fitcsvm

Train support vector machine (SVM) classifier for one-class and binary classification

Syntax

```
Mdl = fitcsvm(Tbl,ResponseVarName)
Mdl = fitcsvm(Tbl,formula)
Mdl = fitcsvm(Tbl,Y)

Mdl = fitcsvm(X,Y)

Mdl = fitcsvm( __ ,Name,Value)
[Mdl,AggregateOptimizationResults] = fitcsvm( __ )
```

Description

`fitcsvm` trains or cross-validates a support vector machine (SVM) model for one-class and two-class (binary) classification on a low-dimensional or moderate-dimensional predictor data set. `fitcsvm` supports mapping the predictor data using kernel functions, and supports sequential minimal optimization (SMO), iterative single data algorithm (ISDA), or $L1$ soft-margin minimization via quadratic programming for objective-function minimization.

To train a linear SVM model for binary classification on a high-dimensional data set, that is, a data set that includes many predictor variables, use `fitclinear` instead.

For multiclass learning with combined binary SVM models, use error-correcting output codes (ECOC). For more details, see `fitcecoc`.

To train an SVM regression model, see `fitrsvm` for low-dimensional and moderate-dimensional predictor data sets, or `fitrlinear` for high-dimensional data sets.

`Mdl = fitcsvm(Tbl,ResponseVarName)` returns a support vector machine (SVM) classifier `Mdl` trained using the sample data contained in the table `Tbl`. `ResponseVarName` is the name of the variable in `Tbl` that contains the class labels for one-class or two-class classification.

If the class label variable contains only one class (for example, a vector of ones), `fitcsvm` trains a model for one-class classification. Otherwise, the function trains a model for two-class classification.

`Mdl = fitcsvm(Tbl,formula)` returns an SVM classifier trained using the sample data contained in the table `Tbl`. `formula` is an explanatory model of the response and a subset of the predictor variables in `Tbl` used to fit `Mdl`.

`Mdl = fitcsvm(Tbl,Y)` returns an SVM classifier trained using the predictor variables in the table `Tbl` and the class labels in vector `Y`.

`Mdl = fitcsvm(X,Y)` returns an SVM classifier trained using the predictors in the matrix `X` and the class labels in vector `Y` for one-class or two-class classification. example

`Mdl = fitcsvm( __ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in previous syntaxes. For example, you can specify the type of cross-validation, the cost for misclassification, and the type of score transformation function. example

`[Mdl,AggregateOptimizationResults] = fitcsvm( __ )` also returns `AggregateOptimizationResults`, which contains hyperparameter optimization results when you specify the `OptimizeHyperparameters` and `HyperparameterOptimizationOptions` name-value arguments. You must also specify the `ConstraintType` and `ConstraintBounds` options of `HyperparameterOptimizationOptions`. You can use this syntax to optimize on

compact model size instead of cross-validation loss, and to perform a set of multiple optimization problems that have the same options but different constraint bounds.

Examples

collapse all

Train SVM Classifier

Load Fisher's iris data set. Remove the sepal lengths and widths and all observed setosa irises.

Open in MATLAB Online

Copy Command

```
load fisheriris
inds = ~strcmp(species,'setosa');
X = meas(inds,3:4);
y = species(inds);
```

Get

Train an SVM classifier using the processed data set.

```
SVMModel = fitcsvm(X,y)
```

Get

```
SVMModel =
  ClassificationSVM
             ResponseName: 'Y'
    CategoricalPredictors: []
               ClassNames: {'versicolor'  'virginica'}
           ScoreTransform: 'none'
          NumObservations: 100
                    Alpha: [24×1 double]
                     Bias: -14.4149
         KernelParameters: [1×1 struct]
           BoxConstraints: [100×1 double]
          ConvergenceInfo: [1×1 struct]
          IsSupportVector: [100×1 logical]
                   Solver: 'SMO'


  Properties, Methods
```

`SVMModel` is a trained `ClassificationSVM` classifier. Display the properties of `SVMModel`. For example, to determine the class order, use dot notation.

```
classOrder = SVMModel.ClassNames
```

Get

```
classOrder = 2×1 cell
    {'versicolor'}
    {'virginica' }
```

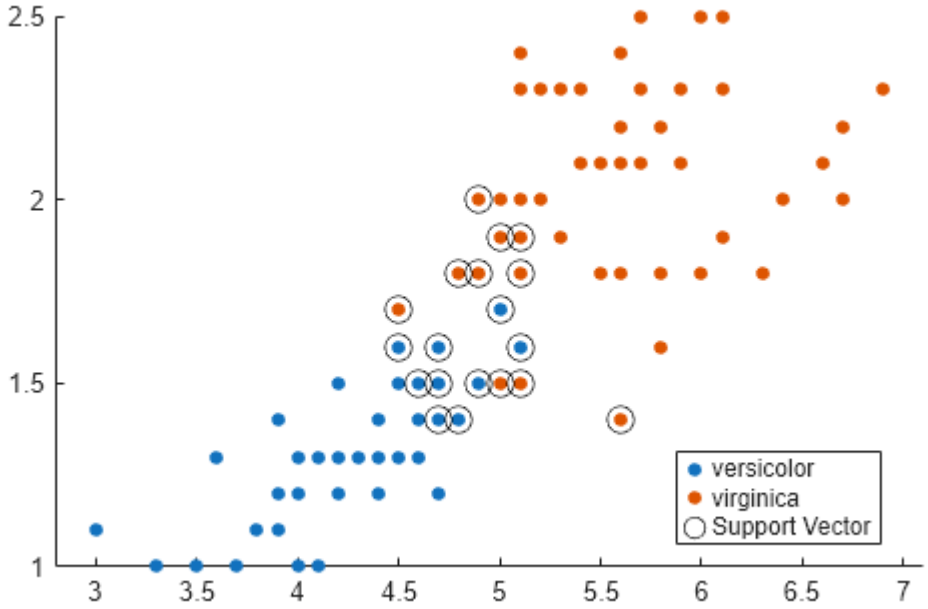
The first class (`'versicolor'`) is the negative class, and the second (`'virginica'`) is the positive class. You can change the class order during training by using the `'ClassNames'` name-value pair argument.

Plot a scatter diagram of the data and circle the support vectors.

```
sv = SVMModel.SupportVectors;
figure
gscatter(X(:,1),X(:,2),y)
```

Get

```
hold on
plot(sv(:,1),sv(:,2),'ko','MarkerSize',10)
legend('versicolor','virginica','Support Vector')
hold off
```



The support vectors are observations that occur on or beyond their estimated class boundaries.

You can adjust the boundaries (and, therefore, the number of support vectors) by setting a box constraint during training using the `'BoxConstraint'` name-value pair argument.

Plot Decision Boundary and Margin Lines for Two-Class SVM Classifier

This example shows how to plot the decision boundary and margin lines of a two-class (binary) SVM classifier with two predictor variables.

Load Fisher's iris data set. Exclude all the versicolor iris species (leaving only the setosa and virginica species), and keep only the sepal length and width measurements.

Open in MATLAB Online

Copy Command

```
load fisheriris;
inds = ~strcmp(species,'versicolor');
X = meas(inds,1:2);
s = species(inds);
```

Get

Train a linear kernel SVM classifier.

```
SVMModel = fitcsvm(X,s);
```

Get

`SVMModel` is a trained `ClassificationSVM` classifier, whose properties include the support vectors, linear predictor coefficients, and bias term.

```
sv = SVMModel.SupportVectors; % Support vectors
beta = SVMModel.Beta; % Linear predictor coefficients
b = SVMModel.Bias; % Bias term
```

Get

Plot a scatter diagram of the data, and circle the support vectors. The support vectors are observations that occur on or beyond their estimated class boundaries.

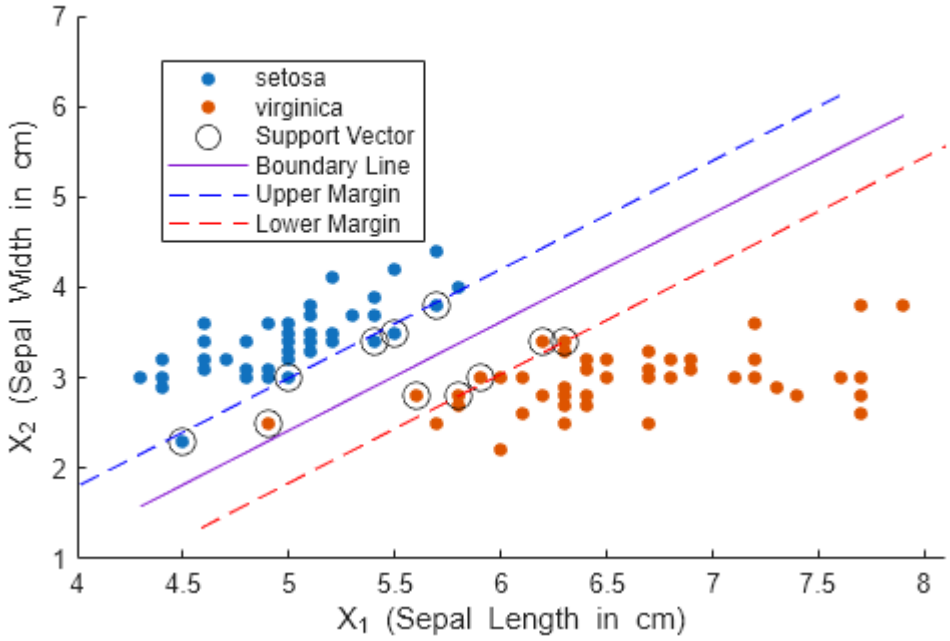
```
hold on
gscatter(X(:,1),X(:,2),s)
plot(sv(:,1),sv(:,2),'ko','MarkerSize',10)
```

The best separating hyperplane for the `SVMModel` classifier is a straight line specified by $\beta_1 X_1 + \beta_2 X_2 + b = 0$. Plot the decision boundary between the two species as a solid line.

```
X1 = linspace(min(X(:,1)),max(X(:,1)),100);
X2 = -(beta(1)/beta(2)*X1)-b/beta(2);
plot(X1,X2,'-')
```

The linear predictor coefficients β define a vector that is orthogonal to the decision boundary. The maximum margin width is $2||\beta||^{-1}$ (for more information, see Support Vector Machines for Binary Classification). Plot the maximum margin boundaries as dashed lines. Label the axes and add a legend.

```
m = 1/sqrt(beta(1)^2 + beta(2)^2);  % Margin half-width
X1margin_low = X1+beta(1)*m^2;
X2margin_low = X2+beta(2)*m^2;
X1margin_high = X1-beta(1)*m^2;
X2margin_high = X2-beta(2)*m^2;
plot(X1margin_high,X2margin_high,'b--')
plot(X1margin_low,X2margin_low,'r--')
xlabel('X_1 (Sepal Length in cm)')
ylabel('X_2 (Sepal Width in cm)')
legend('setosa','virginica','Support Vector', ...
    'Boundary Line','Upper Margin','Lower Margin')
hold off
```



Train and Cross-Validate SVM Classifier

Load the `ionosphere` data set.

Open in MATLAB Online

Copy Command

```
load ionosphere
rng(1); % For reproducibility
```

Train an SVM classifier using the radial basis kernel. Let the software find a scale value for the kernel function. Standardize the predictors.

```
SVMModel = fitcsvm(X,Y,'Standardize',true,'KernelFunction','RBF',...
    'KernelScale','auto');
```

`SVMModel` is a trained `ClassificationSVM` classifier.

Cross-validate the SVM classifier. By default, the software uses 10-fold cross-validation.

```
CVSVMModel = crossval(SVMModel);
```

`CVSVMModel` is a `ClassificationPartitionedModel` cross-validated classifier.

Estimate the out-of-sample misclassification rate.

```
classLoss = kfoldLoss(CVSVMModel)
```

```
classLoss =
0.0484
```

The generalization rate is approximately 5%.

Detect Outliers Using SVM and One-Class Learning

Modify Fisher's iris data set by assigning all the irises to the same class. Detect outliers in the modified data set, and confirm the expected proportion of the observations that are outliers.

Open in MATLAB Online

Copy Command

Load Fisher's iris data set. Remove the petal lengths and widths. Treat all irises as coming from the same class.

```
load fisheriris
X = meas(:,1:2);
y = ones(size(X,1),1);
```

Train an SVM classifier using the modified data set. Assume that 5% of the observations are outliers. Standardize the predictors.

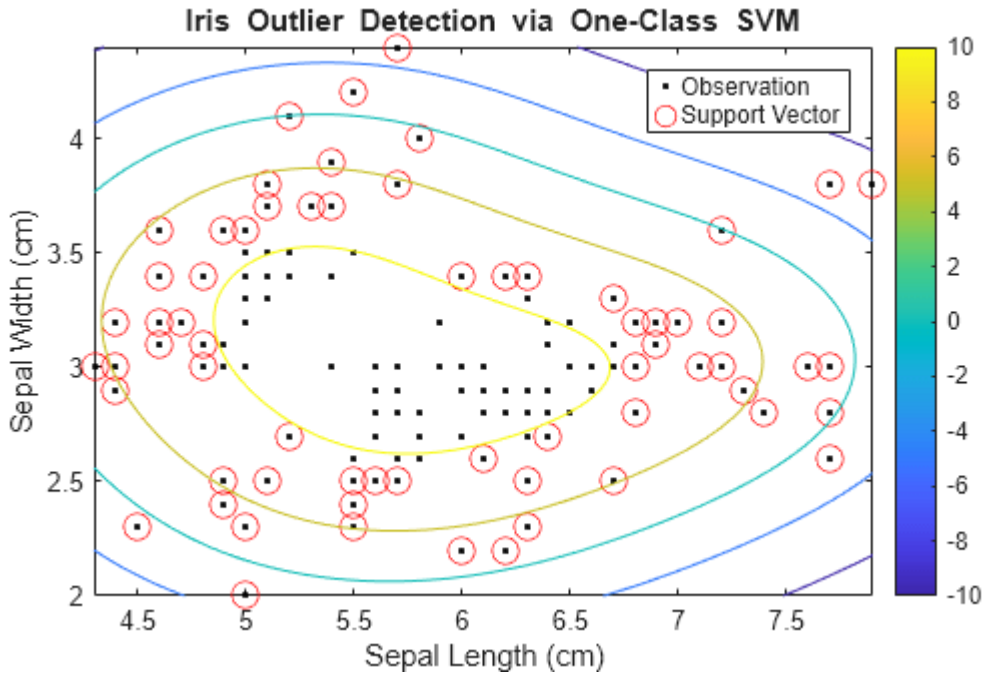
```
rng(1);
SVMModel = fitcsvm(X,y,'KernelScale','auto','Standardize',true,...
    'OutlierFraction',0.05);
```

`SVMModel` is a trained `ClassificationSVM` classifier. By default, the software uses the Gaussian kernel for one-class learning.

Plot the observations and the decision boundary. Flag the support vectors and potential outliers.

```
svInd = SVMModel.IsSupportVector;
h = 0.02; % Mesh grid step size
[X1,X2] = meshgrid(min(X(:,1)):h:max(X(:,1)),...
    min(X(:,2)):h:max(X(:,2)));
[~,score] = predict(SVMModel,[X1(:),X2(:)]);
scoreGrid = reshape(score,size(X1,1),size(X2,2));

figure
```

```
plot(X(:,1),X(:,2),'k.')
hold on
plot(X(svInd,1),X(svInd,2),'ro','MarkerSize',10)
contour(X1,X2,scoreGrid)
colorbar;
title('{\bf Iris Outlier Detection via One-Class SVM}')
xlabel('Sepal Length (cm)')
ylabel('Sepal Width (cm)')
legend('Observation','Support Vector')
hold off
```



The boundary separating the outliers from the rest of the data occurs where the contour value is `0`.

Verify that the fraction of observations with negative scores in the cross-validated data is close to 5%.

```
CVSVMModel = crossval(SVMModel);
[~,scorePred] = kfoldPredict(CVSVMModel);
outlierRate = mean(scorePred<0)
```

Get

```
outlierRate =
0.0467
```

Find Multiple Class Boundaries Using Binary SVM

Create a scatter plot of the `fisheriris` data set. Treat coordinates of a grid within the plot as new observations from the distribution of the data set, and find class boundaries by assigning the coordinates to one of the three classes in the data set.

Load Fisher's iris data set. Use the petal lengths and widths as the predictors.

Open in MATLAB Online

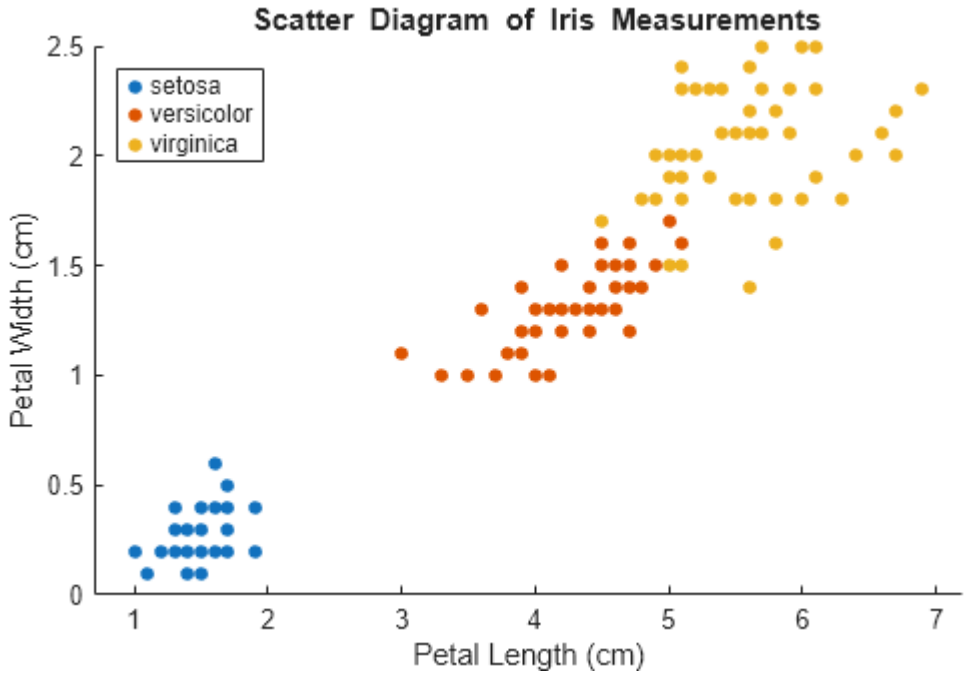
Copy Command

```
load fisheriris
X = meas(:,3:4);
Y = species;
```

Get

Examine a scatter plot of the data.

```
figure
gscatter(X(:,1),X(:,2),Y);
h = gca;
lims = [h.XLim h.YLim]; % Extract the x and y axis limits
title('{\bf Scatter Diagram of Iris Measurements}');
xlabel('Petal Length (cm)');
ylabel('Petal Width (cm)');
legend('Location','Northwest');
```



The data contains three classes, one of which is linearly separable from the others.

For each class:

1. Create a logical vector (`indx`) indicating whether an observation is a member of the class.
2. Train an SVM classifier using the predictor data and `indx`.
3. Store the classifier in a cell of a cell array.

Define the class order.

```
SVMModels = cell(3,1);
classes = unique(Y);
rng(1); % For reproducibility

for j = 1:numel(classes)
    indx = strcmp(Y,classes(j)); % Create binary classes for each classifier
    SVMModels{j} = fitcsvm(X,indx,'ClassNames',[false true],'Standardize',true,
        'KernelFunction','rbf','BoxConstraint',1);
end
```

`SVMModels` is a 3-by-1 cell array, with each cell containing a `ClassificationSVM` classifier. For each cell, the positive class is setosa, versicolor, and virginica, respectively.

Define a fine grid within the plot, and treat the coordinates as new observations from the distribution of the training data. Estimate the score of the new observations using each classifier.

```
d = 0.02;
[x1Grid,x2Grid] = meshgrid(min(X(:,1)):d:max(X(:,1)),...
    min(X(:,2)):d:max(X(:,2)));
```

```
xGrid = [x1Grid(:),x2Grid(:)];
N = size(xGrid,1);
Scores = zeros(N,numel(classes));

for j = 1:numel(classes)
    [~,score] = predict(SVMModels{j},xGrid);
    Scores(:,j) = score(:,2); % Second column contains positive-class scores
end
```

Each row of `Scores` contains three scores. The index of the element with the largest score is the index of the class to which the new class observation most likely belongs.

Associate each new observation with the classifier that gives it the maximum score.

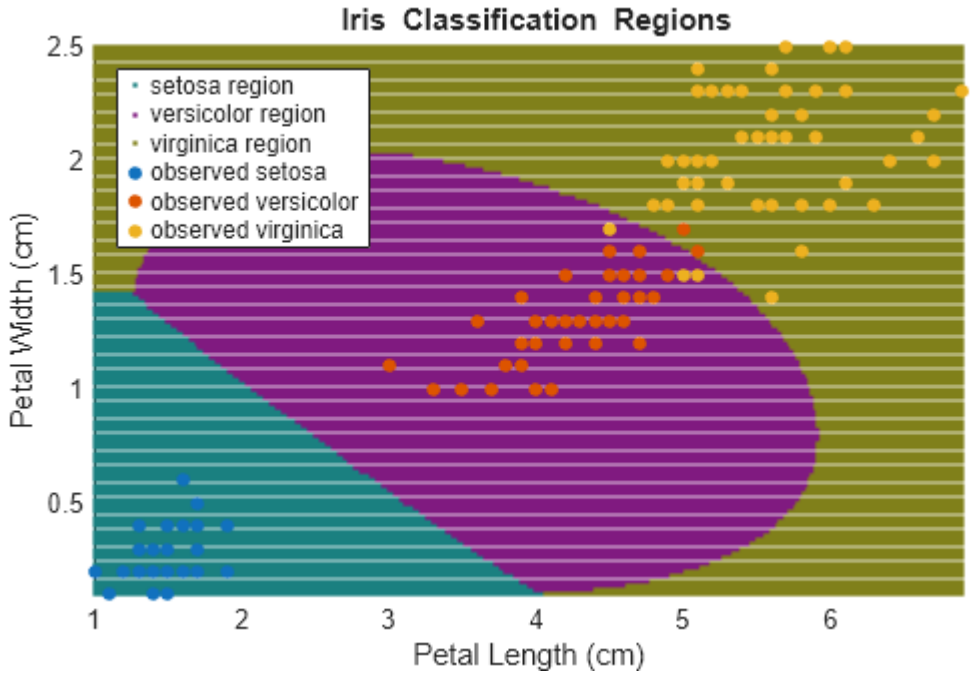
```
[~,maxScore] = max(Scores,[],2);
```

Get

Color in the regions of the plot based on the class to which the corresponding new observation belongs.

```
figure
h(1:3) = gscatter(xGrid(:,1),xGrid(:,2),maxScore,...
    [0.1 0.5 0.5; 0.5 0.1 0.5; 0.5 0.5 0.1]);
hold on
h(4:6) = gscatter(X(:,1),X(:,2),Y);
title('{\bf Iris Classification Regions}');
xlabel('Petal Length (cm)');
ylabel('Petal Width (cm)');
legend(h,{'setosa region','versicolor region','virginica region',...
    'observed setosa','observed versicolor','observed virginica'},...
    'Location','Northwest');
axis tight
hold off
```

Get



Optimize SVM Classifier

Optimize hyperparameters automatically using `fitcsvm`.

Load the `ionosphere` data set.

Open in MATLAB Online

Copy Command

```
load ionosphere
```

Get

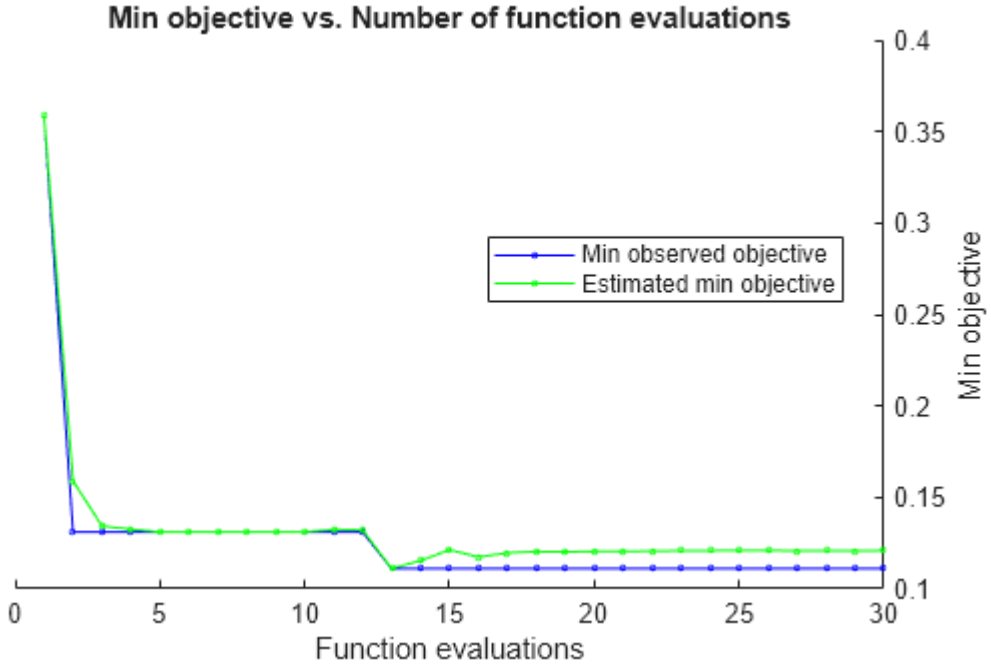
Find hyperparameters that minimize five-fold cross-validation loss by using automatic hyperparameter optimization. For reproducibility, set the random seed and use the `'expected-improvement-plus'` acquisition function.

```
rng default
Mdl = fitcsvm(X,Y,'OptimizeHyperparameters','auto', ...
    'HyperparameterOptimizationOptions',struct('AcquisitionFunctionName', ...
    'expected-improvement-plus'))
```

Get

```
|=====================================================================================================|
| Iter | Eval   | Objective   | Objective   | BestSoFar   | BestSoFar   | BoxConstraint|  Kerr
|      | result |             | runtime     | (observed)  | (estim.)    |              |
|=====================================================================================================|
|    1 | Best   |     0.35897 |     0.21586 |     0.35897 |     0.35897 |       3.8653 |
|    2 | Best   |     0.12821 |      8.5911 |     0.12821 |     0.15646 |       429.99 |
|    3 | Accept |     0.35897 |    0.071978 |     0.12821 |      0.1315 |      0.11801 |
|    4 | Accept |      0.1339 |      4.2217 |     0.12821 |     0.12965 |    0.0010694 |    0.
|    5 | Accept |     0.15954 |      10.111 |     0.12821 |     0.12824 |       973.65 |
|    6 | Accept |     0.13675 |     0.16644 |     0.12821 |      0.1283 |       234.28 |
|    7 | Accept |     0.35897 |     0.04513 |     0.12821 |     0.12826 |     0.005253 |
|    8 | Accept |     0.14245 |      10.194 |     0.12821 |     0.12829 |       91.176 |     0
|    9 | Accept |       0.151 |      9.4829 |     0.12821 |     0.12831 |    0.0064316 |    0.
|   10 | Accept |      0.1339 |      5.0189 |     0.12821 |     0.12835 |       153.27 |
|   11 | Accept |     0.26781 |      10.317 |     0.12821 |     0.12948 |       260.21 |     0
|   12 | Accept |     0.12821 |     0.21803 |     0.12821 |     0.12935 |    0.0034086 |     0
|   13 | Best   |     0.11966 |    0.072008 |     0.11966 |     0.12144 |    0.0010229 |     0
|   14 | Accept |     0.35897 |    0.052694 |     0.11966 |     0.12177 |       987.92 |
|   15 | Accept |     0.12821 |    0.053089 |     0.11966 |     0.11979 |    0.0010124 |
|   16 | Accept |     0.22792 |      10.006 |     0.11966 |     0.11995 |       6.9606 |     0
|   17 | Accept |     0.11966 |    0.064647 |     0.11966 |     0.11979 |    0.0010509 |     0
|   18 | Accept |     0.12251 |    0.054625 |     0.11966 |     0.12049 |    0.0010131 |     0
|   19 | Accept |      0.1339 |      4.8934 |     0.11966 |     0.12071 |      0.08684 |     0
|   20 | Accept |     0.11966 |     0.18885 |     0.11966 |     0.12123 |     0.048335 |
|=====================================================================================================|
| Iter | Eval   | Objective   | Objective   | BestSoFar   | BestSoFar   | BoxConstraint|  Kerr
|      | result |             | runtime     | (observed)  | (estim.)    |              |
|=====================================================================================================|
|   21 | Accept |     0.11966 |     0.08286 |     0.11966 |     0.12119 |     0.013573 |
|   22 | Accept |      0.1339 |     0.55852 |     0.11966 |     0.12123 |       1.2995 |
|   23 | Accept |      0.1396 |      7.9392 |     0.11966 |     0.12127 |    0.0019603 |    0.
|   24 | Accept |     0.13675 |      6.5142 |     0.11966 |     0.12136 |      0.17347 |     0
|   25 | Accept |     0.14245 |      0.1391 |     0.11966 |     0.11914 |    0.0010001 |    0.
|   26 | Accept |     0.12536 |     0.13991 |     0.11966 |     0.12144 |     0.035903 |
|   27 | Accept |      0.1339 |      3.3223 |     0.11966 |     0.12144 |       897.97 |
|   28 | Accept |      0.1339 |      5.0618 |     0.11966 |     0.12146 |       6.6245 |     0
|   29 | Accept |     0.12251 |    0.090061 |     0.11966 |      0.1214 |    0.0069215 |
|   30 | Accept |     0.12251 |     0.11955 |     0.11966 |     0.12136 |     0.049433 |


__________________________________________________________
Optimization completed.
MaxObjectiveEvaluations of 30 reached.
Total function evaluations: 30
```

```
Total elapsed time: 107.0032 seconds
Total objective function evaluation time: 98.0074

Best observed feasible point:
    BoxConstraint    KernelScale    Standardize
    _____________    ___________    ___________

      0.0010229       0.032368         true

Observed objective function value = 0.11966
Estimated objective function value = 0.12229
Function evaluation time = 0.072008

Best estimated feasible point (according to models):
    BoxConstraint    KernelScale    Standardize
    _____________    ___________    ___________

      0.013573         0.11789         true

Estimated objective function value = 0.12136
Estimated function evaluation time = 0.098473
```




```
Mdl =
  ClassificationSVM
                         ResponseName: 'Y'
                CategoricalPredictors: []
                           ClassNames: {'b'  'g'}
                      ScoreTransform: 'none'
                     NumObservations: 351
    HyperparameterOptimizationResults: [1×1 BayesianOptimization]
                                Alpha: [90×1 double]
                                 Bias: -0.1318
                    KernelParameters: [1×1 struct]
                                   Mu: [0.8917 0 0.6413 0.0444 0.6011 0.1159 0.5501 0.1194 0.5
                                Sigma: [0.3112 0 0.4977 0.4414 0.5199 0.4608 0.4927 0.5207 0.5
                      BoxConstraints: [351×1 double]
                     ConvergenceInfo: [1×1 struct]
                     IsSupportVector: [351×1 logical]
```

```
                    Solver: 'SMO'


  Properties, Methods
```

Input Arguments

collapse all

`Tbl` — Sample data

table

Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. Multicolumn variables and cell arrays other than cell arrays of character vectors are not allowed.

Optionally, `Tbl` can contain a column for the response variable and a column for the observation weights.

- The response variable must be a categorical, character, or string array, a logical or numeric vector, or a cell array of character vectors.
  - `fitcsvm` supports only one-class and two-class (binary) classification. Either the response variable must contain at most two distinct classes, or you must specify one or two classes for training by using the `ClassNames` name-value argument. For multiclass learning, see `fitcecoc`.
  - A good practice is to specify the order of the classes in the response variable by using the `ClassNames` name-value argument.
- The column for the weights must be a numeric vector.
- You must specify the response variable in `Tbl` by using `ResponseVarName` or `formula` and specify the observation weights in `Tbl` by using `Weights`.
  - Specify the response variable by using `ResponseVarName` — `fitcsvm` uses the remaining variables as predictors. To use a subset of the remaining variables in `Tbl` as predictors, specify predictor variables by using `PredictorNames`.
  - Define a model specification by using `formula` — `fitcsvm` uses a subset of the variables in `Tbl` as predictor variables and the response variable, as specified in `formula`.

If `Tbl` does not contain the response variable, then specify a response variable by using `Y`. The length of the response variable `Y` and the number of rows in `Tbl` must be equal. To use a subset of the variables in `Tbl` as predictors, specify predictor variables by using `PredictorNames`.

Data Types: `table`

`ResponseVarName` — Response variable name

name of variable in `Tbl`

Response variable name, specified as the name of a variable in `Tbl`.

You must specify `ResponseVarName` as a character vector or string scalar. For example, if the response variable `Y` is stored as `Tbl.Y`, then specify it as `"Y"`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

The response variable must be a categorical, character, or string array; a logical or numeric vector; or a cell array of character vectors. If `Y` is a character array, then each element of the response variable must correspond to one row of the array.

A good practice is to specify the order of the classes by using the `ClassNames` name-value argument.

Data Types: `char` | `string`

`formula` — Explanatory model of response variable and subset of predictor variables
character vector | string scalar

Explanatory model of the response variable and a subset of the predictor variables, specified as a character vector or string scalar in the form `"Y~x1+x2+x3"`. In this form, `Y` represents the response variable, and `x1`, `x2`, and `x3` represent the predictor variables.

To specify a subset of variables in `Tbl` as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in `Tbl` that do not appear in `formula`.

The variable names in the formula must be both variable names in `Tbl` (`Tbl.Properties.VariableNames`) and valid MATLAB® identifiers. You can verify the variable names in `Tbl` by using the `isvarname` function. If the variable names are not valid, then you can convert them by using the `matlab.lang.makeValidName` function.

Data Types: `char` | `string`

`Y` — Class labels
categorical array | character array | string array | logical vector | numeric vector | cell array of character vectors

Class labels to which the SVM model is trained, specified as a categorical, character, or string array, logical or numeric vector, or cell array of character vectors.

- `fitcsvm` supports only one-class and two-class (binary) classification. Either `Y` must contain at most two distinct classes, or you must specify one or two classes for training by using the `ClassNames` name-value argument. For multiclass learning, see `fitcecoc`.
- The length of `Y` and the number of rows in `Tbl` or `X` must be equal.
- If `Y` is a character array, then each label must correspond to one row of the array.
- It is a good practice to specify the class order by using the `ClassNames` name-value pair argument.

Data Types: `categorical` | `char` | `string` | `logical` | `single` | `double` | `cell`

`X` — Predictor data
matrix of numeric values

Predictor data to which the SVM classifier is trained, specified as a matrix of numeric values.

Each row of `X` corresponds to one observation (also known as an instance or example), and each column corresponds to one predictor (also known as a feature).

The length of `Y` and the number of rows in `X` must be equal.

To specify the names of the predictors in the order of their appearance in `X`, use the `'PredictorNames'` name-value pair argument.

Data Types: `double` | `single`

Name-Value Arguments

collapse all

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the

pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `fitcsvm(X,Y,'KFold',10,'Cost',[0 2;1 0],'ScoreTransform','sign')` performs 10-fold cross-validation, applies double the penalty to false positives compared to false negatives, and transforms scores using the sign function.

SVM Options

collapse all

`BoxConstraint` — Box constraint
1 (default) | positive scalar

Box constraint, specified as the comma-separated pair consisting of `'BoxConstraint'` and a positive scalar.

For one-class learning, the software always sets the box constraint to `1`.

For more details on the relationships and algorithmic behavior of `BoxConstraint`, `Cost`, `Prior`, `Standardize`, and `Weights`, see Algorithms.

Example: `'BoxConstraint',100`

Data Types: `double` | `single`

`KernelFunction` — Kernel function
`'linear'` | `'gaussian'` | `'rbf'` | `'polynomial'` | function name

Kernel function used to compute the elements of the Gram matrix, specified as the comma-separated pair consisting of `'KernelFunction'` and a kernel function name. Suppose $G(x_j,x_k)$ is element (j,k) of the Gram matrix, where x_j and x_k are p-dimensional vectors representing observations j and k in `X`. This table describes supported kernel function names and their functional forms.

Kernel Function Name	Description	Formula
`'gaussian'` or `'rbf'`	Gaussian or Radial Basis Function (RBF) kernel, default for one-class learning	$G(x_j,x_k)=\exp\left(-\|x_j-x_k\|^2\right)$
`'linear'`	Linear kernel, default for two-class learning	$G(x_j,x_k)=x_j'x_k$
`'polynomial'`	Polynomial kernel. Use `'PolynomialOrder',q` to specify a polynomial kernel of order q.	$G(x_j,x_k)=(1+x_j'x_k)^q$

You can set your own kernel function, for example, `kernel`, by setting `'KernelFunction','kernel'`. The value `kernel` must have this form.

```
function G = kernel(U,V)
```

where:

- `U` is an m-by-p matrix. Columns correspond to predictor variables, and rows correspond to observations.
- `V` is an n-by-p matrix. Columns correspond to predictor variables, and rows correspond to observations.
- `G` is an m-by-n Gram matrix of the rows of `U` and `V`.

`kernel.m` must be on the MATLAB path.

It is a good practice to avoid using generic names for kernel functions. For example, call a sigmoid kernel function `'mysigmoid'` rather than `'sigmoid'`.

Example: `'KernelFunction','gaussian'`
Data Types: `char` | `string`

`KernelScale` — Kernel scale parameter

`1` (default) | `'auto'` | positive scalar

Kernel scale parameter, specified as the comma-separated pair consisting of `'KernelScale'` and `'auto'` or a positive scalar. The software divides all elements of the predictor matrix `X` by the value of `KernelScale`. Then, the software applies the appropriate kernel norm to compute the Gram matrix.

- If you specify `'auto'`, then the software selects an appropriate scale factor using a heuristic procedure. This heuristic procedure uses subsampling, so estimates can vary from one call to another. Therefore, to reproduce results, set a random number seed using `rng` before training.
- If you specify `KernelScale` and your own kernel function, for example, `'KernelFunction','kernel'`, then the software throws an error. You must apply scaling within `kernel`.

Example: `'KernelScale','auto'`

Data Types: `double` | `single` | `char` | `string`

`PolynomialOrder` — Polynomial kernel function order

`3` (default) | positive integer

Polynomial kernel function order, specified as the comma-separated pair consisting of `'PolynomialOrder'` and a positive integer.

If you set `'PolynomialOrder'` and `KernelFunction` is not `'polynomial'`, then the software throws an error.

Example: `'PolynomialOrder',2`

Data Types: `double` | `single`

`KernelOffset` — Kernel offset parameter

nonnegative scalar

Kernel offset parameter, specified as the comma-separated pair consisting of `'KernelOffset'` and a nonnegative scalar.

The software adds `KernelOffset` to each element of the Gram matrix.

The defaults are:

- `0` if the solver is SMO (that is, you set `'Solver','SMO'`)
- `0.1` if the solver is ISDA (that is, you set `'Solver','ISDA'`)

Example: `'KernelOffset',0`

Data Types: `double` | `single`

`Standardize` — Flag to standardize predictor data
`false` (default) | `true`

Flag to standardize the predictor data, specified as the comma-separated pair consisting of `'Standardize'` and `true` (`1`) or `false` (`0`).

If you set `'Standardize',true`:

- The software centers and scales each predictor variable (`X` or `Tbl`) by the corresponding weighted column mean and standard deviation. For details on weighted standardizing, see Algorithms. MATLAB does not standardize the data contained in the dummy variable columns generated for categorical predictors.
- The software trains the classifier using the standardized predictors, but stores the unstandardized predictors as a matrix or table in the classifier property `X`.

Example: `'Standardize',true`

Data Types: `logical`

`Solver` — Optimization routine
`'ISDA'` | `'L1QP'` | `'SMO'`

Optimization routine, specified as the comma-separated pair consisting of `'Solver'` and a value in this table.

Value	Description
`'ISDA'`	Iterative Single Data Algorithm (see [4])
`'L1QP'`	Uses `quadprog` (Optimization Toolbox) to implement $L1$ soft-margin minimization by quadratic programming. This option requires an Optimization Toolbox™ license. For more details, see Quadratic Programming Definition (Optimization Toolbox).
`'SMO'`	Sequential Minimal Optimization (see [2])

The default value is `'ISDA'` if you set `'OutlierFraction'` to a positive value for two-class learning, and `'SMO'` otherwise.

Example: `'Solver','ISDA'`

`Alpha` — Initial estimates of alpha coefficients
numeric vector of nonnegative values

Initial estimates of alpha coefficients, specified as the comma-separated pair consisting of `'Alpha'` and a numeric vector of nonnegative values. The length of `Alpha` must be equal to the number of rows in `X`.

- Each element of `'Alpha'` corresponds to an observation in `X`.
- `'Alpha'` cannot contain any `NaN`s.
- If you specify `'Alpha'` and any one of the cross-validation name-value pair arguments (`'CrossVal'`, `'CVPartition'`, `'Holdout'`, `'KFold'`, or `'Leaveout'`), then the software returns an error.

If `Y` contains any missing values, then remove all rows of `Y`, `X`, and `'Alpha'` that correspond to the missing values. That is, enter:

```
idx = ~isundefined(categorical(Y));
Y = Y(idx,:);
X = X(idx,:);
alpha = alpha(idx);
```

Then pass `Y`, `X`, and `alpha` as the response, predictors, and initial alpha estimates, respectively.

The default values are:

- `0.5*ones(size(X,1),1)` for one-class learning
- `zeros(size(X,1),1)` for two-class learning

Example: `'Alpha',0.1*ones(size(X,1),1)`
Data Types: `double` | `single`

`CacheSize` — Cache size

`1000` (default) | `'maximal'` | positive scalar

Cache size, specified as the comma-separated pair consisting of `'CacheSize'` and `'maximal'` or a positive scalar.

If `CacheSize` is `'maximal'`, then the software reserves enough memory to hold the entire n-by-n Gram matrix.

If `CacheSize` is a positive scalar, then the software reserves `CacheSize` megabytes of memory for training the model.

Example: `'CacheSize','maximal'`

Data Types: `double` | `single` | `char` | `string`

`ClipAlphas` — Flag to clip alpha coefficients

`true` (default) | `false`

Flag to clip alpha coefficients, specified as the comma-separated pair consisting of `'ClipAlphas'` and either `true` or `false`.

Suppose that the alpha coefficient for observation j is α_j and the box constraint of observation j is C_j, j = 1,...,n, where n is the training sample size.

Value	Description
`true`	At each iteration, if α_j is near 0 or near C_j, then MATLAB sets α_j to 0 or to C_j, respectively.
`false`	MATLAB does not change the alpha coefficients during optimization.

MATLAB stores the final values of α in the `Alpha` property of the trained SVM model object.

`ClipAlphas` can affect SMO and ISDA convergence.

Example: `'ClipAlphas',false`

Data Types: `logical`

`Nu` — ν parameter for one-class learning
`0.5` (default) | positive scalar

ν parameter for One-Class Learning, specified as the comma-separated pair consisting of `'Nu'` and a positive scalar. `Nu` must be greater than `0` and at most `1`.

Set `Nu` to control the tradeoff between ensuring that most training examples are in the positive class and minimizing the weights in the score function.

Example: `'Nu',0.25`

Data Types: `double` | `single`

`NumPrint` — Number of iterations between optimization diagnostic message output
`1000` (default) | nonnegative integer

Number of iterations between optimization diagnostic message output, specified as the comma-separated pair consisting of `'NumPrint'` and a nonnegative integer.

If you specify `'Verbose',1` and `'NumPrint',numprint`, then the software displays all optimization diagnostic messages from SMO and ISDA every `numprint` iterations in the Command Window.

Example: `'NumPrint',500`

Data Types: `double` | `single`

`OutlierFraction` — Expected proportion of outliers in training data
`0` (default) | numeric scalar in the interval [0,1)

Expected proportion of outliers in the training data, specified as the comma-separated pair consisting of `'OutlierFraction'` and a numeric scalar in the interval [0,1).

Suppose that you set `'OutlierFraction',outlierfraction`, where `outlierfraction` is a value greater than 0.

- For two-class learning, the software implements *robust learning*. In other words, the software attempts to remove 100*`outlierfraction`% of the observations when the optimization algorithm converges. The removed observations correspond to gradients that are large in magnitude.
- For one-class learning, the software finds an appropriate bias term such that `outlierfraction` of the observations in the training set have negative scores.

Example: `'OutlierFraction',0.01`

Data Types: `double` | `single`

`RemoveDuplicates` — Flag to replace duplicate observations with single observations
`false` (default) | `true`

Flag to replace duplicate observations with single observations in the training data, specified as the comma-separated pair consisting of `'RemoveDuplicates'` and `true` or `false`.

If `RemoveDuplicates` is `true`, then `fitcsvm` replaces duplicate observations in the training data with a single observation of the same value. The weight of the single observation is equal to the sum of the weights of the corresponding removed duplicates (see `Weights`).

Tip

If your data set contains many duplicate observations, then specifying `'RemoveDuplicates',true` can decrease convergence time considerably.

Data Types: `logical`

`Verbose` — Verbosity level

`0` (default) | `1` | `2`

Verbosity level, specified as the comma-separated pair consisting of `'Verbose'` and `0`, `1`, or `2`. The value of `Verbose` controls the amount of optimization information that the software displays in the Command Window and saves the information as a structure to `Mdl.ConvergenceInfo.History`.

This table summarizes the available verbosity level options.

Value	Description
`0`	The software does not display or save convergence information.
`1`	The software displays diagnostic messages and saves convergence criteria every `numprint` iterations, where `numprint` is the value of the name-value pair argument `'NumPrint'`.
`2`	The software displays diagnostic messages and saves convergence criteria at every iteration.

Example: `'Verbose',1`

Data Types: `double` | `single`

Other Classification Options

collapse all

`CategoricalPredictors` — Categorical predictors list

vector of positive integers | logical vector | character matrix | string array | cell array of character vectors | `'all'`

Categorical predictors list, specified as one of the values in this table.

Value	Description
Vector of positive integers	Each entry in the vector is an index value indicating that the corresponding predictor is categorical. The index values are between 1 and `p`, where `p` is the number of predictors used to train the model. If `fitcsvm` uses a subset of input variables as predictors, then the function indexes the predictors using only the subset. The `CategoricalPredictors` values do not count any response variable, observation weights variable, or other variable that the function does not use.
Logical vector	A `true` entry means that the corresponding predictor is categorical. The length of the vector is `p`.

Value	Description
Character matrix	Each row of the matrix is the name of a predictor variable. The names must match the entries in `PredictorNames`. Pad the names with extra blanks so each row of the character matrix has the same length.
String array or cell array of character vectors	Each element in the array is the name of a predictor variable. The names must match the entries in `PredictorNames`.
`"all"`	All predictors are categorical.

By default, if the predictor data is in a table (`Tbl`), `fitcsvm` assumes that a variable is categorical if it is a logical vector, categorical vector, character array, string array, or cell array of character vectors. If the predictor data is a matrix (`X`), `fitcsvm` assumes that all predictors are continuous. To identify any other predictors as categorical predictors, specify them by using the `CategoricalPredictors` name-value argument.

For the identified categorical predictors, `fitcsvm` creates dummy variables using two different schemes, depending on whether a categorical variable is unordered or ordered. For an unordered categorical variable, `fitcsvm` creates one dummy variable for each level of the categorical variable. For an ordered categorical variable, `fitcsvm` creates one less dummy variable than the number of categories. For details, see Automatic Creation of Dummy Variables.

Example: `'CategoricalPredictors','all'`
Data Types: `single` | `double` | `logical` | `char` | `string` | `cell`

`ClassNames` — Names of classes to use for two-class learning

categorical array | character array | string array | logical vector | numeric vector | cell array of character vectors

Names of classes to use for two-class learning, specified as a categorical, character, or string array; a logical or numeric vector; or a cell array of character vectors. `ClassNames` must have the same data type as the response variable in `Tbl` or `Y`.

If `ClassNames` is a character array, then each element must correspond to one row of the array.

Use `ClassNames` to:

- Specify the order of the classes during training.
- Specify the order of any input or output argument dimension that corresponds to the class order. For example, use `ClassNames` to specify the order of the dimensions of `Cost` or the column order of classification scores returned by `predict`.
- Select a subset of classes for training. For example, suppose that the set of all distinct class names in `Y` is `["a","b","c"]`. To train the model using observations from classes `"a"` and `"c"` only, specify `ClassNames=["a","c"]`.

The default value for `ClassNames` is the set of all distinct class names in the response variable in `Tbl` or `Y`.

This argument is valid only for two-class learning.

Example: `"ClassNames",["b","g"]`

Data Types: `categorical` | `char` | `string` | `logical` | `single` | `double` | `cell`

`Cost` — Misclassification cost for two-class learning

`[0 1; 1 0]` (default) | square matrix | structure array

Misclassification cost for two-class learning, specified as the comma-separated pair consisting of `'Cost'` and a square matrix or structure array.

- If you specify the square matrix `Cost` and the true class of an observation is `i`, then `Cost(i,j)` is the cost of classifying a point into class `j`. That is, rows correspond to the true classes and columns correspond to predicted classes. To specify the class order for the corresponding rows and columns of `Cost`, also specify the `ClassNames` name-value pair argument.
- If you specify the structure `S`, then it must have two fields:
  - `S.ClassNames`, which contains the class names as a variable of the same data type as `Y`
  - `S.ClassificationCosts`, which contains the cost matrix with rows and columns ordered as in `S.ClassNames`

If you specify a cost matrix, then the software updates the prior probabilities by incorporating the penalties described in the cost matrix for training, and stores the user-specified value in the `Cost` property of the trained SVM model object. For more details on the relationships and algorithmic behavior of `BoxConstraint`, `Cost`, `Prior`, `Standardize`, and `Weights`, see Algorithms.

This argument is valid only for two-class learning.

Example: `'Cost',[0,1;2,0]`

Data Types: `double` | `single` | `struct`

`PredictorNames` — Predictor variable names

string array of unique names | cell array of unique character vectors

Predictor variable names, specified as a string array of unique names or cell array of unique character vectors. The functionality of `PredictorNames` depends on the way you supply the training data.

- If you supply `X` and `Y`, then you can use `PredictorNames` to assign names to the predictor variables in `X`.
  - The order of the names in `PredictorNames` must correspond to the column order of `X`. That is, `PredictorNames{1}` is the name of `X(:,1)`, `PredictorNames{2}` is the name of `X(:,2)`, and so on. Also, `size(X,2)` and `numel(PredictorNames)` must be equal.
  - By default, `PredictorNames` is `{'x1','x2',...}`.
- If you supply `Tbl`, then you can use `PredictorNames` to choose which predictor variables to use in training. That is, `fitcsvm` uses only the predictor variables in `PredictorNames` and the response variable during training.
  - `PredictorNames` must be a subset of `Tbl.Properties.VariableNames` and cannot include the name of the response variable.
  - By default, `PredictorNames` contains the names of all predictor variables.
  - A good practice is to specify the predictors for training using either `PredictorNames` or `formula`, but not both.

Example: `"PredictorNames",["SepalLength","SepalWidth","PetalLength","PetalWidth"]`

Data Types: `string` | `cell`

`Prior` — Prior probability for each class for two-class learning

`'empirical'` (default) | `'uniform'` | numeric vector | structure array

Prior probability for each class for two-class learning, specified as the comma-separated pair consisting of `'Prior'` and a value in this table.

Value	Description
`'empirical'`	The class prior probabilities are the class relative frequencies in `Y`.
`'uniform'`	All class prior probabilities are equal to $1/K$, where K is the number of classes.
numeric vector	Each element in the vector is a class prior probability. Order the elements according to `Mdl.ClassNames` or specify the order using the `ClassNames` name-value pair argument. The software normalizes the elements to sum to `1`.
structure	A structure `S` with two fields: • `S.ClassNames` contains the class names as a variable of the same type as `Y`. • `S.ClassProbs` contains a vector of corresponding prior probabilities. The software normalizes the elements of the vector to sum to `1`.

If you specify a cost matrix, then the software updates the prior probabilities by incorporating the penalties described in the cost matrix for training. The software stores the user-specified prior probabilities in the `Prior` property of the trained model object after normalizing the probabilities to sum to `1`. For more details on the relationships and algorithmic behavior of `BoxConstraint`, `Cost`, `Prior`, `Standardize`, and `Weights`, see Algorithms.

This argument is valid only for two-class learning.

Example: `struct('ClassNames',{{'setosa','versicolor','virginica'}},'ClassProbs',1:3)`
Data Types: `char` | `string` | `double` | `single` | `struct`

`ResponseName` — Response variable name

`"Y"` (default) | character vector | string scalar

Response variable name, specified as a character vector or string scalar.

- If you supply `Y`, then you can use `ResponseName` to specify a name for the response variable.
- If you supply `ResponseVarName` or `formula`, then you cannot use `ResponseName`.

Example: `ResponseName="response"`

Data Types: `char` | `string`

`ScoreTransform` — Score transformation

`"none"` (default) | `"doublelogit"` | `"invlogit"` | `"ismax"` | `"logit"` | function handle | ...

Score transformation, specified as a character vector, string scalar, or function handle.

This table summarizes the available character vectors and string scalars.

Value	Description
`"doublelogit"`	$1/(1 + e^{-2x})$
`"invlogit"`	$\log(x / (1 - x))$

Value	Description
`"ismax"`	Sets the score for the class with the largest score to 1, and sets the scores for all other classes to 0
`"logit"`	$1/(1 + e^{-x})$
`"none"` or `"identity"`	x (no transformation)
`"sign"`	−1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
`"symmetric"`	$2x - 1$
`"symmetricismax"`	Sets the score for the class with the largest score to 1, and sets the scores for all other classes to −1
`"symmetriclogit"`	$2/(1 + e^{-x}) - 1$

For a MATLAB function or a function you define, use its function handle for the score transform. The function handle must accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Example: `ScoreTransform="logit"`
Data Types: `char` | `string` | `function_handle`

`Weights` — Observation weights
numeric vector | name of variable in `Tbl`

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a numeric vector of positive values or the name of a variable in `Tbl`. The software weighs the observations in each row of `X` or `Tbl` with the corresponding value in `Weights`. The size of `Weights` must equal the number of rows in `X` or `Tbl`.

If you specify the input data as a table `Tbl`, then `Weights` can be the name of a variable in `Tbl` that contains a numeric vector. In this case, you must specify `Weights` as a character vector or string scalar. For example, if the weights vector `W` is stored as `Tbl.W`, then specify it as `'W'`. Otherwise, the software treats all columns of `Tbl`, including `W`, as predictors or the response variable when training the model.

By default, `Weights` is `ones(`*n*`,1)`, where *n* is the number of observations in `X` or `Tbl`.

The software normalizes `Weights` to sum up to the value of the prior probability in the respective class. `Inf` weights are not supported. For more details on the relationships and algorithmic behavior of `BoxConstraint`, `Cost`, `Prior`, `Standardize`, and `Weights`, see Algorithms.

Data Types: `double` | `single` | `char` | `string`

Note

You cannot use any cross-validation name-value argument together with the `OptimizeHyperparameters` name-value argument. You can modify the cross-validation for `OptimizeHyperparameters` only by using the `HyperparameterOptimizationOptions` name-value argument.

Cross-Validation Options

`CrossVal` — Flag to train cross-validated classifier
`'off'` (default) | `'on'`

Flag to train a cross-validated classifier, specified as the comma-separated pair consisting of `'Crossval'` and `'on'` or `'off'`.

If you specify `'on'`, then the software trains a cross-validated classifier with 10 folds.

You can override this cross-validation setting using the `CVPartition`, `Holdout`, `KFold`, or `Leaveout` name-value pair argument. You can use only one cross-validation name-value pair argument at a time to create a cross-validated model.

Alternatively, cross-validate later by passing `Mdl` to `crossval`.

Example: `'Crossval','on'`

`CVPartition` — Cross-validation partition
`[]` (default) | `cvpartition` object

Cross-validation partition, specified as a `cvpartition` object that specifies the type of cross-validation and the indexing for the training and validation sets.

To create a cross-validated model, you can specify only one of these four name-value arguments: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: Suppose you create a random partition for 5-fold cross-validation on 500 observations by using `cvp = cvpartition(500,KFold=5)`. Then, you can specify the cross-validation partition by setting `CVPartition=cvp`.

`Holdout` — Fraction of data for holdout validation
scalar value in the range (0,1)

Fraction of the data used for holdout validation, specified as a scalar value in the range (0,1). If you specify `Holdout=p`, then the software completes these steps:

1. Randomly select and reserve `p*100`% of the data as validation data, and train the model using the rest of the data.
2. Store the compact trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can specify only one of these four name-value arguments: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `Holdout=0.1`

Data Types: `double` | `single`

`KFold` — Number of folds
`10` (default) | positive integer value greater than 1

Number of folds to use in the cross-validated model, specified as a positive integer value greater than 1. If you specify `KFold=k`, then the software completes these steps:

1. Randomly partition the data into `k` sets.
2. For each set, reserve the set as validation data, and train the model using the other $k - 1$ sets.

3. Store the k compact trained models in a k-by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can specify only one of these four name-value arguments: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `KFold=5`
Data Types: `single` | `double`

`Leaveout` — Leave-one-out cross-validation flag
`"off"` (default) | `"on"`

Leave-one-out cross-validation flag, specified as `"on"` or `"off"`. If you specify `Leaveout="on"`, then for each of the n observations (where n is the number of observations, excluding missing observations, specified in the `NumObservations` property of the model), the software completes these steps:

1. Reserve the one observation as validation data, and train the model using the other n – 1 observations.
2. Store the n compact trained models in an n-by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can specify only one of these four name-value arguments: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `Leaveout="on"`

Data Types: `char` | `string`

Convergence Control Options

collapse all

`DeltaGradientTolerance` — Tolerance for gradient difference
nonnegative scalar

Tolerance for the gradient difference between upper and lower violators obtained by Sequential Minimal Optimization (SMO) or Iterative Single Data Algorithm (ISDA), specified as the comma-separated pair consisting of `'DeltaGradientTolerance'` and a nonnegative scalar.

If `DeltaGradientTolerance` is `0`, then the software does not use the tolerance for the gradient difference to check for optimization convergence.

The default values are:

- `1e-3` if the solver is SMO (for example, you set `'Solver','SMO'`)
- `0` if the solver is ISDA (for example, you set `'Solver','ISDA'`)

Example: `'DeltaGradientTolerance',1e-2`

Data Types: `double` | `single`

`GapTolerance` — Feasibility gap tolerance
`0` (default) | nonnegative scalar

Feasibility gap tolerance obtained by SMO or ISDA, specified as the comma-separated pair consisting of `'GapTolerance'` and a nonnegative scalar.

If `GapTolerance` is `0`, then the software does not use the feasibility gap tolerance to check for optimization convergence.

Example: `'GapTolerance',1e-2`
Data Types: `double` | `single`

`IterationLimit` — Maximal number of numerical optimization iterations
`1e6` (default) | positive integer

Maximal number of numerical optimization iterations, specified as the comma-separated pair consisting of `'IterationLimit'` and a positive integer.

The software returns a trained model regardless of whether the optimization routine successfully converges. `Mdl.ConvergenceInfo` contains convergence information.

Example: `'IterationLimit',1e8`

Data Types: `double` | `single`

`KKTTolerance` — Karush-Kuhn-Tucker complementarity conditions violation tolerance
nonnegative scalar

Karush-Kuhn-Tucker (KKT) complementarity conditions violation tolerance, specified as the comma-separated pair consisting of `'KKTTolerance'` and a nonnegative scalar.

If `KKTTolerance` is `0`, then the software does not use the KKT complementarity conditions violation tolerance to check for optimization convergence.

The default values are:

- `0` if the solver is SMO (for example, you set `'Solver','SMO'`)
- `1e-3` if the solver is ISDA (for example, you set `'Solver','ISDA'`)

Example: `'KKTTolerance',1e-2`

Data Types: `double` | `single`

`ShrinkagePeriod` — Number of iterations between reductions of active set
`0` (default) | nonnegative integer

Number of iterations between reductions of the active set, specified as the comma-separated pair consisting of `'ShrinkagePeriod'` and a nonnegative integer.

If you set `'ShrinkagePeriod',0`, then the software does not shrink the active set.

Example: `'ShrinkagePeriod',1000`

Data Types: `double` | `single`

Hyperparameter Optimization Options

`OptimizeHyperparameters` — Parameters to optimize for two-class learning
`'none'` (default) | `'auto'` | `'all'` | string array or cell array of eligible parameter names | vector of `optimizableVariable` objects

Parameters to optimize for two-class learning, specified as the comma-separated pair consisting of `'OptimizeHyperparameters'` and one of these values:

- `'none'` — Do not optimize.
- `'auto'` — Use `{'BoxConstraint','KernelScale','Standardize'}`.
- `'all'` — Optimize all eligible parameters.
- String array or cell array of eligible parameter names.
- Vector of `optimizableVariable` objects, typically the output of `hyperparameters`.

The optimization attempts to minimize the cross-validation loss (error) for `fitcsvm` by varying the parameters. To control the cross-validation type and other aspects of the optimization, use the `HyperparameterOptimizationOptions` name-value argument. When you use `HyperparameterOptimizationOptions`, you can use the (compact) model size instead of the cross-validation loss as the optimization objective by setting the `ConstraintType` and `ConstraintBounds` options.

> **Note**
>
> The values of `OptimizeHyperparameters` override any values you specify using other name-value arguments. For example, setting `OptimizeHyperparameters` to `"auto"` causes `fitcsvm` to optimize hyperparameters corresponding to the `"auto"` option and to ignore any specified values for the hyperparameters.

The eligible parameters for `fitcsvm` are:

- `BoxConstraint` — `fitcsvm` searches among positive values, by default log-scaled in the range `[1e-3,1e3]`.
- `KernelFunction` — `fitcsvm` searches among `'gaussian'`, `'linear'`, and `'polynomial'`.
- `KernelScale` — `fitcsvm` searches among positive values, by default log-scaled in the range `[1e-3,1e3]`.
- `PolynomialOrder` — `fitcsvm` searches among integers in the range `[2,4]`.
- `Standardize` — `fitcsvm` searches among `'true'` and `'false'`.

Set nondefault parameters by passing a vector of `optimizableVariable` objects that have nondefault values. For example:

```
load fisheriris
params = hyperparameters('fitcsvm',meas,species);
params(1).Range = [1e-4,1e6];
```

Pass `params` as the value of `OptimizeHyperparameters`.

By default, the iterative display appears at the command line, and plots appear according to the number of hyperparameters in the optimization. For the optimization and plots, the objective function is the misclassification rate. To control the iterative display, set the `Verbose` option of the `HyperparameterOptimizationOptions` name-value argument. To control the plots, set the `ShowPlots` field of the `HyperparameterOptimizationOptions` name-value argument.

For an example, see Optimize SVM Classifier.

This argument is valid only for two-class learning.

Example: `'OptimizeHyperparameters','auto'`

`HyperparameterOptimizationOptions` — Optimization options for two-class learning
structure

Optimization options for two-class learning, specified as a `HyperparameterOptimizationOptions` object or a structure. This argument modifies the effect of the `OptimizeHyperparameters` name-value argument. If you specify `HyperparameterOptimizationOptions`, you must also specify `OptimizeHyperparameters`. All the options are optional. However, you must set `ConstraintBounds` and `ConstraintType` to return `AggregateOptimizationResults`. The options that you can set in a structure are the same as those in the `HyperparameterOptimizationOptions` object, and are described in the following table.

Option	Values	Default
`Optimizer`	• `"bayesopt"` — Use Bayesian optimization. Internally, this setting calls `bayesopt`. • `"gridsearch"` — Use grid search with `NumGridDivisions` values per dimension. `"gridsearch"` searches in a random order, using uniform sampling without replacement from the grid. After optimization, you can get a table in grid order by using the command `sortrows(Mdl.HyperparameterOptimizationResults)`. • `"randomsearch"` — Search at random among `MaxObjectiveEvaluations` points.	`"bayes`
`ConstraintBounds`	Constraint bounds for N optimization problems, specified as an N-by-2 numeric matrix or `[]`. The columns of `ConstraintBounds` contain the lower and upper bound values of the optimization problems. If you specify `ConstraintBounds` as a numeric vector, the software assigns the values to the second column of `ConstraintBounds`, and zeros to the first column. If you specify `ConstraintBounds`, you must also specify `ConstraintType`.	`[]`
`ConstraintTarget`	Constraint target for the optimization problems, specified as `"matlab"` or `"coder"`. If `ConstraintBounds` and `ConstraintType` are `[]` and you set `ConstraintTarget`, then the software sets `ConstraintTarget` to `[]`. The values of `ConstraintTarget` and `ConstraintType` determine the objective and constraint functions. For more information, see `HyperparameterOptimizationOptions`.	If you s `Constr` and `Constr` then th value is Otherw default
`ConstraintType`	Constraint type for the optimization problems, specified as `"size"` or `"loss"`. If you specify `ConstraintType`, you must also specify `ConstraintBounds`. The values of `ConstraintTarget` and `ConstraintType` determine the objective and constraint functions. For more information, see `HyperparameterOptimizationOptions`.	`[]`
`AcquisitionFunctionName`	Type of acquisition function: • `"expected-improvement-per-second-plus"` • `"expected-improvement"` • `"expected-improvement-plus"` • `"expected-improvement-per-second"` • `"lower-confidence-bound"` • `"probability-of-improvement"`	`"expec improv second`

Option	Values	Default
	Acquisition functions whose names include `per-second` do not yield reproducible results, because the optimization depends on the run time of the objective function. Acquisition functions whose names include `plus` modify their behavior when they overexploit an area. For more details, see Acquisition Function Types.	
`MaxObjectiveEvaluations`	Maximum number of objective function evaluations. If you specify multiple optimization problems using `ConstraintBounds`, the value of `MaxObjectiveEvaluations` applies to each optimization problem individually.	`30` for " and `"rando` and the for `"gr`
`MaxTime`	Time limit for the optimization, specified as a nonnegative real scalar. The time limit is in seconds, as measured by `tic` and `toc`. The software performs at least one optimization iteration, regardless of the value of `MaxTime`. The run time can exceed `MaxTime` because `MaxTime` does not interrupt function evaluations. If you specify multiple optimization problems using `ConstraintBounds`, the time limit applies to each optimization problem individually.	`Inf`
`NumGridDivisions`	For `Optimizer="gridsearch"`, the number of values in each dimension. The value can be a vector of positive integers giving the number of values for each dimension, or a scalar that applies to all dimensions. The software ignores this option for categorical variables.	`10`
`ShowPlots`	Logical value indicating whether to show plots of the optimization progress. If this option is `true`, the software plots the best observed objective function value against the iteration number. If you use Bayesian optimization (`Optimizer="bayesopt"`), the software also plots the best estimated objective function value. The best observed objective function values and best estimated objective function values correspond to the values in the `BestSoFar (observed)` and `BestSoFar (estim.)` columns of the iterative display, respectively. You can find these values in the properties `ObjectiveMinimumTrace` and `EstimatedObjectiveMinimumTrace` of `Mdl.HyperparameterOptimizationResults`. If the problem includes one or two optimization parameters for Bayesian optimization, then `ShowPlots` also plots a model of the objective function against the parameters.	`true`
`SaveIntermediateResults`	Logical value indicating whether to save the optimization results. If this option is `true`, the software overwrites a workspace variable named `"BayesoptResults"` at each iteration. The variable is a `BayesianOptimization` object. If you specify multiple optimization problems using `ConstraintBounds`, the workspace variable is an `AggregateBayesianOptimization` object named `"AggregateBayesoptResults"`.	`false`
`Verbose`	Display level at the command line: • `0` — No iterative display • `1` — Iterative display • `2` — Iterative display with additional information	`1`

Option	Values	Default
	For details, see the `bayesopt` `Verbose` name-value argument and the example Optimize Classifier Fit Using Bayesian Optimization.	
`UseParallel`	Logical value indicating whether to run the Bayesian optimization in parallel, which requires Parallel Computing Toolbox™. Due to the nonreproducibility of parallel timing, parallel Bayesian optimization does not necessarily yield reproducible results. For details, see Parallel Bayesian Optimization.	`false`
`Repartition`	Logical value indicating whether to repartition the cross-validation at every iteration. If this option is `false`, the optimizer uses a single partition for the optimization. A value of `true` usually gives the most robust results because this setting takes partitioning noise into account. However, for optimal results, `true` requires at least twice as many function evaluations.	`false`
Specify only one of the following three options.		
`CVPartition`	`cvpartition` object created by `cvpartition`	`KFold=` not spe validati
`Holdout`	Scalar in the range `(0,1)` representing the holdout fraction	
`KFold`	Integer greater than 1	

This argument is valid only for two-class learning.

Example: `'HyperparameterOptimizationOptions',struct('MaxObjectiveEvaluations',60)`
Data Types: `struct`

Output Arguments

collapse all

`Mdl` — Trained SVM classification model
`ClassificationSVM` model object | `ClassificationPartitionedModel` cross-validated model object

Trained SVM classification model, returned as a `ClassificationSVM` model object or `ClassificationPartitionedModel` cross-validated model object.

If you set any of the name-value pair arguments `KFold`, `Holdout`, `Leaveout`, `CrossVal`, or `CVPartition`, then `Mdl` is a `ClassificationPartitionedModel` cross-validated model object. Otherwise, `Mdl` is a `ClassificationSVM` model object.

To reference properties of `Mdl`, use dot notation. For example, enter `Mdl.Alpha` in the Command Window to display the trained Lagrange multipliers.

If you specify `OptimizeHyperparameters` and set the `ConstraintType` and `ConstraintBounds` options of `HyperparameterOptimizationOptions`, then `Mdl` is an N-by-1 cell array of model objects, where N is equal to the number of rows in `ConstraintBounds`. If none of the optimization problems yields a feasible model, then each cell array value is `[]`.

`AggregateOptimizationResults` — Aggregate optimization results
`AggregateBayesianOptimization` object

Aggregate optimization results for multiple optimization problems, returned as an `AggregateBayesianOptimization` object. To return `AggregateOptimizationResults`, you must specify `OptimizeHyperparameters` and `HyperparameterOptimizationOptions`. You must also specify the `ConstraintType` and `ConstraintBounds` options of `HyperparameterOptimizationOptions`. For an example that shows how to produce this output, see Hyperparameter Optimization with Multiple Constraint Bounds.

Limitations

- `fitcsvm` trains SVM classifiers for one-class or two-class learning applications. To train SVM classifiers using data with more than two classes, use `fitcecoc`.
- `fitcsvm` supports low-dimensional and moderate-dimensional data sets. For high-dimensional data sets, use `fitclinear` instead.

More About

collapse all

Box Constraint

A box constraint is a parameter that controls the maximum penalty imposed on margin-violating observations, which helps to prevent overfitting (regularization).

If you increase the box constraint, then the SVM classifier assigns fewer support vectors. However, increasing the box constraint can lead to longer training times.

Gram Matrix

The Gram matrix of a set of n vectors $\{x_1,..,x_n; x_j \in R^p\}$ is an n-by-n matrix with element (j,k) defined as $G(x_j,x_k) = <\phi(x_j),\phi(x_k)>$, an inner product of the transformed predictors using the kernel function ϕ.

For nonlinear SVM, the algorithm forms a Gram matrix using the rows of the predictor data X. The dual formalization replaces the inner product of the observations in X with corresponding elements of the resulting Gram matrix (called the “kernel trick”). Consequently, nonlinear SVM operates in the transformed predictor space to find a separating hyperplane.

Karush-Kuhn-Tucker (KKT) Complementarity Conditions

KKT complementarity conditions are optimization constraints required for optimal nonlinear programming solutions.

In SVM, the KKT complementarity conditions are

$$\begin{cases} \alpha_j[y_j f(x_j) - 1 + \xi_j] = 0 \\ \xi_j(C - \alpha_j) = 0 \end{cases}$$

for all $j = 1,...,n$, where $f(x_j) = \phi(x_j)'\beta + b$, ϕ is a kernel function (see Gram matrix), and ξ_j is a slack variable. If the classes are perfectly separable, then $\xi_j = 0$ for all $j = 1,...,n$.

One-Class Learning

One-class learning, or unsupervised SVM, aims to separate data from the origin in the high-dimensional predictor space (not the original predictor space), and is an algorithm used for outlier detection.

The algorithm resembles that of SVM for binary classification. The objective is to minimize the dual expression

$$0.5\sum_{jk}\alpha_j\alpha_k G(x_j,x_k)$$

with respect to $\alpha_1,...,\alpha_n$, subject to

$$\sum\alpha_j = n\nu$$

and $0 \le \alpha_j \le 1$ for all j = 1,...,n. The value of $G(x_j,x_k)$ is in element (j,k) of the Gram matrix.

A small value of ν leads to fewer support vectors and, therefore, a smooth, crude decision boundary. A large value of ν leads to more support vectors and, therefore, a curvy, flexible decision boundary. The optimal value of ν should be large enough to capture the data complexity and small enough to avoid overtraining. Also, 0 < ν ≤ 1.

For more details, see [5].

Support Vector

Support vectors are observations corresponding to strictly positive estimates of $\alpha_1,...,\alpha_n$.

SVM classifiers that yield fewer support vectors for a given training set are preferred.

Support Vector Machines for Binary Classification

The SVM binary classification algorithm searches for an optimal hyperplane that separates the data into two classes. For separable classes, the optimal hyperplane maximizes a *margin* (space that does not contain any observations) surrounding itself, which creates boundaries for the positive and negative classes. For inseparable classes, the objective is the same, but the algorithm imposes a penalty on the length of the margin for every observation that is on the wrong side of its class boundary.

The linear SVM score function is

$$f(x) = x'\beta + b,$$

where:

- x is an observation (corresponding to a row of `X`).
- The vector β contains the coefficients that define an orthogonal vector to the hyperplane (corresponding to `Mdl.Beta`). For separable data, the optimal margin length is $2/\|\beta\|$.
- b is the bias term (corresponding to `Mdl.Bias`).

The root of $f(x)$ for particular coefficients defines a hyperplane. For a particular hyperplane, $f(z)$ is the distance from point z to the hyperplane.

The algorithm searches for the maximum margin length, while keeping observations in the positive (y = 1) and negative (y = −1) classes separate.

- For separable classes, the objective is to minimize $\|\beta\|$ with respect to the β and b subject to $y_j f(x_j) \ge 1$, for all j = 1,..,n. This is the *primal* formalization for separable classes.
- For inseparable classes, the algorithm uses slack variables (ξ_j) to penalize the objective function for observations that cross the margin boundary for their class. ξ_j = 0 for observations that do not cross the margin boundary for their class, otherwise $\xi_j \ge 0$.

  The objective is to minimize $0.5\|\beta\|^2 + C\sum\xi_j$ with respect to the β, b, and ξ_j subject to $y_j f(x_j) \ge 1-\xi_j$ and $\xi_j \ge 0$ for all j = 1,..,n, and for a positive scalar box constraint C. This is the primal formalization for inseparable classes.

The algorithm uses the Lagrange multipliers method to optimize the objective, which introduces n coefficients $\alpha_1,...,\alpha_n$ (corresponding to `Mdl.Alpha`). The dual formalizations for linear SVM are as follows:

- For separable classes, minimize

  $$0.5\sum_{j=1}^{n}\sum_{k=1}^{n}\alpha_j\alpha_k y_j y_k x_j' x_k - \sum_{j=1}^{n}\alpha_j$$

  with respect to $\alpha_1,...,\alpha_n$, subject to $\sum \alpha_j y_j = 0$, $\alpha_j \geq 0$ for all j = 1,...,n, and Karush-Kuhn-Tucker (KKT) complementarity conditions.
- For inseparable classes, the objective is the same as for separable classes, except for the additional condition $0 \leq \alpha_j \leq C$ for all j = 1,..,n.

The resulting score function is

$$\hat{f}(x) = \sum_{j=1}^{n}\hat{\alpha}_j y_j x' x_j + \hat{b}.$$

$\hat{b}$ is the estimate of the bias and $\hat{\alpha}_j$ is the jth estimate of the vector $\hat{\alpha}$, j = 1,...,n. Written this way, the score function is free of the estimate of β as a result of the primal formalization.

The SVM algorithm classifies a new observation z using $\text{sign}\left(\hat{f}(z)\right)$.

In some cases, a nonlinear boundary separates the classes. *Nonlinear SVM* works in a transformed predictor space to find an optimal, separating hyperplane.

The dual formalization for nonlinear SVM is

$$0.5\sum_{j=1}^{n}\sum_{k=1}^{n}\alpha_j\alpha_k y_j y_k G(x_j, x_k) - \sum_{j=1}^{n}\alpha_j$$

with respect to $\alpha_1,...,\alpha_n$, subject to $\sum \alpha_j y_j = 0$, $0 \leq \alpha_j \leq C$ for all j = 1,..,n, and the KKT complementarity conditions. $G(x_k,x_j)$ are elements of the Gram matrix. The resulting score function is

$$\hat{f}(x) = \sum_{j=1}^{n}\hat{\alpha}_j y_j G(x, x_j) + \hat{b}.$$

For more details, see Understanding Support Vector Machines, [1], and [3].

Tips

- Unless your data set is large, always try to standardize the predictors (see `Standardize`). Standardization makes predictors insensitive to the scales on which they are measured.
- It is a good practice to cross-validate using the `KFold` name-value pair argument. The cross-validation results determine how well the SVM classifier generalizes.
- For one-class learning:
  - The default setting for the name-value pair argument `Alpha` can lead to long training times. To speed up training, set `Alpha` to a vector mostly composed of `0`s.
  - Set the name-value pair argument `Nu` to a value closer to `0` to yield fewer support vectors and, therefore, a smoother but crude decision boundary.
- Sparsity in support vectors is a desirable property of an SVM classifier. To decrease the number of support vectors, set `BoxConstraint` to a large value. This action increases the training time.
- For optimal training time, set `CacheSize` as high as the memory limit your computer allows.
- If you expect many fewer support vectors than observations in the training set, then you can significantly speed up convergence by shrinking the active set using the name-value pair argument `'ShrinkagePeriod'`. It is a good practice to specify `'ShrinkagePeriod',1000`.

- Duplicate observations that are far from the decision boundary do not affect convergence. However, just a few duplicate observations that occur near the decision boundary can slow down convergence considerably. To speed up convergence, specify `'RemoveDuplicates',true` if:
  - Your data set contains many duplicate observations.
  - You suspect that a few duplicate observations fall near the decision boundary.

  To maintain the original data set during training, `fitcsvm` must temporarily store separate data sets: the original and one without the duplicate observations. Therefore, if you specify `true` for data sets containing few duplicates, then `fitcsvm` consumes close to double the memory of the original data.
- After training a model, you can generate C/C++ code that predicts labels for new data. Generating C/C++ code requires MATLAB Coder™. For details, see Introduction to Code Generation.

Algorithms

- For the mathematical formulation of the SVM binary classification algorithm, see Support Vector Machines for Binary Classification and Understanding Support Vector Machines.
- `NaN`, `<undefined>`, empty character vector (`''`), empty string (`""`), and `<missing>` values indicate missing values. `fitcsvm` removes entire rows of data corresponding to a missing response. When computing total weights (see the next bullets), `fitcsvm` ignores any weight corresponding to an observation with at least one missing predictor. This action can lead to unbalanced prior probabilities in balanced-class problems. Consequently, observation box constraints might not equal `BoxConstraint`.
- If you specify the `Cost`, `Prior`, and `Weights` name-value arguments, the output model object stores the specified values in the `Cost`, `Prior`, and `W` properties, respectively. The `Cost` property stores the user-specified cost matrix (C) without modification. The `Prior` and `W` properties store the prior probabilities and observation weights, respectively, after normalization. For model training, the software updates the prior probabilities and observation weights to incorporate the penalties described in the cost matrix. For details, see Misclassification Cost Matrix, Prior Probabilities, and Observation Weights.

  Note that the `Cost` and `Prior` name-value arguments are used for two-class learning. For one-class learning, the `Cost` and `Prior` properties store `0` and `1`, respectively.
- For two-class learning, `fitcsvm` assigns a box constraint to each observation in the training data. The formula for the box constraint of observation j is

  $$C_j = nC_0w_j^*,$$

  where C_0 is the initial box constraint (see the `BoxConstraint` name-value argument), and w_j^* is the observation weight adjusted by `Cost` and `Prior` for observation j. For details about the observation weights, see Adjust Prior Probabilities and Observation Weights for Misclassification Cost Matrix.
- If you specify `Standardize` as `true` and set the `Cost`, `Prior`, or `Weights` name-value argument, then `fitcsvm` standardizes the predictors using their corresponding weighted means and weighted standard deviations. That is, `fitcsvm` standardizes predictor j (x_j) using

  $$x_j^* = \frac{x_j - \mu_j^*}{\sigma_j^*},$$

  where x_{jk} is observation k (row) of predictor j (column), and

  $$\mu_j^* = \frac{1}{\sum_k w_k^*}\sum_k w_k^* x_{jk},$$

  $$\left(\sigma_j^*\right)^2 = \frac{v_1}{v_1^2 - v_2}\sum_k w_k^*\left(x_{jk} - \mu_j^*\right)^2,$$

  $$v_1 = \sum_j w_j^*,$$

  $$v_2 = \sum_j \left(w_j^*\right)^2.$$
- Assume that `p` is the proportion of outliers that you expect in the training data, and that you set `'OutlierFraction',p`.

  - For one-class learning, the software trains the bias term such that 100p% of the observations in the training data have negative scores.
  - The software implements *robust learning* for two-class learning. In other words, the software attempts to remove 100p% of the observations when the optimization algorithm converges. The removed observations correspond to gradients that are large in magnitude.
- If your predictor data contains categorical variables, then the software generally uses full dummy encoding for these variables. The software creates one dummy variable for each level of each categorical variable.
  - The `PredictorNames` property stores one element for each of the original predictor variable names. For example, assume that there are three predictors, one of which is a categorical variable with three levels. Then `PredictorNames` is a 1-by-3 cell array of character vectors containing the original names of the predictor variables.
  - The `ExpandedPredictorNames` property stores one element for each of the predictor variables, including the dummy variables. For example, assume that there are three predictors, one of which is a categorical variable with three levels. Then `ExpandedPredictorNames` is a 1-by-5 cell array of character vectors containing the names of the predictor variables and the new dummy variables.
  - Similarly, the `Beta` property stores one beta coefficient for each predictor, including the dummy variables.
  - The `SupportVectors` property stores the predictor values for the support vectors, including the dummy variables. For example, assume that there are m support vectors and three predictors, one of which is a categorical variable with three levels. Then `SupportVectors` is an m-by-5 matrix.
  - The `X` property stores the training data as originally input and does not include the dummy variables. When the input is a table, `X` contains only the columns used as predictors.
- For predictors specified in a table, if any of the variables contain ordered (ordinal) categories, the software uses ordinal encoding for these variables.
  - For a variable with k ordered levels, the software creates $k-1$ dummy variables. The jth dummy variable is -1 for levels up to j, and $+1$ for levels $j+1$ through k.
  - The names of the dummy variables stored in the `ExpandedPredictorNames` property indicate the first level with the value $+1$. The software stores $k-1$ additional predictor names for the dummy variables, including the names of levels 2, 3, ..., k.
- All solvers implement $L1$ soft-margin minimization.
- For one-class learning, the software estimates the Lagrange multipliers, $\alpha_1,...,\alpha_n$, such that

$$\sum_{j=1}^{n}\alpha_j = n\nu.$$

Alternative Functionality

You can also use the `ocsvm` function to train a one-class SVM model for anomaly detection.

- The `ocsvm` function provides a simpler and preferred workflow for anomaly detection than the `fitcsvm` function.
  - The `ocsvm` function returns a `OneClassSVM` object, anomaly indicators, and anomaly scores. You can use the outputs to identify anomalies in training data. To find anomalies in new data, you can use the `isanomaly` object function of `OneClassSVM`. The `isanomaly` function returns anomaly indicators and scores for the new data.
  - The `fitcsvm` function supports both one-class and binary classification. If the class label variable contains only one class (for example, a vector of ones), `fitcsvm` trains a model for one-class classification and returns a `ClassificationSVM` object. To identify anomalies, you must first compute anomaly scores by using the `resubPredict` or `predict` object function of `ClassificationSVM`, and then identify anomalies by finding observations that have negative scores.
  - Note that a large positive anomaly score indicates an anomaly in `ocsvm`, whereas a negative score indicates an anomaly in `predict` of `ClassificationSVM`.
- The `ocsvm` function finds the decision boundary based on the primal form of SVM, whereas the `fitcsvm` function finds the decision boundary based on the dual form of SVM.

- The solver in `ocsvm` is computationally less expensive than the solver in `fitcsvm` for a large data set (large n). Unlike solvers in `fitcsvm`, which require computation of the n-by-n Gram matrix, the solver in `ocsvm` only needs to form a matrix of size n-by-m. Here, m is the number of dimensions of expanded space, which is typically much less than n for big data.

References

[1] Christianini, N., and J. C. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge, UK: Cambridge University Press, 2000.

[2] Fan, R.-E., P.-H. Chen, and C.-J. Lin. "Working set selection using second order information for training support vector machines." *Journal of Machine Learning Research*, Vol. 6, 2005, pp. 1889–1918.

[3] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, Second Edition. NY: Springer, 2008.

[4] Kecman V., T. -M. Huang, and M. Vogt. "Iterative Single Data Algorithm for Training Kernel Machines from Huge Data Sets: Theory and Performance." *Support Vector Machines: Theory and Applications*. Edited by Lipo Wang, 255–274. Berlin: Springer-Verlag, 2005.

[5] Scholkopf, B., J. C. Platt, J. C. Shawe-Taylor, A. J. Smola, and R. C. Williamson. "Estimating the Support of a High-Dimensional Distribution." *Neural Comput*., Vol. 13, Number 7, 2001, pp. 1443–1471.

[6] Scholkopf, B., and A. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization and Beyond, Adaptive Computation and Machine Learning*. Cambridge, MA: The MIT Press, 2002.

Extended Capabilities

collapse all

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To perform parallel hyperparameter optimization, use the `UseParallel=true` option in the `HyperparameterOptimizationOptions` name-value argument in the call to the `fitcsvm` function.

For more information on parallel hyperparameter optimization, see Parallel Bayesian Optimization.

For general information about parallel computing, see Run MATLAB Functions with Automatic Parallel Support (Parallel Computing Toolbox).

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- One-class classification is not supported. The labels must contain two different classes.
- You cannot specify the `KernelFunction` name-value argument as a custom kernel function.
- You can specify the `Solver` name-value argument only as `"SMO"`.
- You cannot specify the `OutlierFraction` or `ShrinkagePeriod` name-value argument.
- The predictor data cannot contain infinite values.
- `fitcsvm` fits the model on a GPU if one of the following applies:
  - The input argument `X` is a `gpuArray` object.
  - The input argument `Tbl` contains `gpuArray` predictor variables.

For more information, see Run MATLAB Functions on a GPU (Parallel Computing Toolbox).

Version History

Introduced in R2014a

R2025a: Compute serially when parallel hyperparameter optimization is not available⚠

`fitcsvm` defaults to serial hyperparameter optimization when `HyperparameterOptimizationOptions` includes `UseParallel=true` and the software cannot open a parallel pool.

In previous releases, the software issues an error under these circumstances.

R2023b: `"auto"` option of `OptimizeHyperparameters` includes `Standardize`⚠

Starting in R2023b, when you specify `"auto"` as the `OptimizeHyperparameters` value, `fitcsvm` includes `Standardize` as an optimizable hyperparameter.

See Also

`ClassificationSVM` | `CompactClassificationSVM` | `ClassificationPartitionedModel` | `predict` | `fitSVMPosterior` | `rng` | `quadprog` (Optimization Toolbox) | `fitcecoc` | `fitclinear` | `ocsvm`

Topics

Train SVM Classifiers Using Gaussian Kernel

Train SVM Classifier Using Custom Kernel

Optimize Cross-Validated Classifier Using bayesopt

Optimize Classifier Fit Using Bayesian Optimization

Understanding Support Vector Machines

Unsupervised Anomaly Detection