Support Vector Machines for Binary Classification

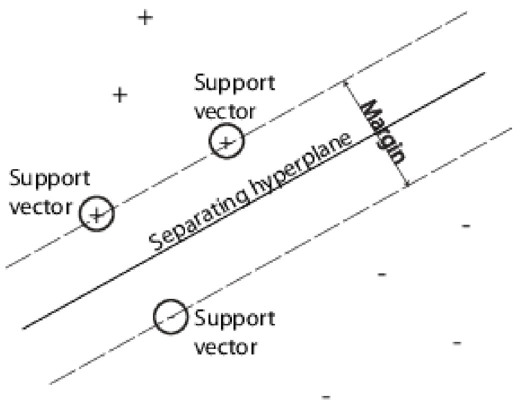
Understanding Support Vector Machines

- Separable Data
- Nonseparable Data
- Nonlinear Transformation with Kernels

Separable Data

You can use a support vector machine (SVM) when your data has exactly two classes. An SVM classifies data by finding the best hyperplane that separates all data points of one class from those of the other class. The *best* hyperplane for an SVM means the one with the largest *margin* between the two classes. Margin means the maximal width of the slab parallel to the hyperplane that has no interior data points.

The *support vectors* are the data points that are closest to the separating hyperplane; these points are on the boundary of the slab. The following figure illustrates these definitions, with + indicating data points of type 1, and – indicating data points of type –1.



Mathematical Formulation: Primal. This discussion follows Hastie, Tibshirani, and Friedman [1] and Christianini and Shawe-Taylor [2].

The data for training is a set of points (vectors) x_j along with their categories y_j. For some dimension d, the $x_j \in R^d$, and the $y_j = \pm 1$. The equation of a hyperplane is

$$f(x) = x'\beta + b = 0$$

where $\beta \in R^d$ and b is a real number.

The following problem defines the *best* separating hyperplane (i.e., the decision boundary). Find β and b that minimize $||\beta||$ such that for all data points (x_j,y_j),

$$y_j f(x_j) \geq 1.$$

The support vectors are the x_j on the boundary, those for which $y_j f(x_j) = 1.$

For mathematical convenience, the problem is usually given as the equivalent problem of minimizing $\|\beta\|$. This is a quadratic programming problem. The optimal solution $(\hat{\beta}, \hat{b})$ enables classification of a vector z as follows:

$$\text{class}(z) = \text{sign}(z'\hat{\beta} + \hat{b}) = \text{sign}(\hat{f}(z)).$$

$\hat{f}(z)$ is the *classification score* and represents the distance z is from the decision boundary.

Mathematical Formulation: Dual. It is computationally simpler to solve the dual quadratic programming problem. To obtain the dual, take positive Lagrange multipliers α_j multiplied by each constraint, and subtract from the objective function:

$$L_P = \frac{1}{2}\beta'\beta - \sum_j \alpha_j (y_j (x_j{'}\beta + b) - 1),$$

where you look for a stationary point of L_P over β and b. Setting the gradient of L_P to 0, you get

$$\beta = \sum_j \alpha_j y_j x_j$$
$$0 = \sum_j \alpha_j y_j. \quad \textbf{(1)}$$

Substituting into L_P, you get the dual L_D:

$$L_D = \sum_j \alpha_j - \frac{1}{2}\sum_j \sum_k \alpha_j \alpha_k y_j y_k x_j{}' x_k,$$

which you maximize over $\alpha_j \geq 0$. In general, many α_j are 0 at the maximum. The nonzero α_j in the solution to the dual problem define the hyperplane, as seen in Equation 1, which gives β as the sum of $\alpha_j y_j x_j$. The data points x_j corresponding to nonzero α_j are the *support vectors*.

The derivative of L_D with respect to a nonzero α_j is 0 at an optimum. This gives

$$y_j f(x_j) - 1 = 0.$$

In particular, this gives the value of b at the solution, by taking any j with nonzero α_j.

The dual is a standard quadratic programming problem. For example, the Optimization Toolbox™ `quadprog` solver solves this type of problem.

Nonseparable Data

Your data might not allow for a separating hyperplane. In that case, SVM can use a *soft margin*, meaning a hyperplane that separates many, but not all data points.

There are two standard formulations of soft margins. Both involve adding slack variables ξ_j and a penalty parameter C.

- The L^1-norm problem is:

  $$\min_{\beta, b, \xi}\left(\frac{1}{2}\beta'\beta + C\sum_j \xi_j\right)$$

  such that

  $$y_j f(x_j) \geq 1 - \xi_j$$
  $$\xi_j \geq 0.$$

  The L^1-norm refers to using ξ_j as slack variables instead of their squares. The three solver options `SMO`, `ISDA`, and `L1QP` of `fitcsvm` minimize the L^1-norm problem.
- The L^2-norm problem is:

  $$\min_{\beta, b, \xi}\left(\frac{1}{2}\beta'\beta + C\sum_j \xi_j^2\right)$$

  subject to the same constraints.

In these formulations, you can see that increasing C places more weight on the slack variables ξ_j, meaning the optimization attempts to make a stricter separation between classes. Equivalently, reducing C towards 0 makes misclassification less important.

Mathematical Formulation: Dual. For easier calculations, consider the L^1 dual problem to this soft-margin formulation. Using Lagrange multipliers μ_j, the function to minimize for the L^1-norm problem is:

$$L_P = \frac{1}{2}\beta'\beta + C\sum_j \xi_j - \sum_j \alpha_j(y_i f(x_j) - (1 - \xi_j)) - \sum_j \mu_j \xi_j,$$

where you look for a stationary point of L_P over β, b, and positive ξ_j. Setting the gradient of L_P to 0, you get

$$\beta = \sum_j \alpha_j y_j x_j$$
$$\sum_j \alpha_j y_j = 0$$
$$\alpha_j = C - \mu_j$$
$$\alpha_j, \mu_j, \xi_j \geq 0.$$

These equations lead directly to the dual formulation:

$$\max_\alpha \sum_j \alpha_j - \frac{1}{2}\sum_j \sum_k \alpha_j \alpha_k y_j y_k x_j{}' x_k$$

subject to the constraints

$$\sum_j y_j \alpha_j = 0$$
$$0 \leq \alpha_j \leq C.$$

The final set of inequalities, $0 \leq \alpha_j \leq C$, shows why C is sometimes called a *box constraint*. C keeps the allowable values of the Lagrange multipliers α_j in a "box", a bounded region.

The gradient equation for b gives the solution b in terms of the set of nonzero α_j, which correspond to the support vectors.

You can write and solve the dual of the L^2-norm problem in an analogous manner. For details, see Christianini and Shawe-Taylor [2], Chapter 6.

`fitcsvm` Implementation. Both dual soft-margin problems are quadratic programming problems. Internally, `fitcsvm` has several different algorithms for solving the problems.

- For one-class or binary classification, if you do not set a fraction of expected outliers in the data (see `OutlierFraction`), then the default solver is Sequential Minimal Optimization (SMO). SMO minimizes the one-norm problem by a series of two-point minimizations. During optimization, SMO respects the linear constraint $\sum_i \alpha_i y_i = 0,$ and explicitly includes the bias term in the model. SMO is relatively fast. For more details on SMO, see [3].
- For binary classification, if you set a fraction of expected outliers in the data, then the default solver is the Iterative Single Data Algorithm. Like SMO, ISDA solves the one-norm problem. Unlike SMO, ISDA minimizes by a series on one-point minimizations, does not respect the linear constraint, and does not explicitly include the bias term in the model. For more details on ISDA, see [4].
- For one-class or binary classification, and if you have an Optimization Toolbox license, you can choose to use `quadprog` to solve the one-norm problem. `quadprog` uses a good deal of memory, but solves quadratic programs to a high degree of precision. For more details, see Quadratic Programming Definition (Optimization Toolbox).

Nonlinear Transformation with Kernels

Some binary classification problems do not have a simple hyperplane as a useful separating criterion. For those problems, there is a variant of the mathematical approach that retains nearly all the simplicity of an SVM separating hyperplane.

This approach uses these results from the theory of reproducing kernels:

- There is a class of functions $G(x_1,x_2)$ with the following property. There is a linear space S and a function φ mapping x to S such that

  $G(x_1,x_2) = \langle\varphi(x_1),\varphi(x_2)\rangle$.

  The dot product takes place in the space S.
- This class of functions includes:
  - Polynomials: For some positive integer p,

    $G(x_1,x_2) = (1 + x_1'x_2)^p$.
  - Radial basis function (Gaussian):

    $G(x_1,x_2) = \exp(-\|x_1-x_2\|^2)$.
  - Multilayer perceptron or sigmoid (neural network): For a positive number p_1 and a negative number p_2,

    $G(x_1,x_2) = \tanh(p_1 x_1'x_2 + p_2)$.

> **Note**
>
> - Not every set of p_1 and p_2 yields a valid reproducing kernel.
> - `fitcsvm` does not support the sigmoid kernel. Instead, you can define the sigmoid kernel and specify it by using the `'KernelFunction'` name-value pair argument. For details, see Train SVM Classifier Using Custom Kernel.

The mathematical approach using kernels relies on the computational method of hyperplanes. All the calculations for hyperplane classification use nothing more than dot products. Therefore, nonlinear kernels can use identical calculations and solution algorithms, and obtain classifiers that are nonlinear. The resulting classifiers are hypersurfaces in some space S, but the space S does not have to be identified or examined.

Using Support Vector Machines

As with any supervised learning model, you first train a support vector machine, and then cross validate the classifier. Use the trained machine to classify (predict) new data. In addition, to obtain satisfactory predictive accuracy, you can use various SVM kernel functions, and you must tune the parameters of the kernel functions.

- Training an SVM Classifier

- Classifying New Data with an SVM Classifier
- Tuning an SVM Classifier

Training an SVM Classifier

Train, and optionally cross validate, an SVM classifier using `fitcsvm`. The most common syntax is:

```
SVMModel = fitcsvm(X,Y,'KernelFunction','rbf',...
    'Standardize',true,'ClassNames',{'negClass','posClass'});
```

The inputs are:

- `X` — Matrix of predictor data, where each row is one observation, and each column is one predictor.
- `Y` — Array of class labels with each row corresponding to the value of the corresponding row in `X`. `Y` can be a categorical, character, or string array, a logical or numeric vector, or a cell array of character vectors.
- `KernelFunction` — The default value is `'linear'` for two-class learning, which separates the data by a hyperplane. The value `'gaussian'` (or `'rbf'`) is the default for one-class learning, and specifies to use the Gaussian (or radial basis function) kernel. An important step to successfully train an SVM classifier is to choose an appropriate kernel function.
- `Standardize` — Flag indicating whether the software should standardize the predictors before training the classifier.
- `ClassNames` — Distinguishes between the negative and positive classes, or specifies which classes to include in the data. The negative class is the first element (or row of a character array), e.g., `'negClass'`, and the positive class is the second element (or row of a character array), e.g., `'posClass'`. `ClassNames` must be the same data type as `Y`. It is good practice to specify the class names, especially if you are comparing the performance of different classifiers.

The resulting, trained model (`SVMModel`) contains the optimized parameters from the SVM algorithm, enabling you to classify new data.

For more name-value pairs you can use to control the training, see the `fitcsvm` reference page.

Classifying New Data with an SVM Classifier

Classify new data using `predict`. The syntax for classifying new data using a trained SVM classifier (`SVMModel`) is:

```
[label,score] = predict(SVMModel,newX);
```

The resulting vector, `label`, represents the classification of each row in `X`. `score` is an n-by-2 matrix of soft scores. Each row corresponds to a row in `X`, which is a new observation. The first column contains the scores for the observations being classified in the negative class, and the second column contains the scores observations being classified in the positive class.

To estimate posterior probabilities rather than scores, first pass the trained SVM classifier (`SVMModel`) to `fitPosterior`, which fits a score-to-posterior-probability transformation function to the scores. The syntax is:

```
ScoreSVMModel = fitPosterior(SVMModel,X,Y);
```

The property `ScoreTransform` of the classifier `ScoreSVMModel` contains the optimal transformation function. Pass `ScoreSVMModel` to `predict`. Rather than returning the scores, the output argument `score` contains the posterior probabilities of an observation being classified in the negative (column 1 of `score`) or positive (column 2 of `score`) class.

Tuning an SVM Classifier

Use the `'OptimizeHyperparameters'` name-value pair argument of `fitcsvm` to find parameter values that minimize the cross-validation loss. The eligible parameters are `'BoxConstraint'`, `'KernelFunction'`, `'KernelScale'`, `'PolynomialOrder'`, and `'Standardize'`. For an example, see Optimize an SVM Classifier Fit Using Bayesian Optimization. Alternatively, you can use the `bayesopt` function, as shown in Optimize a Cross-Validated SVM Classifier Using bayesopt. The `bayesopt` function allows more flexibility to customize optimization. You can use the `bayesopt` function to optimize any parameters, including parameters that are not eligible to optimize when you use the `fitcsvm` function.

You can also try tuning parameters of your classifier manually according to this scheme:

1. Pass the data to `fitcsvm`, and set the name-value pair argument `'KernelScale','auto'`. Suppose that the trained SVM model is called `SVMModel`. The software uses a heuristic procedure to select the kernel scale. The heuristic procedure uses subsampling. Therefore, to reproduce results, set a random number seed using `rng` before training the classifier.
2. Cross validate the classifier by passing it to `crossval`. By default, the software conducts 10-fold cross validation.
3. Pass the cross-validated SVM model to `kfoldLoss` to estimate and retain the classification error.

4. Retrain the SVM classifier, but adjust the `'KernelScale'` and `'BoxConstraint'` name-value pair arguments.

   - `BoxConstraint` — One strategy is to try a geometric sequence of the box constraint parameter. For example, take 11 values, from `1e-5` to `1e5` by a factor of 10. Increasing `BoxConstraint` might decrease the number of support vectors, but also might increase training time.
   - `KernelScale` — One strategy is to try a geometric sequence of the RBF sigma parameter scaled at the original kernel scale. Do this by:

     a. Retrieving the original kernel scale, e.g., `ks`, using dot notation: `ks = SVMModel.KernelParameters.Scale`.

     b. Use as new kernel scales factors of the original. For example, multiply `ks` by the 11 values `1e-5` to `1e5`, increasing by a factor of 10.

Choose the model that yields the lowest classification error. You might want to further refine your parameters to obtain better accuracy. Start with your initial parameters and perform another cross-validation step, this time using a factor of 1.2.

Train SVM Classifiers Using a Gaussian Kernel

Try it in MATLAB

This example shows how to generate a nonlinear classifier with Gaussian kernel function. First, generate one class of points inside the unit disk in two dimensions, and another class of points in the annulus from radius 1 to radius 2. Then, generates a classifier based on the data with the Gaussian radial basis function kernel. The default linear classifier is obviously unsuitable for this problem, since the model is circularly symmetric. Set the box constraint parameter to `Inf` to make a strict classification, meaning no misclassified training points. Other kernel functions might not work with this strict box constraint, since they might be unable to provide a strict classification. Even though the rbf classifier can separate the classes, the result can be overtrained.

Generate 100 points uniformly distributed in the unit disk. To do so, generate a radius *r* as the square root of a uniform random variable, generate an angle *t* uniformly in (0, 2π), and put the point at (*r* cos(*t*), *r* sin(*t*)).

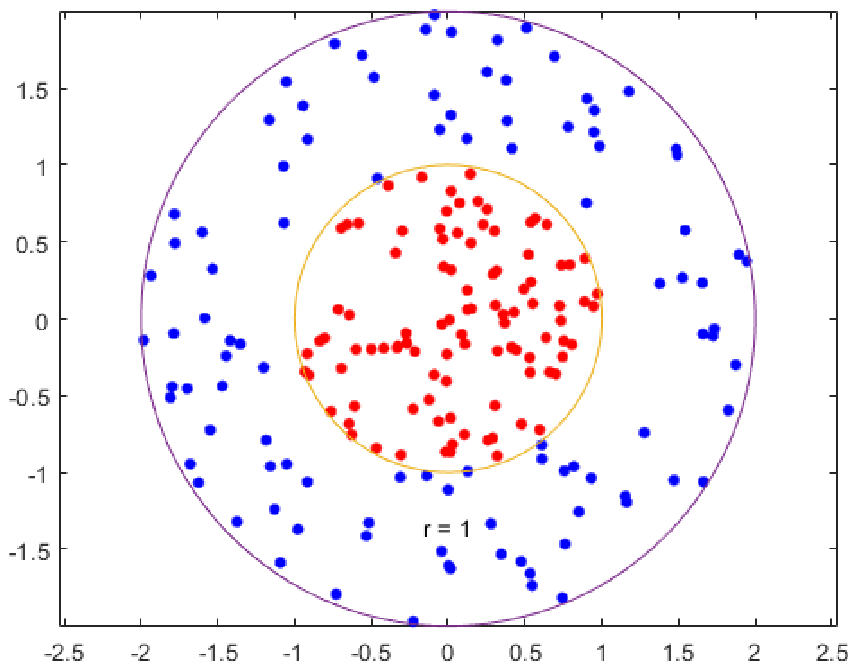
```
rng(1); % For reproducibility
r = sqrt(rand(100,1)); % Radius
t = 2*pi*rand(100,1);  % Angle
data1 = [r.*cos(t), r.*sin(t)]; % Points
```

Generate 100 points uniformly distributed in the annulus. The radius is again proportional to a square root, this time a square root of the uniform distribution from 1 through 4.

```
r2 = sqrt(3*rand(100,1)+1); % Radius
t2 = 2*pi*rand(100,1);      % Angle
data2 = [r2.*cos(t2), r2.*sin(t2)]; % points
```

Plot the points, and plot circles of radii 1 and 2 for comparison.

```
figure;
plot(data1(:,1),data1(:,2),'r.','MarkerSize',15)
hold on
plot(data2(:,1),data2(:,2),'b.','MarkerSize',15)
ezpolar(@(x)1);ezpolar(@(x)2);
axis equal
hold off
```

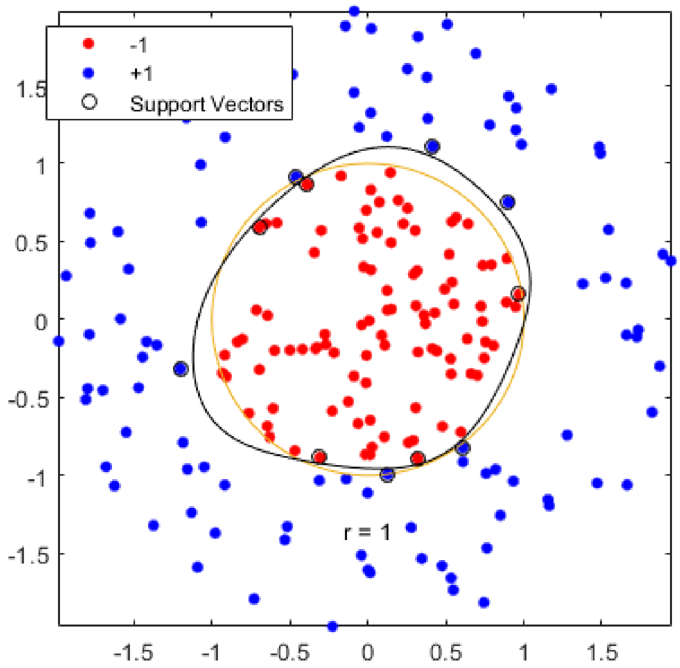


Put the data in one matrix, and make a vector of classifications.

```
data3 = [data1;data2];
theclass = ones(200,1);
theclass(1:100) = -1;
```

Train an SVM classifier with `KernelFunction` set to `'rbf'` and `BoxConstraint` set to `Inf`. Plot the decision boundary and flag the support vectors.

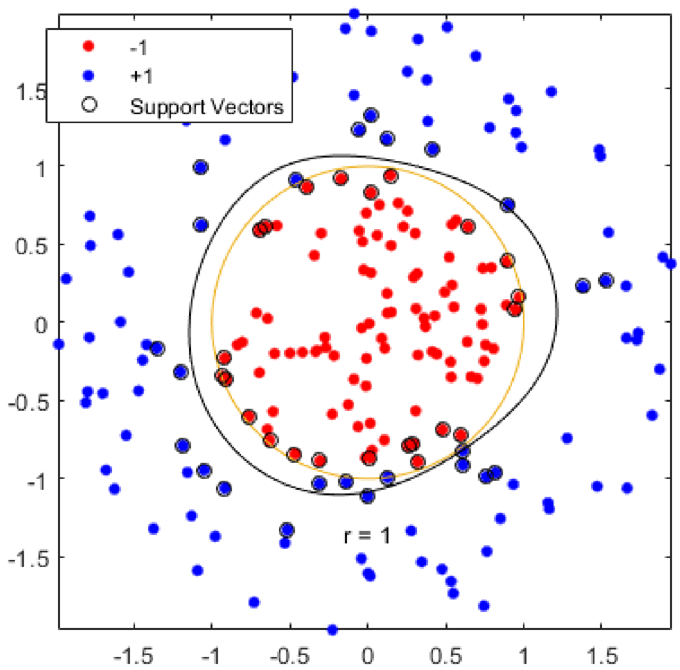
```
%Train the SVM Classifier
cl = fitcsvm(data3,theclass,'KernelFunction','rbf',...
    'BoxConstraint',Inf,'ClassNames',[-1,1]);

% Predict scores over the grid
d = 0.02;
[x1Grid,x2Grid] = meshgrid(min(data3(:,1)):d:max(data3(:,1)),...
    min(data3(:,2)):d:max(data3(:,2)));
xGrid = [x1Grid(:),x2Grid(:)];
[~,scores] = predict(cl,xGrid);

% Plot the data and the decision boundary
figure;
h(1:2) = gscatter(data3(:,1),data3(:,2),theclass,'rb','.');
hold on
ezpolar(@(x)1);
h(3) = plot(data3(cl.IsSupportVector,1),data3(cl.IsSupportVector,2),'ko');
contour(x1Grid,x2Grid,reshape(scores(:,2),size(x1Grid)),[0 0],'k');
legend(h,{'-1','+1','Support Vectors'});
axis equal
hold off
```



`fitcsvm` generates a classifier that is close to a circle of radius 1. The difference is due to the random training data.

Training with the default parameters makes a more nearly circular classification boundary, but one that misclassifies some training data. Also, the default value of `BoxConstraint` is `1`, and, therefore, there are more support vectors.

```
cl2 = fitcsvm(data3,theclass,'KernelFunction','rbf');
[~,scores2] = predict(cl2,xGrid);

figure;
h(1:2) = gscatter(data3(:,1),data3(:,2),theclass,'rb','.');
hold on
ezpolar(@(x)1);
h(3) = plot(data3(cl2.IsSupportVector,1),data3(cl2.IsSupportVector,2),'ko');
contour(x1Grid,x2Grid,reshape(scores2(:,2),size(x1Grid)),[0 0],'k');
legend(h,{'-1','+1','Support Vectors'});
axis equal
hold off
```



Train SVM Classifier Using Custom Kernel

This example shows how to use a custom kernel function, such as the sigmoid kernel, to train SVM classifiers, and adjust custom kernel function parameters.

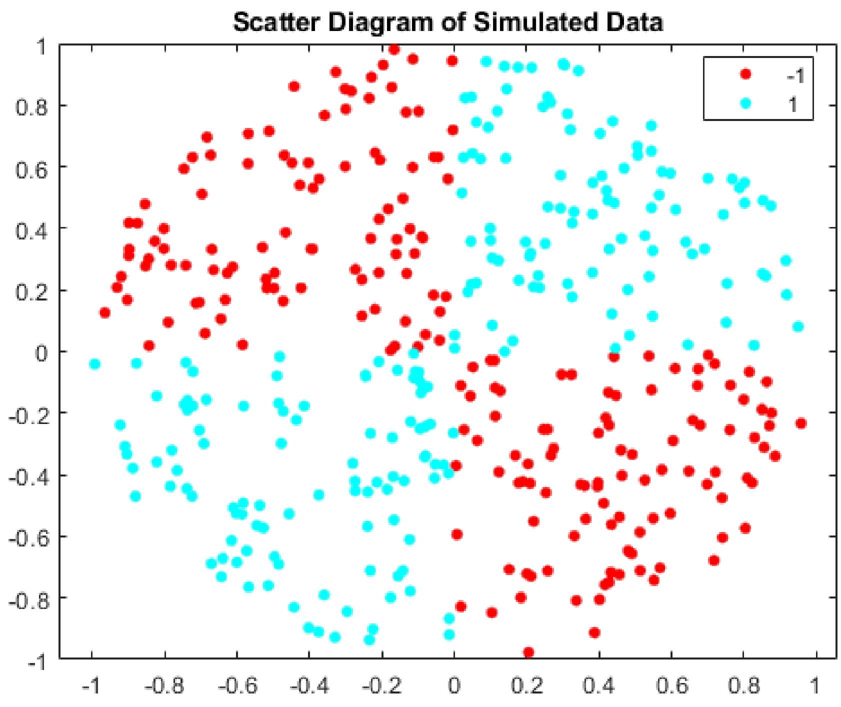
Try it in MATLAB

Generate a random set of points within the unit circle. Label points in the first and third quadrants as belonging to the positive class, and those in the second and fourth quadrants in the negative class.

```
rng(1);  % For reproducibility
n = 100; % Number of points per quadrant

r1 = sqrt(rand(2*n,1));                      % Random radii
t1 = [pi/2*rand(n,1); (pi/2*rand(n,1)+pi)]; % Random angles for Q1 and Q3
X1 = [r1.*cos(t1) r1.*sin(t1)];             % Polar-to-Cartesian conversion

r2 = sqrt(rand(2*n,1));
t2 = [pi/2*rand(n,1)+pi/2; (pi/2*rand(n,1)-pi/2)]; % Random angles for Q2 and Q4
X2 = [r2.*cos(t2) r2.*sin(t2)];

X = [X1; X2];        % Predictors
Y = ones(4*n,1);
Y(2*n + 1:end) = -1; % Labels
```

Plot the data.

```
figure;
gscatter(X(:,1),X(:,2),Y);
title('Scatter Diagram of Simulated Data')
```



Write a function that accepts two matrices in the feature space as inputs, and transforms them into a Gram matrix using the sigmoid kernel.

```
function G = mysigmoid(U,V)
% Sigmoid kernel function with slope gamma and intercept c
gamma = 1;
c = -1;
G = tanh(gamma*U*V' + c);
end
```

Save this code as a file named `mysigmoid` on your MATLAB® path.

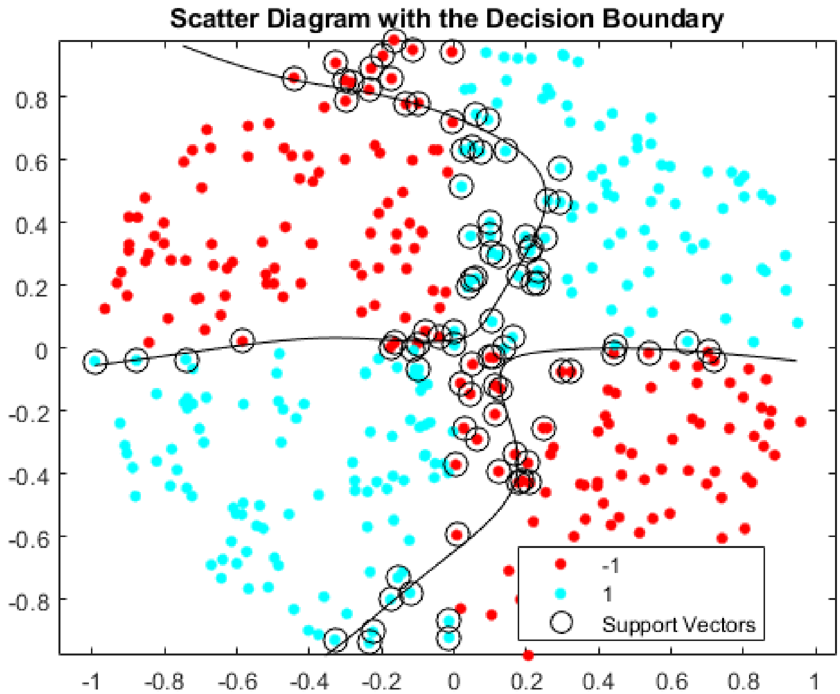
Train an SVM classifier using the sigmoid kernel function. It is good practice to standardize the data.

```
Mdl1 = fitcsvm(X,Y,'KernelFunction','mysigmoid','Standardize',true);
```

`Mdl1` is a `ClassificationSVM` classifier containing the estimated parameters.

Plot the data, and identify the support vectors and the decision boundary.

```
    % Compute the scores over a grid
d = 0.02; % Step size of the grid
[x1Grid,x2Grid] = meshgrid(min(X(:,1)):d:max(X(:,1)),...
    min(X(:,2)):d:max(X(:,2)));
xGrid = [x1Grid(:),x2Grid(:)];        % The grid
[~,scores1] = predict(Mdl1,xGrid); % The scores

figure;
h(1:2) = gscatter(X(:,1),X(:,2),Y);
hold on
h(3) = plot(X(Mdl1.IsSupportVector,1),...
    X(Mdl1.IsSupportVector,2),'ko','MarkerSize',10);
    % Support vectors
contour(x1Grid,x2Grid,reshape(scores1(:,2),size(x1Grid)),[0 0],'k');
    % Decision boundary
title('Scatter Diagram with the Decision Boundary')
legend({'-1','1','Support Vectors'},'Location','Best');
hold off
```



You can adjust the kernel parameters in an attempt to improve the shape of the decision boundary. This might also decrease the within-sample misclassification rate, but, you should first determine the out-of-sample misclassification rate.

Determine the out-of-sample misclassification rate by using 10-fold cross validation.

```
CVMdl1 = crossval(Mdl1);
misclass1 = kfoldLoss(CVMdl1);
misclass1
```

```
misclass1 =

    0.1375
```

The out-of-sample misclassification rate is 13.5%.

Write another sigmoid function, but Set `gamma = 0.5;`.

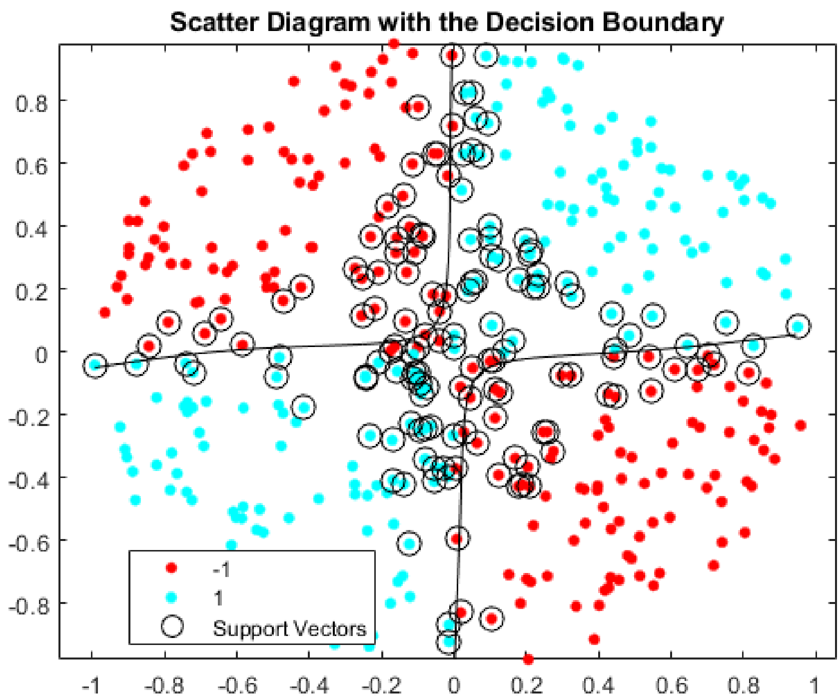
```
function G = mysigmoid2(U,V)
% Sigmoid kernel function with slope gamma and intercept c
gamma = 0.5;
c = -1;
G = tanh(gamma*U*V' + c);
end
```

Save this code as a file named `mysigmoid2` on your MATLAB® path.

Train another SVM classifier using the adjusted sigmoid kernel. Plot the data and the decision region, and determine the out-of-sample misclassification rate.

```
Mdl2 = fitcsvm(X,Y,'KernelFunction','mysigmoid2','Standardize',true);
[~,scores2] = predict(Mdl2,xGrid);

figure;
h(1:2) = gscatter(X(:,1),X(:,2),Y);
hold on
h(3) = plot(X(Mdl2.IsSupportVector,1),...
    X(Mdl2.IsSupportVector,2),'ko','MarkerSize',10);
title('Scatter Diagram with the Decision Boundary')
contour(x1Grid,x2Grid,reshape(scores2(:,2),size(x1Grid)),[0 0],'k');
legend({'-1','1','Support Vectors'},'Location','Best');
hold off

CVMdl2 = crossval(Mdl2);
misclass2 = kfoldLoss(CVMdl2);
misclass2
```

```
misclass2 =

    0.0450
```



After the sigmoid slope adjustment, the new decision boundary seems to provide a better within-sample fit, and the cross-validation rate contracts by more than 66%.

Optimize an SVM Classifier Fit Using Bayesian Optimization

Try it in MATLAB

This example shows how to optimize an SVM classification using the `fitcsvm` function and `OptimizeHyperparameters` name-value pair. The classification works on locations of points from a Gaussian mixture model. In *The Elements of Statistical Learning*, Hastie, Tibshirani, and Friedman (2009), page 17 describes the model. The model begins with generating 10 base points for a "green" class, distributed as 2-D independent normals with mean (1,0) and unit variance. It also generates 10 base points for a "red" class, distributed as 2-D independent normals with mean (0,1) and unit variance. For each class (green and red), generate 100 random points as follows:

1. Choose a base point *m* of the appropriate color uniformly at random.
2. Generate an independent random point with 2-D normal distribution with mean *m* and variance I/5, where I is the 2-by-2 identity matrix. In this example, use a variance I/50 to show the advantage of optimization more clearly.

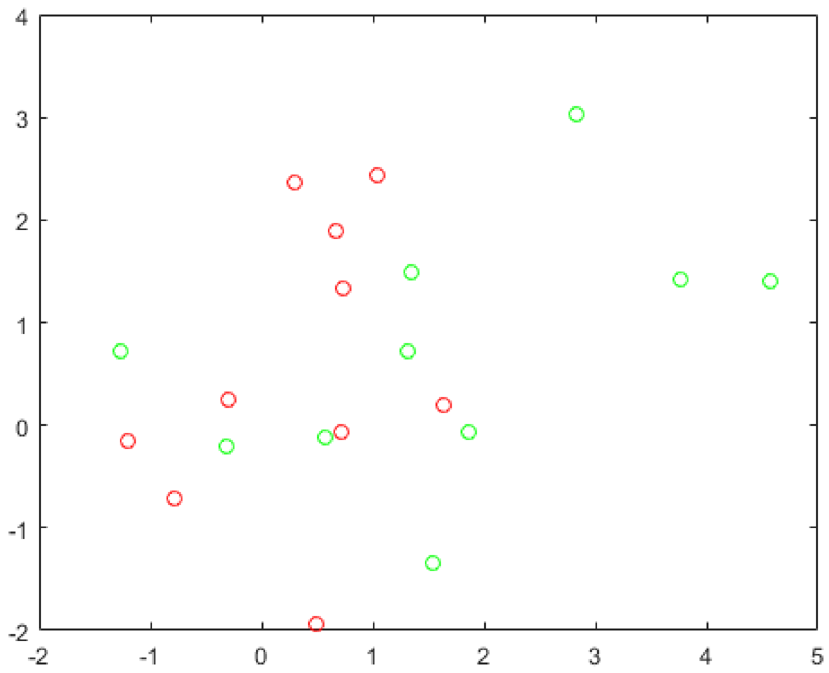
Generate the Points and Classifier

Generate the 10 base points for each class.

```
rng default % For reproducibility
grnpop = mvnrnd([1,0],eye(2),10);
redpop = mvnrnd([0,1],eye(2),10);
```

View the base points.

```
plot(grnpop(:,1),grnpop(:,2),'go')
hold on
plot(redpop(:,1),redpop(:,2),'ro')
hold off
```



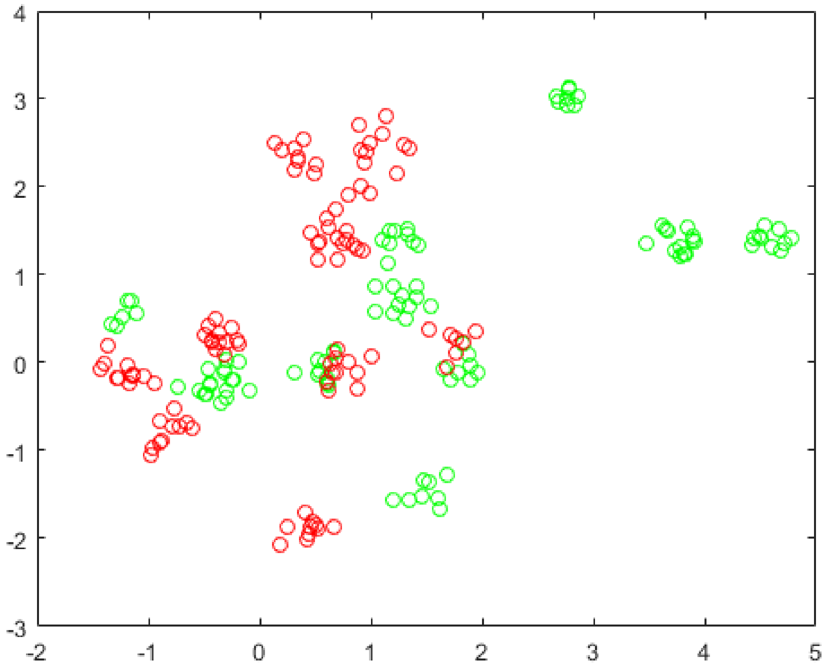
Since some red base points are close to green base points, it can be difficult to classify the data points based on location alone.

Generate the 100 data points of each class.

```
redpts = zeros(100,2);grnpts = redpts;
for i = 1:100
    grnpts(i,:) = mvnrnd(grnpop(randi(10),:),eye(2)*0.02);
    redpts(i,:) = mvnrnd(redpop(randi(10),:),eye(2)*0.02);
end
```

View the data points.

```
figure
plot(grnpts(:,1),grnpts(:,2),'go')
hold on
plot(redpts(:,1),redpts(:,2),'ro')
hold off
```



Prepare Data For Classification

Put the data into one matrix, and make a vector `grp` that labels the class of each point.

```
cdata = [grnpts;redpts];
grp = ones(200,1);
% Green label 1, red label -1
grp(101:200) = -1;
```

Prepare Cross-Validation

Set up a partition for cross-validation. This step fixes the train and test sets that the optimization uses at each step.

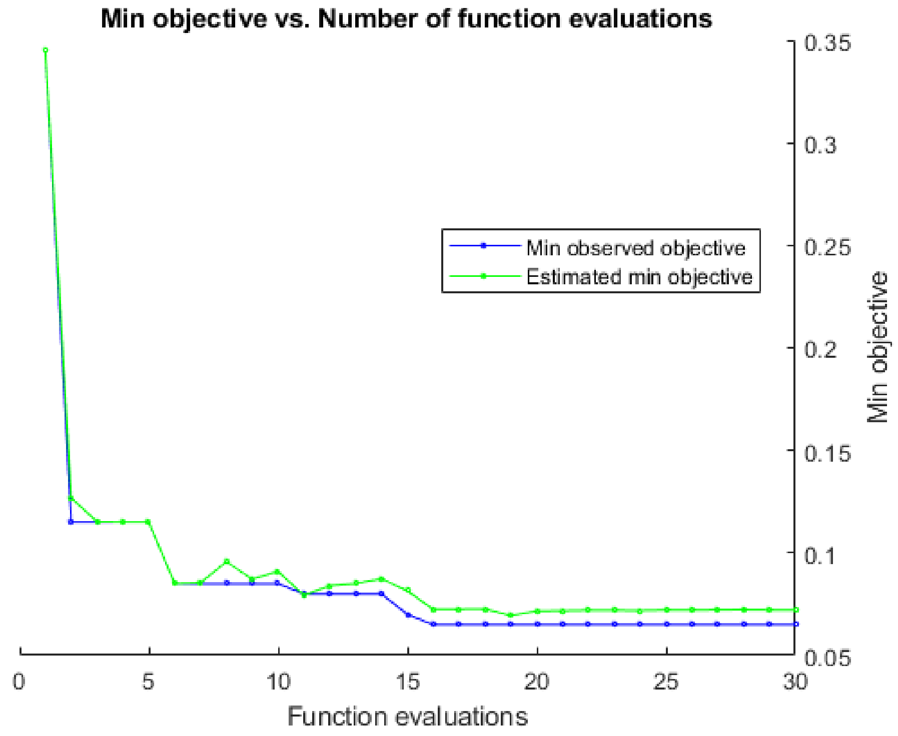
```
c = cvpartition(200,'KFold',10);
```

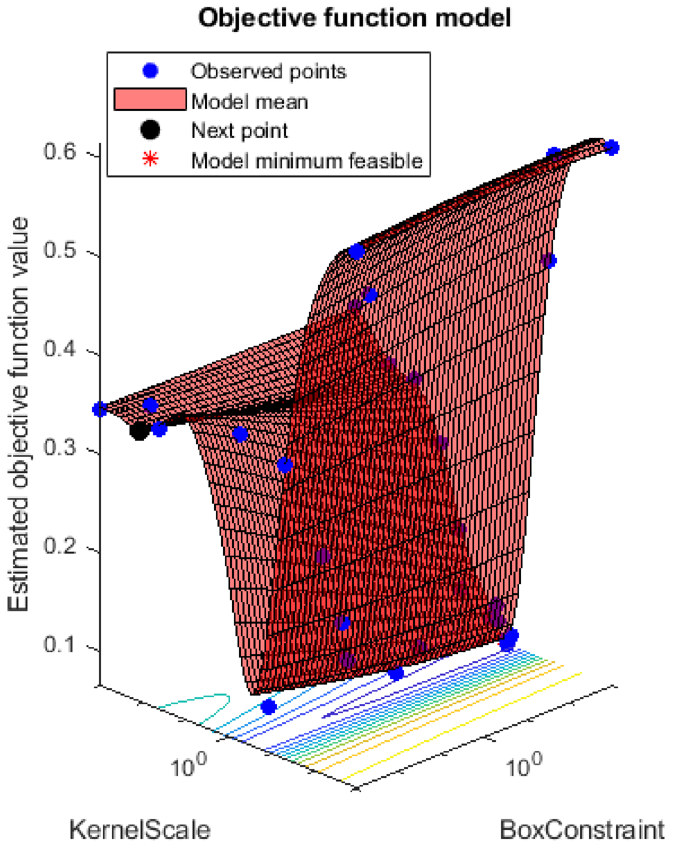
Optimize the Fit

To find a good fit, meaning one with a low cross-validation loss, set options to use Bayesian optimization. Use the same cross-validation partition `c` in all optimizations.

For reproducibility, use the `'expected-improvement-plus'` acquisition function.

```
opts = struct('Optimizer','bayesopt','ShowPlots',true,'CVPartition',c,...
    'AcquisitionFunctionName','expected-improvement-plus');
svmmod = fitcsvm(cdata,grp,'KernelFunction','rbf',...
    'OptimizeHyperparameters','auto','HyperparameterOptimizationOptions',opts)
```





Iter	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)	BoxConstraint	KernelScale
1	Best	0.345	0.34358	0.345	0.345	0.00474	306.44
2	Best	0.115	0.3085	0.115	0.12678	430.31	1.4864
3	Accept	0.52	0.36849	0.115	0.1152	0.028415	0.014369
4	Accept	0.61	0.16208	0.115	0.11504	133.94	0.0031427
5	Accept	0.34	0.25453	0.115	0.11504	0.010993	5.7742
6	Best	0.085	0.15841	0.085	0.085039	885.63	0.68403
7	Accept	0.105	0.29711	0.085	0.085428	0.3057	0.58118

```
|    8 | Accept |        0.21 |     0.52313 |       0.085 |     0.09566 |      0.16044 |      0.91824 |
|    9 | Accept |       0.085 |     0.21356 |       0.085 |     0.08725 |       972.19 |      0.46259 |
|   10 | Accept |         0.1 |     0.18961 |       0.085 |    0.090952 |       990.29 |        0.491 |
|   11 | Best   |        0.08 |     0.16313 |        0.08 |    0.079362 |       2.5195 |        0.291 |
|   12 | Accept |        0.09 |     0.15401 |        0.08 |     0.08402 |       14.338 |      0.44386 |
|   13 | Accept |         0.1 |     0.34121 |        0.08 |     0.08508 |    0.0022577 |      0.23803 |
|   14 | Accept |        0.11 |     0.36934 |        0.08 |    0.087378 |       0.2115 |      0.32109 |
|   15 | Best   |        0.07 |     0.39858 |        0.07 |    0.081507 |        910.2 |      0.25218 |
|   16 | Best   |       0.065 |     0.18705 |       0.065 |    0.072457 |       953.22 |      0.26253 |
|   17 | Accept |       0.075 |     0.30259 |       0.065 |    0.072554 |       998.74 |      0.23087 |
|   18 | Accept |       0.295 |     0.22177 |       0.065 |    0.072647 |       996.18 |       44.626 |
|   19 | Accept |        0.07 |     0.24671 |       0.065 |     0.06946 |       985.37 |      0.27389 |
|   20 | Accept |       0.165 |      0.1913 |       0.065 |    0.071622 |     0.065103 |      0.13679 |
|====================================================================================================|
| Iter | Eval   | Objective   | Objective   | BestSoFar   | BestSoFar   | BoxConstraint|  KernelScale |
|      | result |             | runtime     | (observed)  | (estim.)    |              |              |
|====================================================================================================|
|   21 | Accept |       0.345 |     0.20242 |       0.065 |    0.071764 |        971.7 |       999.01 |
|   22 | Accept |        0.61 |      0.3307 |       0.065 |    0.071967 |    0.0010168 |    0.0010005 |
|   23 | Accept |       0.345 |      0.2003 |       0.065 |    0.071959 |    0.0010674 |       999.18 |
|   24 | Accept |        0.35 |     0.12521 |       0.065 |    0.071863 |    0.0010003 |       40.628 |
|   25 | Accept |        0.24 |     0.29768 |       0.065 |    0.072124 |       996.55 |       10.423 |
|   26 | Accept |        0.61 |     0.16789 |       0.065 |    0.072068 |       958.64 |    0.0010026 |
|   27 | Accept |        0.47 |     0.14131 |       0.065 |     0.07218 |       993.69 |     0.029723 |
|   28 | Accept |         0.3 |      0.2961 |       0.065 |    0.072291 |       993.15 |       170.01 |
|   29 | Accept |        0.16 |     0.46427 |       0.065 |    0.072104 |       992.81 |       3.8594 |
|   30 | Accept |       0.365 |     0.14473 |       0.065 |    0.072112 |    0.0010017 |     0.044287 |

__________________________________________________________
Optimization completed.
MaxObjectiveEvaluations of 30 reached.
Total function evaluations: 30
Total elapsed time: 65.3337 seconds.
Total objective function evaluation time: 7.7653

Best observed feasible point:
    BoxConstraint    KernelScale
    _____________    ___________

       953.22          0.26253

Observed objective function value = 0.065
Estimated objective function value = 0.072112
Function evaluation time = 0.18705

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

       985.37          0.27389

Estimated objective function value = 0.072112
Estimated function evaluation time = 0.24085
svmmod = 
  ClassificationSVM
                         ResponseName: 'Y'
                CategoricalPredictors: []
                           ClassNames: [-1 1]
                       ScoreTransform: 'none'
                      NumObservations: 200
    HyperparameterOptimizationResults: [1x1 BayesianOptimization]
                                Alpha: [77x1 double]
                                 Bias: -0.2352
                     KernelParameters: [1x1 struct]
```

```
            BoxConstraints: [200x1 double]
           ConvergenceInfo: [1x1 struct]
           IsSupportVector: [200x1 logical]
                    Solver: 'SMO'


  Properties, Methods
```

Find the loss of the optimized model.

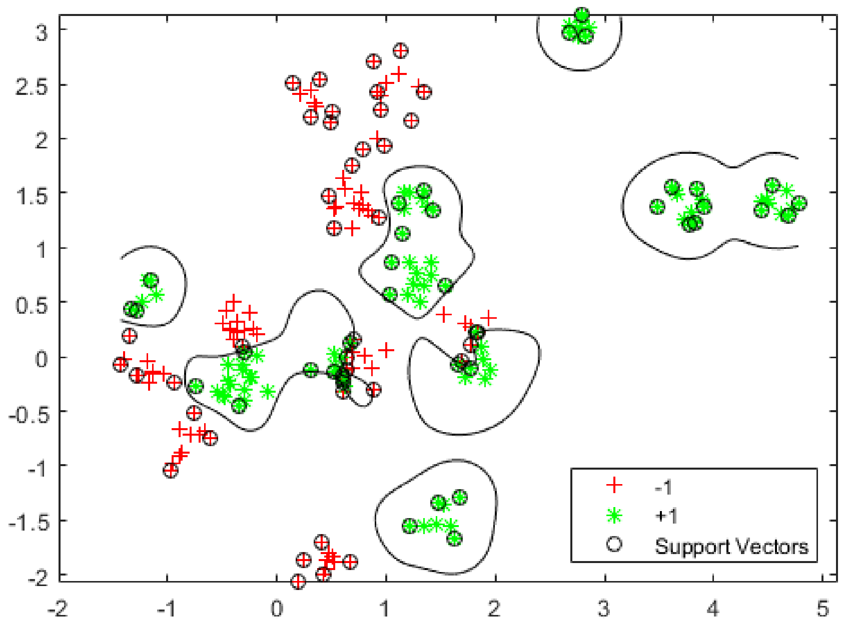
```
lossnew = kfoldLoss(fitcsvm(cdata,grp,'CVPartition',c,'KernelFunction','rbf',...
    'BoxConstraint',svmmod.HyperparameterOptimizationResults.XAtMinObjective.BoxConstraint,...
    'KernelScale',svmmod.HyperparameterOptimizationResults.XAtMinObjective.KernelScale))
```

```
lossnew = 0.0650
```

This loss is the same as the loss reported in the optimization output under "Observed objective function value".

Visualize the optimized classifier.

```
d = 0.02;
[x1Grid,x2Grid] = meshgrid(min(cdata(:,1)):d:max(cdata(:,1)),...
    min(cdata(:,2)):d:max(cdata(:,2)));
xGrid = [x1Grid(:),x2Grid(:)];
[~,scores] = predict(svmmod,xGrid);
figure;
h = nan(3,1); % Preallocation
h(1:2) = gscatter(cdata(:,1),cdata(:,2),grp,'rg','+*');
hold on
h(3) = plot(cdata(svmmod.IsSupportVector,1),...
    cdata(svmmod.IsSupportVector,2),'ko');
contour(x1Grid,x2Grid,reshape(scores(:,2),size(x1Grid)),[0 0],'k');
legend(h,{'-1','+1','Support Vectors'},'Location','Southeast');
axis equal
hold off
```



Plot Posterior Probability Regions for SVM Classification Models

This example shows how to predict posterior probabilities of SVM models over a grid of observations, and then plot the posterior probabilities over the grid. Plotting posterior probabilities exposes decision boundaries.

Try it in MATLAB

Load Fisher's iris data set. Train the classifier using the petal lengths and widths, and remove the virginica species from the data.

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

Train an SVM classifier using the data. It is good practice to specify the order of the classes.

```
SVMModel = fitcsvm(X,y,'ClassNames',{'setosa','versicolor'});
```

Estimate the optimal score transformation function.

```
rng(1); % For reproducibility
[SVMModel,ScoreParameters] = fitPosterior(SVMModel);
```

```
Warning: Classes are perfectly separated. The optimal score-to-posterior transformation is a step function.
```

```
ScoreParameters
```

```
ScoreParameters = struct with fields:
                        Type: 'step'
                  LowerBound: -0.8431
                  UpperBound: 0.6897
    PositiveClassProbability: 0.5000
```

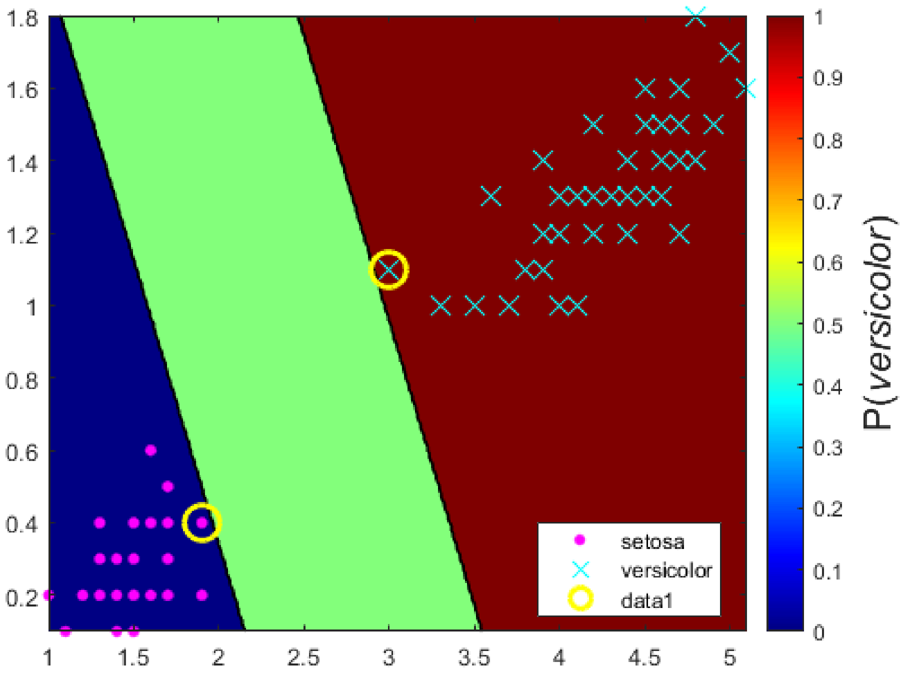
The optimal score transformation function is the step function because the classes are separable. The fields `LowerBound` and `UpperBound` of `ScoreParameters` indicate the lower and upper end points of the interval of scores corresponding to observations within the class-separating hyperplanes (the margin). No training observation falls within the margin. If a new score is in the interval, then the software assigns the corresponding observation a positive class posterior probability, i.e., the value in the `PositiveClassProbability` field of `ScoreParameters`.

Define a grid of values in the observed predictor space. Predict the posterior probabilities for each instance in the grid.

```
xMax = max(X);
xMin = min(X);
d = 0.01;
[x1Grid,x2Grid] = meshgrid(xMin(1):d:xMax(1),xMin(2):d:xMax(2));

[~,PosteriorRegion] = predict(SVMModel,[x1Grid(:),x2Grid(:)]);
```

Plot the positive class posterior probability region and the training data.

```
figure;
contourf(x1Grid,x2Grid,...
        reshape(PosteriorRegion(:,2),size(x1Grid,1),size(x1Grid,2)));
h = colorbar;
h.Label.String = 'P({\it{versicolor}})';
h.YLabel.FontSize = 16;
caxis([0 1]);
colormap jet;

hold on
gscatter(X(:,1),X(:,2),y,'mc','.x',[15,10]);
sv = X(SVMModel.IsSupportVector,:);
plot(sv(:,1),sv(:,2),'yo','MarkerSize',15,'LineWidth',2);
axis tight
hold off
```



In two-class learning, if the classes are separable, then there are three regions: one where observations have positive class posterior probability `0`, one where it is `1`, and the other where it is the positive class prior probability.

Analyze Images Using Linear Support Vector Machines

Try it in MATLAB

This example shows how to determine which quadrant of an image a shape occupies by training an error-correcting output codes (ECOC) model comprised of linear SVM binary learners. This example also illustrates the disk-space consumption of ECOC models that store support vectors, their labels, and the estimated α coefficients.

Create the Data Set

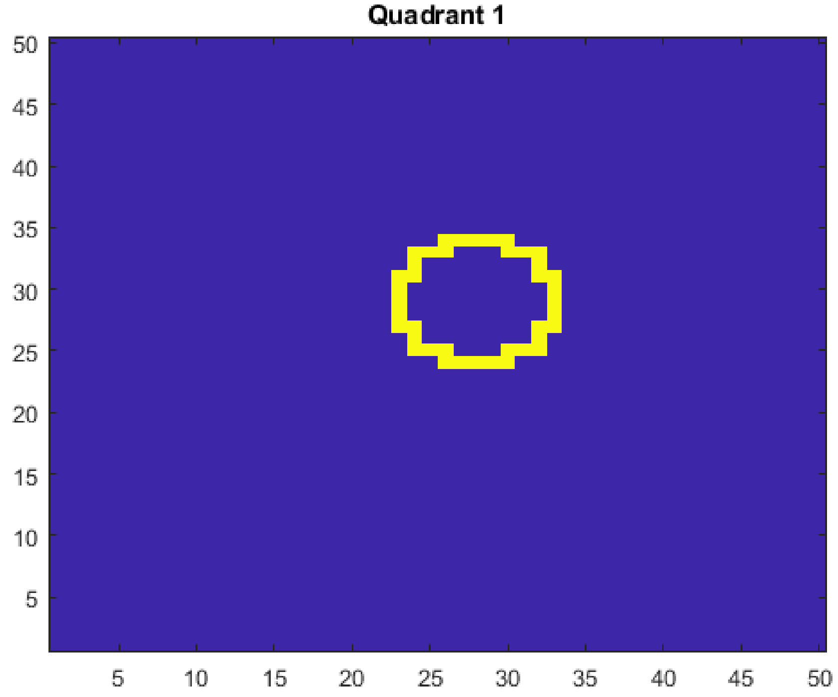
Randomly place a circle with radius five in a 50-by-50 image. Make 5000 images. Create a label for each image indicating the quadrant that the circle occupies. Quadrant 1 is in the upper right, quadrant 2 is in the upper left, quadrant 3 is in the lower left, and quadrant 4 is in the lower right. The predictors are the intensities of each pixel.

```
d = 50;  % Height and width of the images in pixels
n = 5e4; % Sample size

X = zeros(n,d^2); % Predictor matrix preallocation
Y = zeros(n,1);   % Label preallocation
theta = 0:(1/d):(2*pi);
r = 5;            % Circle radius
rng(1);           % For reproducibility

for j = 1:n
    figmat = zeros(d);                       % Empty image
    c = datasample((r + 1):(d - r - 1),2);   % Random circle center
    x = r*cos(theta) + c(1);                 % Make the circle
    y = r*sin(theta) + c(2);
    idx = sub2ind([d d],round(y),round(x));  % Convert to linear indexing
    figmat(idx) = 1;                         % Draw the circle
    X(j,:) = figmat(:);                % Store the data
    Y(j) = (c(2) >= floor(d/2)) + 2*(c(2) < floor(d/2)) + ...
        (c(1) < floor(d/2)) + ...
        2*((c(1) >= floor(d/2)) & (c(2) < floor(d/2))); % Determine the quadrant
end
```

Plot an observation.

```
figure
imagesc(figmat)
```

```
h = gca;
h.YDir = 'normal';
title(sprintf('Quadrant %d',Y(end)))
```



Train the ECOC Model

Use a 25% holdout sample and specify the training and holdout sample indices.

```
p = 0.25;
CVP = cvpartition(Y,'Holdout',p); % Cross-validation data partition
isIdx = training(CVP);            % Training sample indices
oosIdx = test(CVP);               % Test sample indices
```

Create an SVM template that specifies storing the support vectors of the binary learners. Pass it and the training data to `fitcecoc` to train the model. Determine the training sample classification error.

```
t = templateSVM('SaveSupportVectors',true);
MdlSV = fitcecoc(X(isIdx,:),Y(isIdx),'Learners',t);
isLoss = resubLoss(MdlSV)
```

```
isLoss = 0
```

`MdlSV` is a trained `ClassificationECOC` multiclass model. It stores the training data and the support vectors of each binary learner. For large data sets, such as those in image analysis, the model can consume a lot of memory.

Determine the amount of disk space that the ECOC model consumes.

```
infoMdlSV = whos('MdlSV');
mbMdlSV = infoMdlSV.bytes/1.049e6
```

```
mbMdlSV = 763.6151
```

The model consumes 763.6 MB.

Improve Model Efficiency

You can assess out-of-sample performance. You can also assess whether the model has been overfit with a compacted model that does not contain the support vectors, their related parameters, and the training data.

Discard the support vectors and related parameters from the trained ECOC model. Then, discard the training data from the resulting model by using `compact`.

```
Mdl = discardSupportVectors(MdlSV);
CMdl = compact(Mdl);
info = whos('Mdl','CMdl');
```

```
[bytesCMdl,bytesMdl] = info.bytes;
memReduction = 1 - [bytesMdl bytesCMdl]/infoMdlSV.bytes
```

```
memReduction = 1×2

    0.0626    0.9996
```

In this case, discarding the support vectors reduces the memory consumption by about 6%. Compacting and discarding support vectors reduces the size by about 99.96%.

An alternative way to manage support vectors is to reduce their numbers during training by specifying a larger box constraint, such as 100. Though SVM models that use fewer support vectors are more desirable and consume less memory, increasing the value of the box constraint tends to increase the training time.

Remove `MdlSV` and `Mdl` from the workspace.

```
clear Mdl MdlSV
```

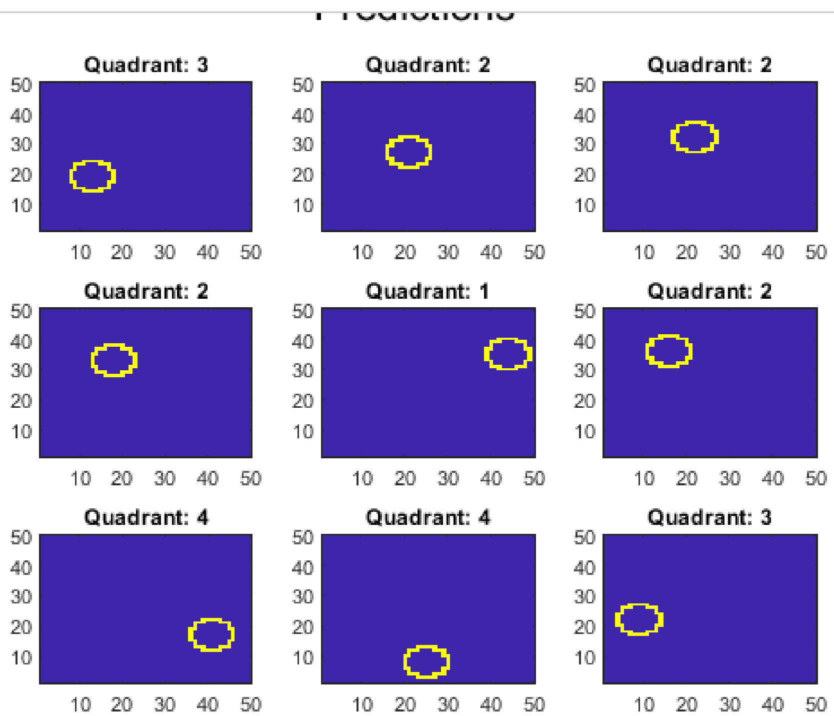
Assess Holdout Sample Performance

Calculate the classification error of the holdout sample. Plot a sample of the holdout sample predictions.

```
oosLoss = loss(CMdl,X(oosIdx,:),Y(oosIdx))
```

```
oosLoss = 0
```

```
yHat = predict(CMdl,X(oosIdx,:));
nVec = 1:size(X,1);
oosIdx = nVec(oosIdx);

figure;
for j = 1:9
    subplot(3,3,j)
    imagesc(reshape(X(oosIdx(j),:),[d d]))
    h = gca;
    h.YDir = 'normal';
    title(sprintf('Quadrant: %d',yHat(j)))
end
text(-1.33*d,4.5*d + 1,'Predictions','FontSize',17)
```



The model does not misclassify any holdout sample observations.

See Also

`bayesopt` | `fitcsvm` | `kfoldLoss`

Related Topics

- Train Support Vector Machines Using Classification Learner App
- Optimize a Cross-Validated SVM Classifier Using bayesopt

References

[1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, second edition. New York: Springer, 2008.

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

[3] 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.

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