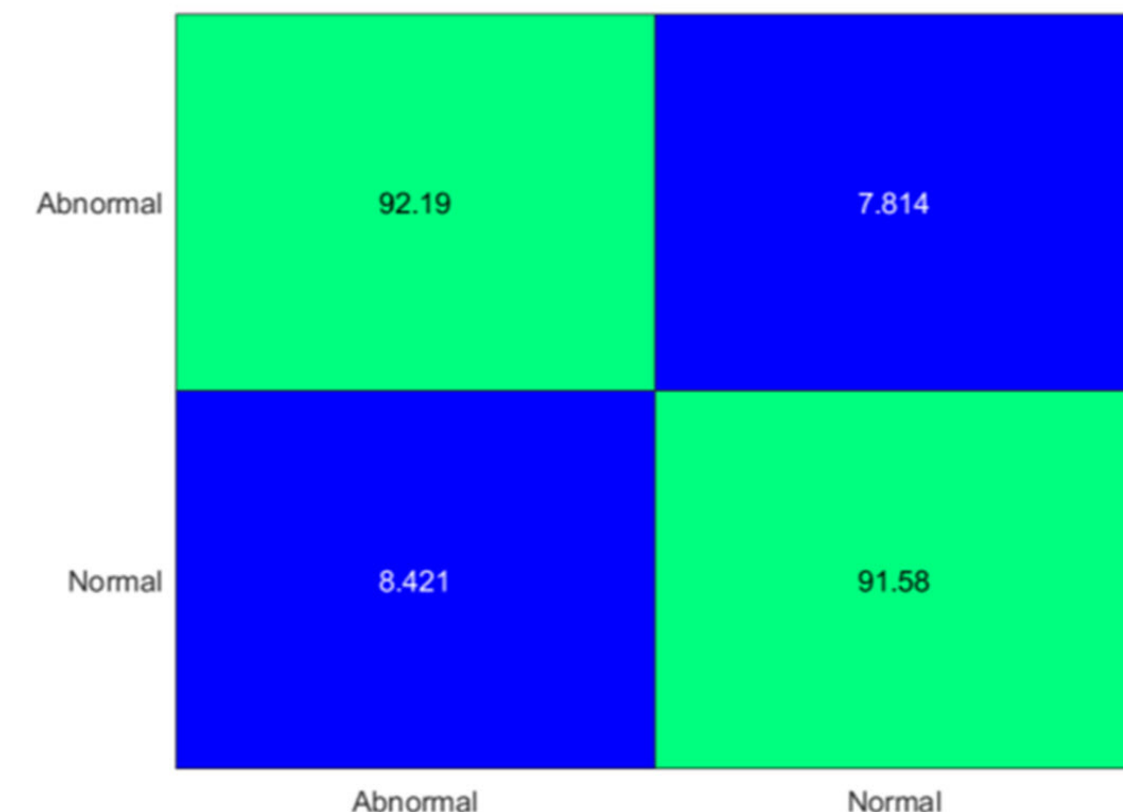
Min objective vs. Number of function evaluations

Min observed objective
Estimated min objective

Function evaluations
Min objective

Striking a different balance between accuracy and tolerance for false negatives by introducing a cost function.

The confusion matrix shows that the resulting model fails to detect fewer than 8% of abnormal sounds while misclassifying slightly more normal sounds as abnormal (8%, compared with 5% in the unbiased model). The overall accuracy of this model remains high at 92%.



Reducing misclassification of abnormal sounds by using a cost function.

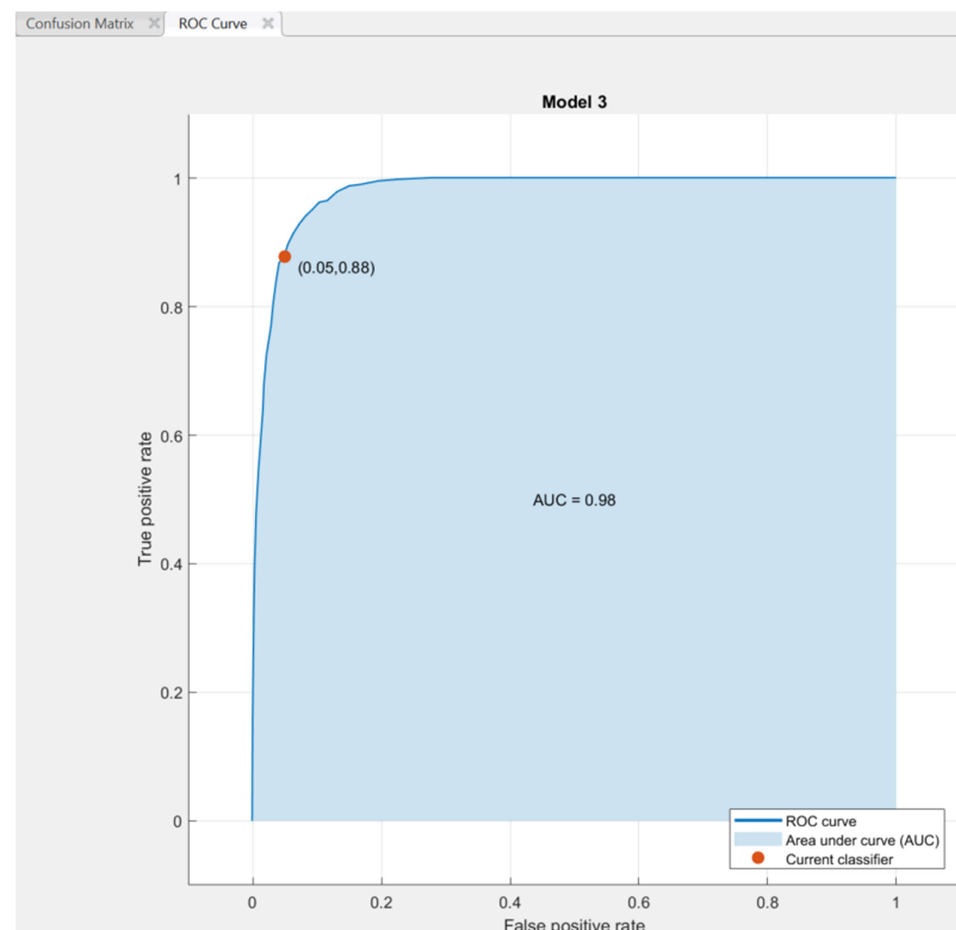
Hands-On Exercise

Run the "Train the classifier with misclassification cost" section in the example script. The script applies the cost matrix and performs Bayesian optimization of hyperparameters at the same time.

MathWorks®

Tuning Model Parameters

Instead of using scatter plots and confusion matrices to explore the tradeoff between true and false positives, we could use a receiver operating characteristic (ROC) curve. The ROC curve is a useful tool for visually exploring the tradeoff between true positives and false positives. You can use an ROC curve to select the best operating point or cutoff for your classifier to minimize the overall "cost" as defined by your cost function.



ROC curve for the bagged tree ensemble classifier.

As we've seen, machine learning algorithm parameters can be tuned to achieve a better fit. The process of identifying the set of parameters that provides the best model is often referred to as "hyperparameter tuning." To make hyperparameter tuning more efficient and to improve the odds of finding optimal parameter values, we can use automated Grid search and Bayesian optimization in MATLAB.

Grid search exhaustively searches a finite set of parameter value combinations, but it can take a long time.

Bayesian optimization develops a statistical model of the hyperparameter space and aims to minimize the number of experiments needed to find the optimal values.

The example script performs the hyperparameter tuning using Bayesian optimization while simultaneously introducing the cost function.

Learn More

Bayesian Optimization Characteristics

MathWorks®

Using Feature Selection to Correct Misclassification and Overfitting

So far, we've used all 26 features that we extracted when we trained the model. A final approach for optimizing performance entails feature selection: removing features that are either redundant or not carrying useful information. This step reduces computational cost and storage requirements, and it results in a simpler model that is less likely to overfit.

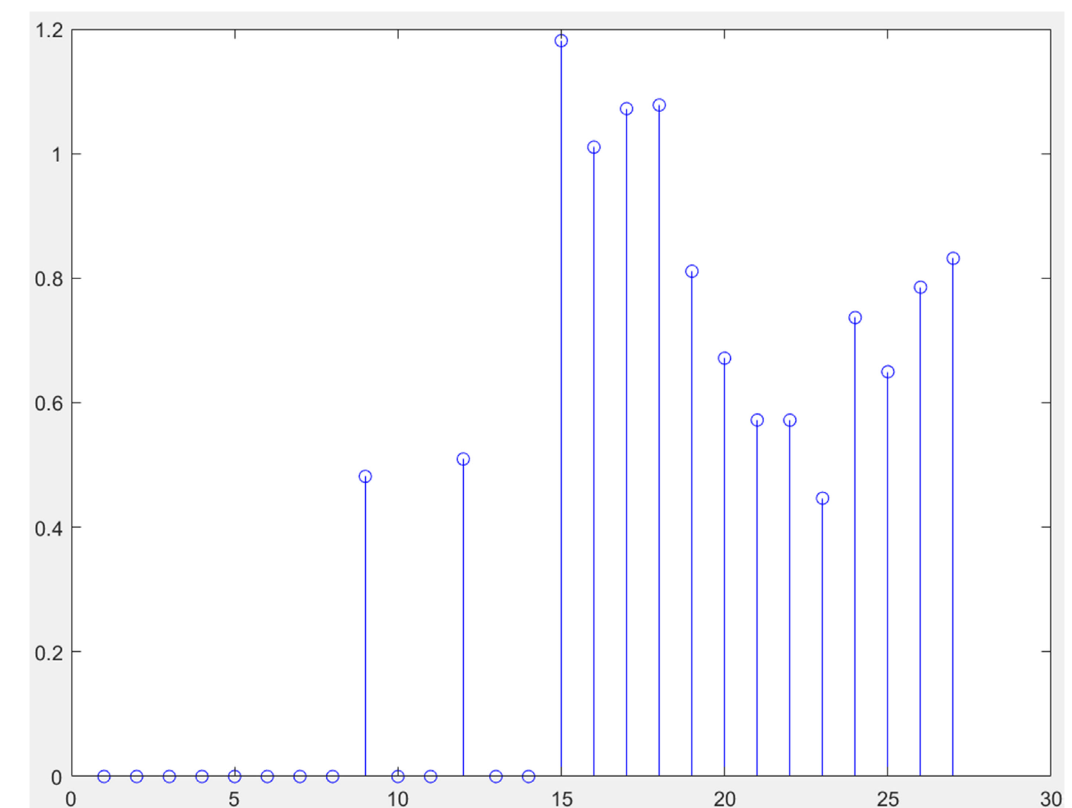
Reducing the feature set is important for the heart sounds application because it reduces the size of the model, making it easier to deploy on an embedded device.

Feature selection approaches include systematically evaluating feature subsets (which is computationally very expensive) and incorporating feature selection into the model construction process by applying weights to each feature (using fewer features helps minimize the objective function used during training).

For our heart sounds classifier, we apply neighborhood component analysis (NCA), a powerful feature selection technique that can handle very high-dimensional datasets. NCA reveals that about half the features do not contribute significantly to the model. We can therefore reduce the number of features from 26 to 15.

Hands-On Exercise

Run the "Perform feature selection using Neighborhood Component Analysis" section in the example script, followed by "Train model with selected features."



Outcome of automated feature selection, identifying the most relevant features using neighborhood component analysis.

Learn More

Selecting Features for Classifying High-Dimensional Data

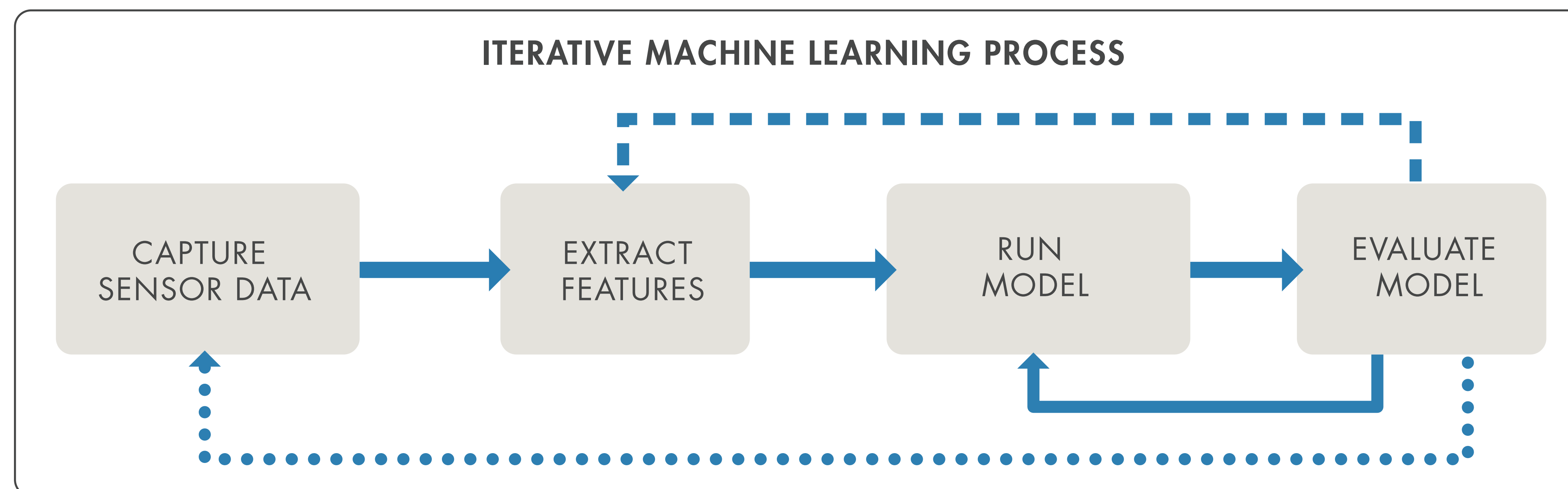
MathWorks®

To evaluate the performance impact of selecting just 15 features, we retrain the model with hyperparameter tuning and the cost function activated. The updated model fails to detect just 6% of abnormal heart conditions—slightly better than before—but it falsely classifies 15% of normal conditions as abnormal, slightly reducing overall accuracy. In medical practice, any positive result would be followed up with additional tests, eliminating most of the 15% false positives from the initial screening.

Iterating to Try Other Algorithms

To further improve the model, we could try the same series of optimization steps with different algorithms, since how much improvement the various optimization techniques yields varies with the algorithm. You can repeat the iterative process described in the previous section—for example, revisiting the KNN algorithm, which performed well initially. You might even go back to the feature extraction phase and look for additional features. To identify the best model, it is invariably necessary to iterate between the different phases of the machine learning workflow. You have mastered machine learning when you can infer what to try next from evaluating your current model.

In our heart sounds example, we are ready move on to the next and final step: deploying the classifier.



STEP 5. Deploy Analytics to a Production System

Machine learning applications can be deployed into production systems on desktops, in enterprise IT systems (either onsite or in the cloud), and embedded systems. For desktops and enterprise IT systems, you can deploy MATLAB applications on servers by compiling the code either as a standalone application or for integration with an application written in another language, such as C/C++, Python®, Java®, or .NET. Many embedded applications require model deployment in C code.

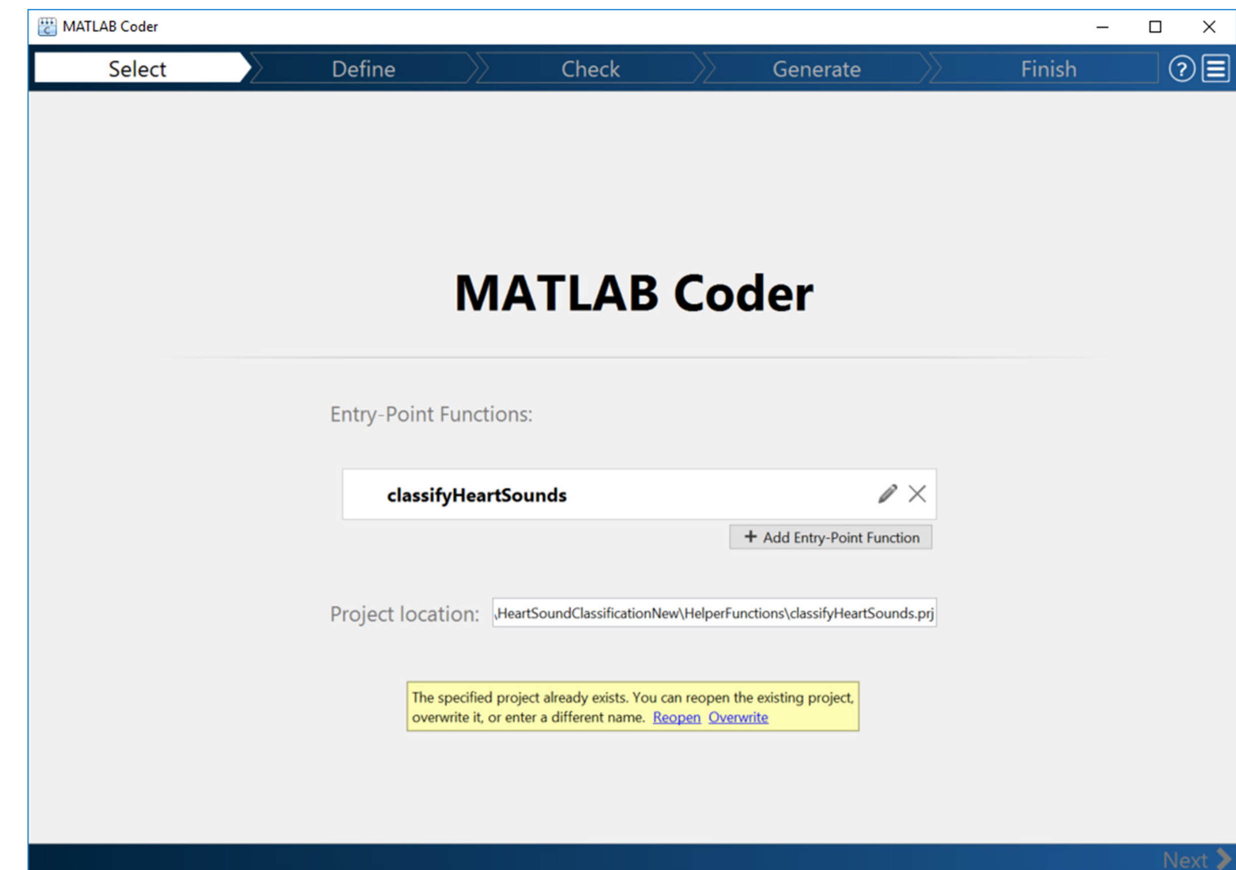
MATLAB Coder™ makes deployment on embedded systems easy by automatically translating MATLAB into C code.

Generating C Code

Our heart sounds diagnostic application will run on a medical device such as a wearable heart monitor or a mobile app. To prepare the application for deployment, we generate C code from the model by performing the following steps:

1. Save the trained model as a compact model.
2. Launch MATLAB Coder.
3. Create an entry point function that takes raw sensor data as input and classifies the patient's heart sound as either normal or abnormal.

MATLAB Coder automatically generates the corresponding C code.



C-code generation with MATLAB Coder.

Tools You'll Need

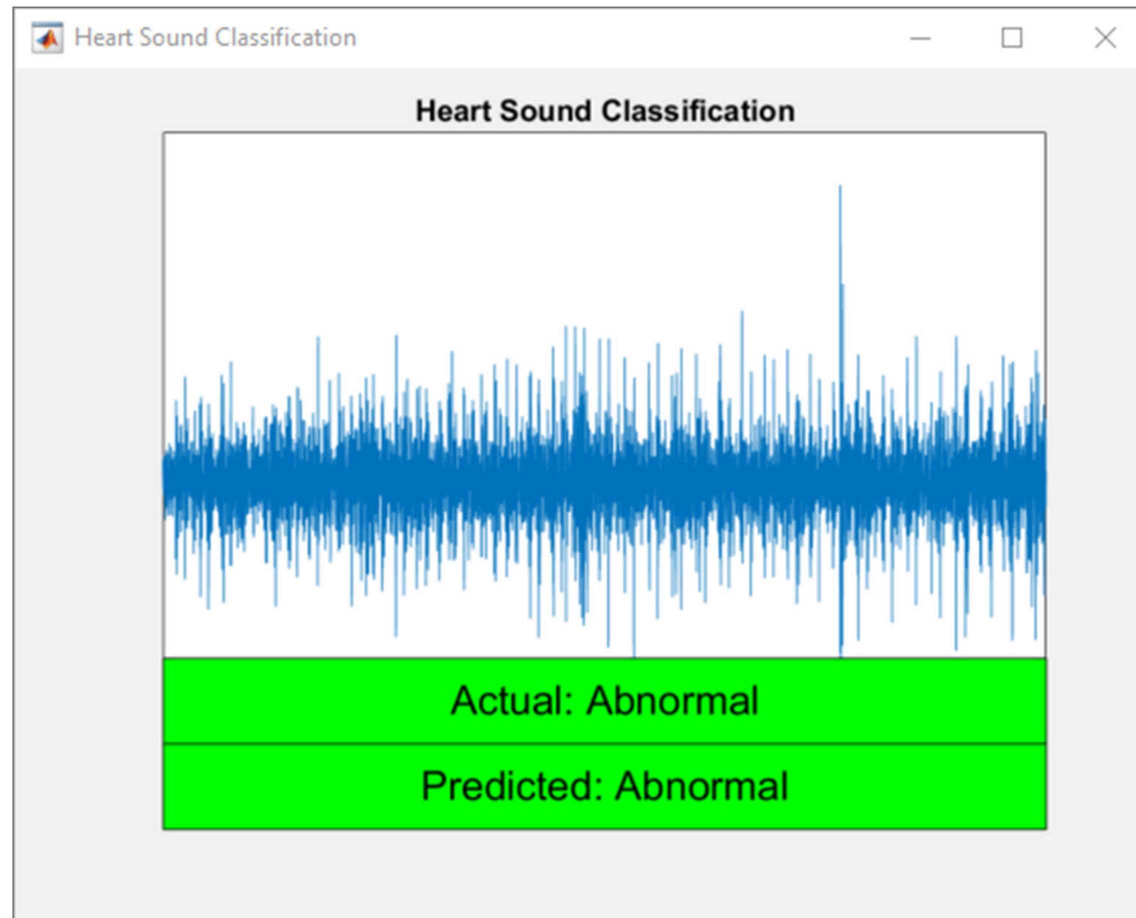
Request a *MATLAB Coder trial.*

Hands-On Exercise

Run the "Validate final model" section in the example script. The application displays the predicted class alongside the true class for several hundred heart sounds in the "validation" dataset, which was downloaded at the very beginning, along with the training data.

MathWorks®

To validate the generated C or C++ code, we implement a simple prototype application that interactively classifies files from the validation dataset. The three-step process described here generates an executable version that can be called from within MATLAB. We will use this version in our prototype (which is written in MATLAB, not C).



Validating the classifier in MATLAB.

At this point, we are ready to implement the application on a hand-held device.

Learn More

- *Embedded Code Generation*
- *MATLAB to C Made Easy* (55:15)

MathWorks®

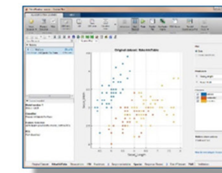
Essential Tools for Machine Learning

With MATLAB and specialized toolboxes, you can develop, tune, and deploy predictive analytics without extensive knowledge of machine learning or data science. MATLAB and related products offer all the tools you need to build and deploy your analytics:

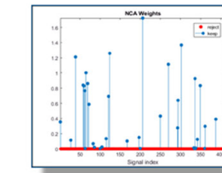
- **Data processing capabilities** that simplify time-consuming tasks, including handling missing data or outliers, removing noise, and time-aligning data with different sample rates
- **Specialized machine learning** apps to speed up your workflow, letting you quickly compare and select algorithms
- **Programmatic workflows** for removing unnecessary features and fine-tuning model parameters to achieve robust performance
- **Tools for scaling the machine learning workflow** to big data and compute clusters
- **Automatic code generation** tools for rapidly deploying your analytics to embedded targets

Prebuilt functions and algorithms support specialized applications, such as deep learning, computer vision, finance, image processing, text analytics, and autonomous agents.

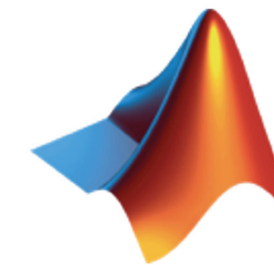
MATLAB for Machine Learning

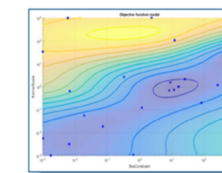


Use apps to quickly train and evaluate algorithms.

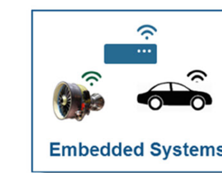


Use feature selection to reduce model size and prevent overfitting.





Use hyperparameter tuning and cost matrix to improve accuracy.



Use automatic code generation to rapidly deploy your analytics to embedded targets.

Access and Explore Data

Extensive data support:

- Work with signal, sound, images, finance data, text, geospatial, and other formats.
- Process and analyze signals.

Related products for accessing and exploring data:

- *Database Toolbox™*
- *Datafeed Toolbox™*
- *OPC Toolbox™*
- *Signal Processing Toolbox™*
- *Vehicle Network Toolbox™*

Preprocess Data and Extract Features

High-quality libraries and domain tools:

- Use industry-standard algorithms for feature extraction in finance, statistics, signal processing, image processing, text analytics, and computer vision.
- Improve your models with filtering, feature selection, and transformation.

Related products for specialized applications:

- *Computer Vision System Toolbox™*
- *Fuzzy Logic Toolbox™*
- *Image Processing Toolbox™*
- *Optimization Toolbox™*
- *Signal Processing Toolbox™*
- *Statistics and Machine Learning Toolbox™*
- *System Identification Toolbox™*
- *Text Analytics Toolbox™*
- *Wavelet Toolbox™*

MathWorks®

Develop and Optimize Predictive Models

Interactive, app-driven workflows:

- Quickly train and compare models.
- Focus on machine learning, not on programming.
- Select the best model and fine-tune model parameters.
- Scale computation to multiple cores or clusters.

Specialized apps and products for refining and tuning your model:

- *Classification Learner App*
- *Deep Learning Toolbox™*
- *Parallel Computing Toolbox™*
- *Regression Learner App*
- *Statistics and Machine Learning Toolbox™*

Deploy Analytics to a Production System

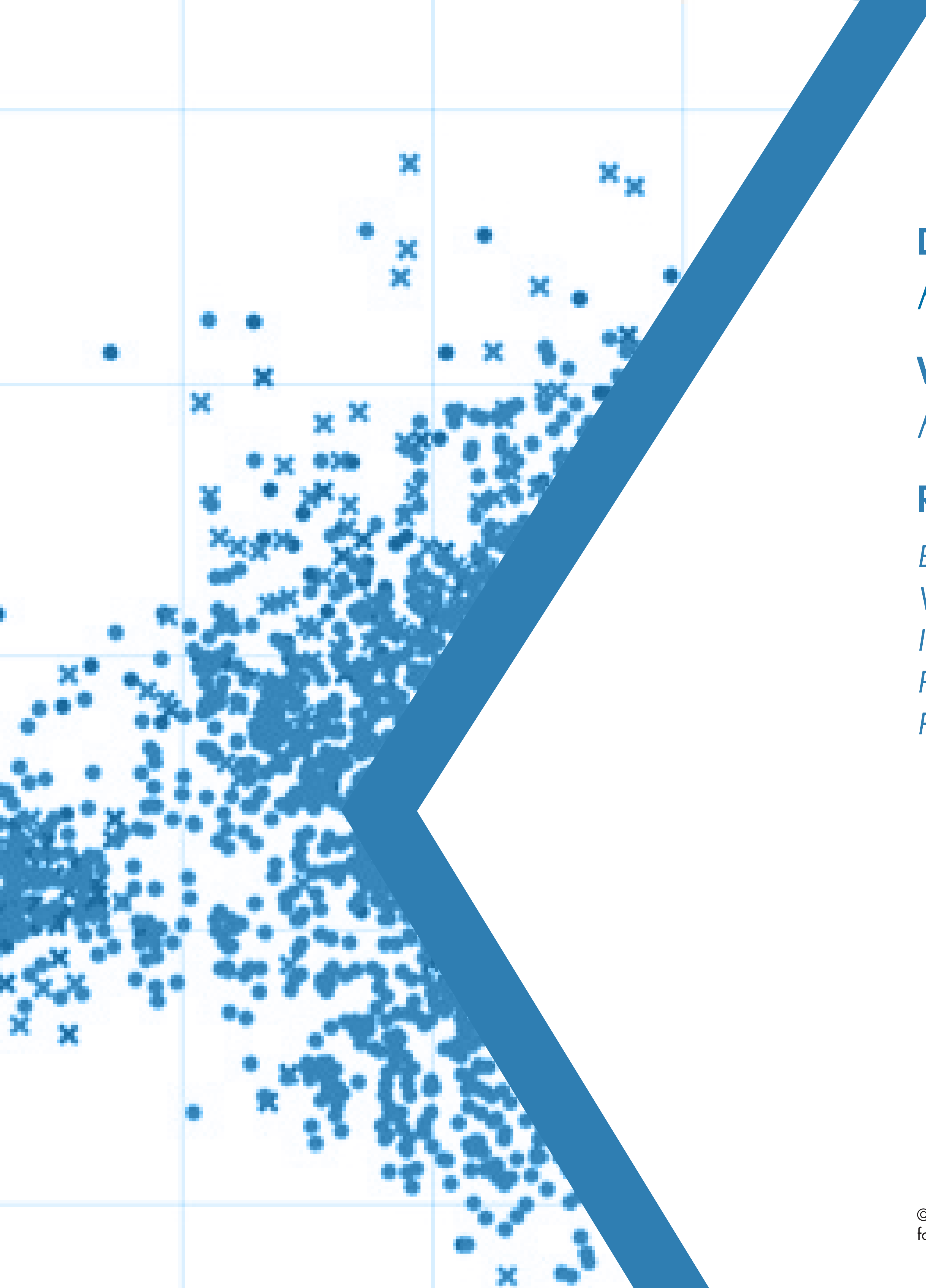
High-quality libraries and domain tools:

- Get tools to translate analytics to production.
- Generate code to deploy to embedded targets.
- Deploy to a broad range of target platforms and enterprise systems.

Related products for specialized applications:

- *HDL Coder™*
- *MATLAB Coder™*
- *MATLAB Compiler™*
- *MATLAB Compiler SDK™*
- *MATLAB Production Server™*

MathWorks®



Download

MATLAB Code for the Heart Sounds Classification Application

Watch

Machine Learning Using Heart Sound Classification Example (22:03)

Read

Big Data with MATLAB
What Is Deep Learning?
Introducing Deep Learning with MATLAB
Parallel Computing on the Cloud with MATLAB
Predictive Analytics

© 2019 The MathWorks, Inc. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See mathworks.com/trademarks for a list of additional trademarks.Other product or brand names may be trademarks or registered trademarks of their respective holders.