Support Vector Machines (SVM)

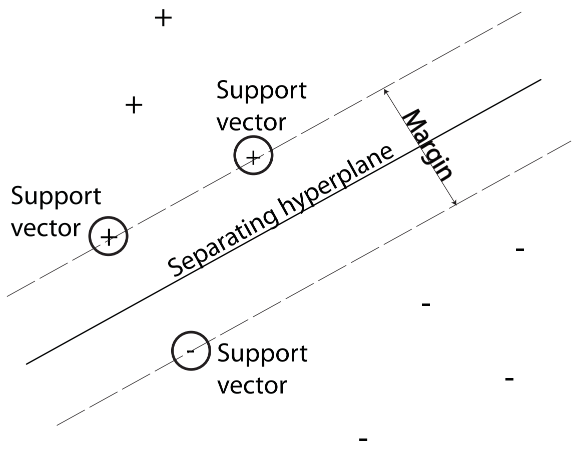
In this section...

Understanding Support Vector Machines

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 [3] and Christianini and Shawe-Taylor [2].

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

$$<w,x> + b = 0,$$

where $w \in R^d$, $<w,x>$ is the inner (dot) product of w and x, and b is real.

The following problem defines the *best* separating hyperplane. Find w and b that minimize $||w||$ such that for all data points (x_i,y_i),

$$y_i(<w,x_i> + b) \geq 1.$$

The support vectors are the x_i on the boundary, those for which $y_i(<w,x_i> + b) = 1.$

For mathematical convenience, the problem is usually given as the equivalent problem of minimizing $<w,w>/2$. This is a quadratic programming problem. The optimal solution w, b enables classification of a vector z as follows:

$$\text{class}(z) = \text{sign}(<w,z> + b).$$

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

$$L_P = \frac{1}{2}\langle w,w\rangle - \sum_i \alpha_i \left(y_i \left(\langle w, x_i \rangle + b \right) - 1 \right),$$

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

$$\begin{aligned} w &= \sum_i \alpha_i y_i x_i \\ 0 &= \sum_i \alpha_i y_i . \end{aligned} \tag{1-1}$$

Substituting into L_P, you get the dual L_D:

$$L_D = \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \left\langle x_i, x_j \right\rangle,$$

which you maximize over $\alpha_i \geq 0$. In general, many α_i are 0 at the maximum. The nonzero α_i in the solution to the dual problem define the hyperplane, as seen in Equation 1-1, which gives w as the sum of $\alpha_i y_i x_i$. The data points x_i corresponding to nonzero α_i are the *support vectors*.

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

$$y_i(<w,x_i> + b) - 1 = 0.$$

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

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 s_i and a penalty parameter C.

- The L^1-norm problem is:

  $$\min_{w,b,s}\left(\frac{1}{2}\langle w,w\rangle + C\sum_i s_i\right)$$

  such that

  $$\begin{aligned} y_i\left(\langle w,x_i\rangle + b\right) &\geq 1 - s_i \\ s_i &\geq 0. \end{aligned}$$

  The L^1-norm refers to using s_i as slack variables instead of their squares. The `SMO` `svmtrain` method minimizes the L^1-norm problem.

- The L^2-norm problem is:

  $$\min_{w,b,s}\left(\frac{1}{2}\langle w,w\rangle + C\sum_i s_i^2\right)$$

  subject to the same constraints. The `QP` `svmtrain` method minimizes the L^2-norm problem.

In these formulations, you can see that increasing C places more weight on the slack variables s_i, 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_i, the function to minimize for the L^1-norm problem is:

$$L_P = \frac{1}{2}\langle w,w\rangle + C\sum_i s_i - \sum_i \alpha_i\left(y_i\left(\langle w,x_i\rangle + b\right) - (1 - s_i)\right) - \sum_i \mu_i s_i,$$

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

$$\begin{aligned} b &= \sum_i \alpha_i y_i x_i \\ \sum_i \alpha_i y_i &= 0 \\ \alpha_i &= C - \mu_i \\ \alpha_i, \mu_i, s_i &\geq 0. \end{aligned}$$

These equations lead directly to the dual formulation:

$$\max_{\alpha} \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \left\langle x_i, x_j \right\rangle$$

subject to the constraints

$$\begin{aligned} &\sum_i y_i \alpha_i = 0 \\ &0 \leq \alpha_i \leq C. \end{aligned}$$

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

The gradient equation for b gives the solution b in terms of the set of nonzero a_i, 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.

svmtrain Implementation. Both dual soft-margin problems are quadratic programming problems. Internally, `svmtrain` has several different algorithms for solving the problems. The default Sequential Minimal Optimization (SMO) algorithm minimizes the one-norm problem. SMO is a relatively fast algorithm. If you have an Optimization Toolbox license, you can choose to use `quadprog` as the algorithm. `quadprog` minimizes the L^2-norm problem. `quadprog` uses a good deal of memory, but solves quadratic programs to a high degree of precision (see Bottou and Lin [1]). For details, see the `svmtrain` function reference page.

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 $K(x,y)$ with the following property. There is a linear space S and a function φ mapping x to S such that

  $K(x,y) = <\varphi(x),\varphi(y)>$.

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

    $K(x,y) = (1 + <x,y>)^d$.
  - Radial basis function: For some positive number σ,

    $K(x,y) = \exp(-<(x-y),(x - y)>/(2\sigma^2))$.
  - Multilayer perceptron (neural network): For a positive number p_1 and a negative number p_2,

    $K(x,y) = \tanh(p_1<x,y> + p_2)$.

Note Not every set of p_1 and p_2 gives a valid reproducing 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, then 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" on page 1-42
- "Classifying New Data with an SVM Classifier" on page 1-43
- "Tuning an SVM Classifier" on page 1-43

Training an SVM Classifier

Train an SVM classifier with the `svmtrain` function. The most common syntax is:

```
SVMstruct = svmtrain(data,groups,'Kernel_Function','rbf');
```

The inputs are:

- `data` — Matrix of data points, where each row is one observation, and each column is one feature.
- `groups` — Column vector with each row corresponding to the value of the corresponding row in `data`. `groups` should have only two types of entries. So `groups` can have logical entries, or can be a double vector or cell array with two values.

- Kernel_Function — The default value of 'linear' separates the data by a hyperplane. The value 'rbf' uses a Gaussian radial basis function. Hsu, Chang, and Lin [4] suggest using 'rbf' as your first try.

The resulting structure, SVMstruct, 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 svmtrain reference page.

Classifying New Data with an SVM Classifier

Classify new data with the svmclassify function. The syntax for classifying new data with a SVMstruct structure is:

```
newClasses = svmclassify(SVMstruct,newData)
```

The resulting vector, newClasses, represents the classification of each row in newData.

Tuning an SVM Classifier

Hsu, Chang, and Lin [4] recommend tuning parameters of your classifier according to this scheme:

- Start with Kernel_Function set to 'rbf' and default parameters.
- Try different parameters for training, and check via cross validation to obtain the best parameters.

The most important parameters to try changing are:

- 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.
- rbf_sigma — One strategy is to try a geometric sequence of the RBF sigma parameter. For example, take 11 values, from 1e-5 to 1e5 by a factor of 10.

For the various parameter settings, try cross validating the resulting classifier. Use crossval with 5-way or the default 10-way cross validation.

After obtaining a reasonable initial parameter, you might want to 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. Alternatively, optimize your parameters with `fminsearch`, as shown in "SVM Classification with Cross Validation" on page 1-48.

Nonlinear Classifier with Gaussian Kernel

This example generates 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. It 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.

Note 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.

---

1 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*).

```
r = sqrt(rand(100,1)); % radius
t = 2*pi*rand(100,1); % angle
data1 = [r.*cos(t), r.*sin(t)]; % points
```

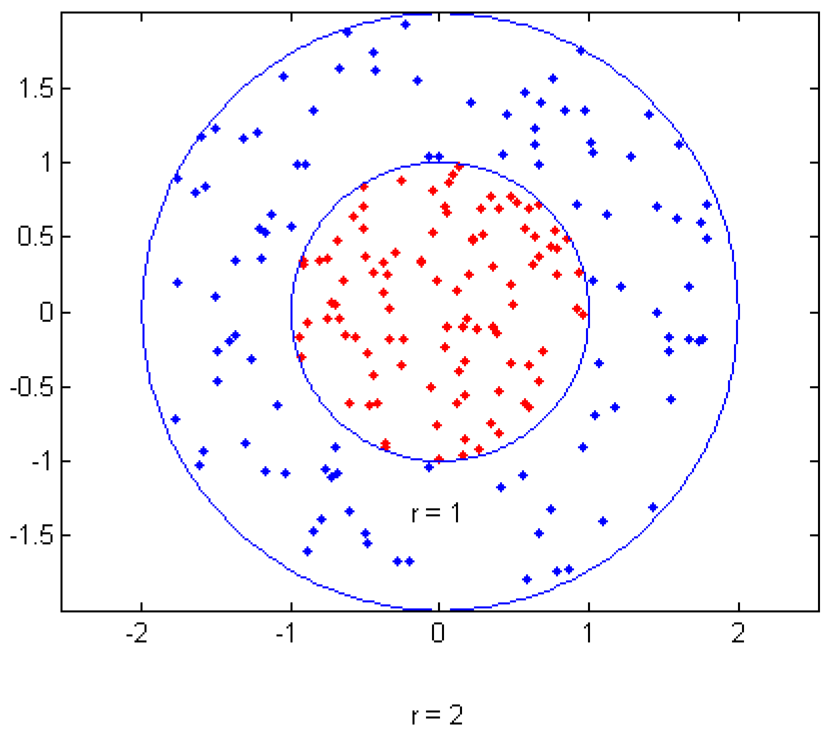
2 Generate 100 points uniformly distributed in the annulus. The radius is again proportional to a square root, this time a square root of the uniform distribution from 1 through 4.

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

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

```
plot(data1(:,1),data1(:,2),'r.')
```

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



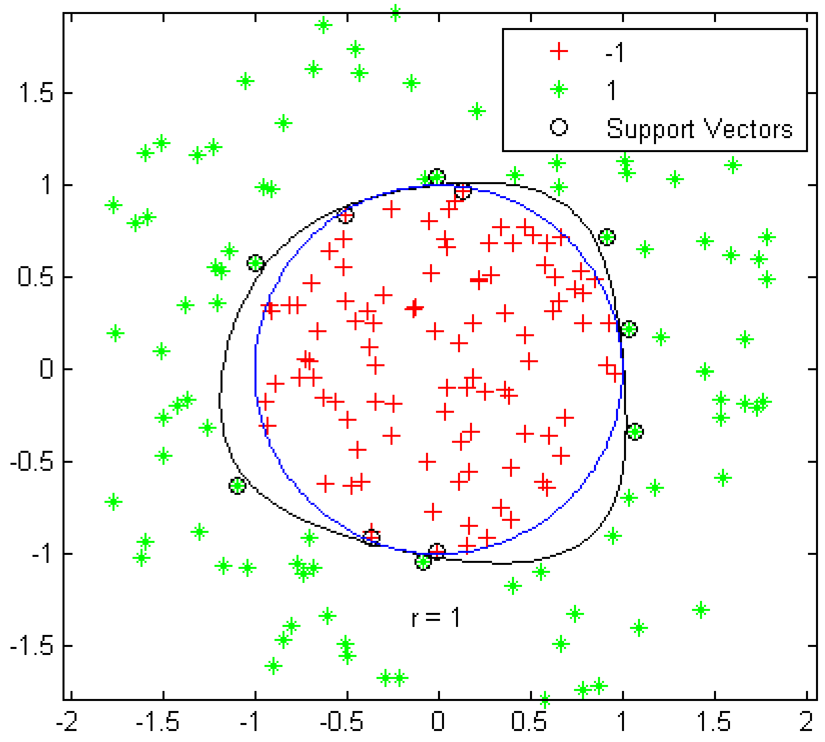
4 Put the data in one matrix, and make a vector of classifications:

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

5 Train an SVM classifier with:

- `Kernel_Function` set to `'rbf'`
- `boxconstraint` set to `Inf`

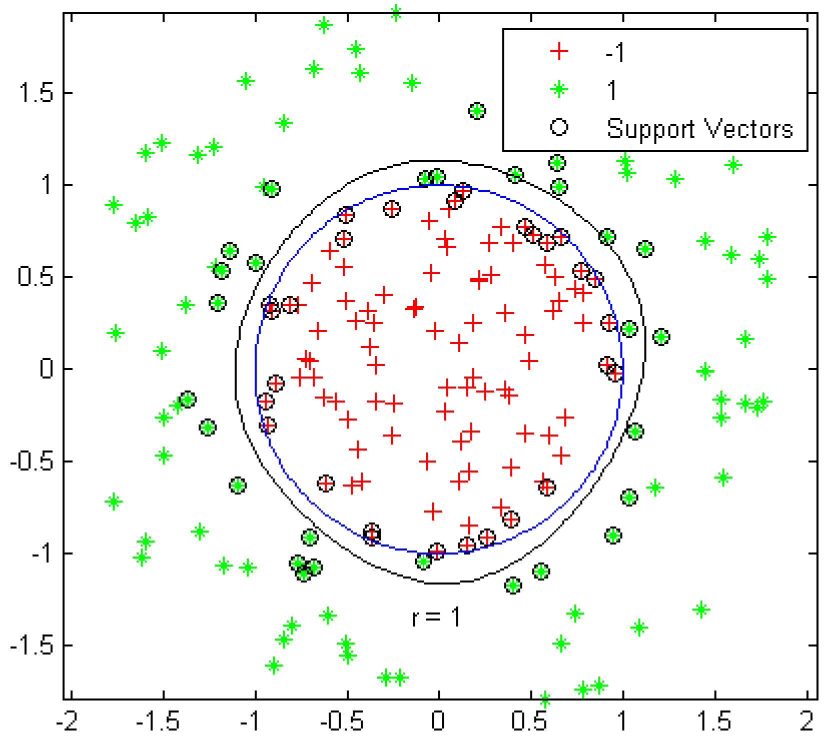
```
cl = svmtrain(data3,theclass,'Kernel_Function','rbf',...
    'boxconstraint',Inf,'showplot',true);
hold on
axis equal
ezpolar(@(x)1)
hold off
```



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

6 Training with the default parameters makes a more nearly circular classification boundary, but one that misclassifies some training data.

```
cl = svmtrain(data3,theclass,'Kernel_Function','rbf',...
    'showplot',true);
hold on
axis equal
ezpolar(@(x)1)
hold off
```



SVM Classification with Cross Validation

This example classifies points from a Gaussian mixture model. The model is described in Hastie, Tibshirani, and Friedman [3], page 17. It 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.

After generating 100 green and 100 red points, classify them using `svmtrain`, and tune the classification using cross validation.

To generate the points and classifier:

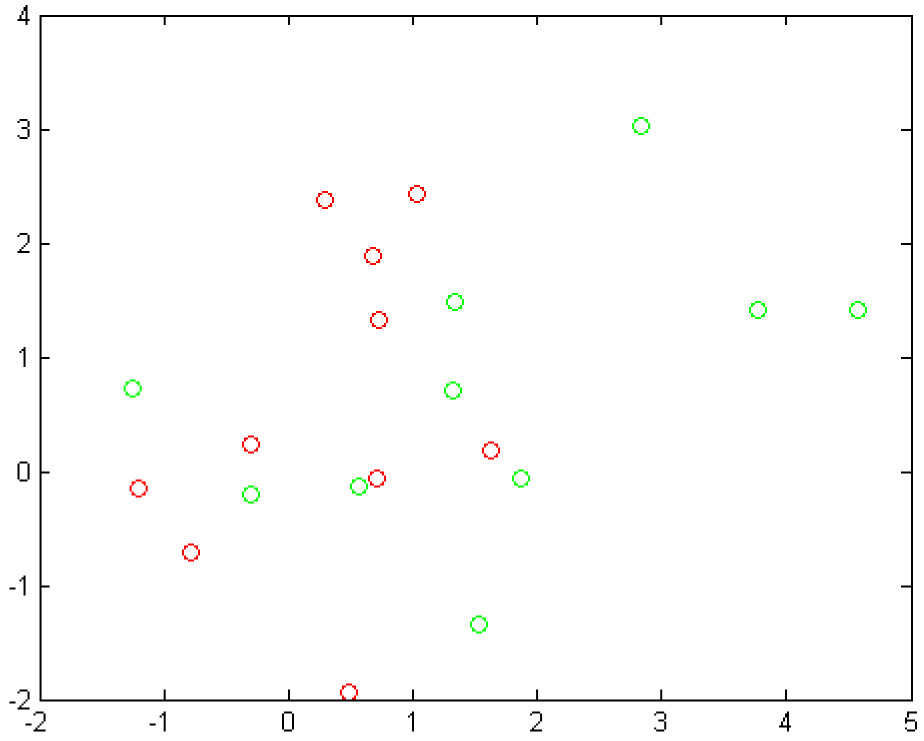
1 Generate the 10 base points for each class:

```
grnpop = mvnrnd([1,0],eye(2),10);
redpop = mvnrnd([0,1],eye(2),10);
```

2 View the base points:

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

Since many red base points are close to green base points, it is difficult to classify the data points.



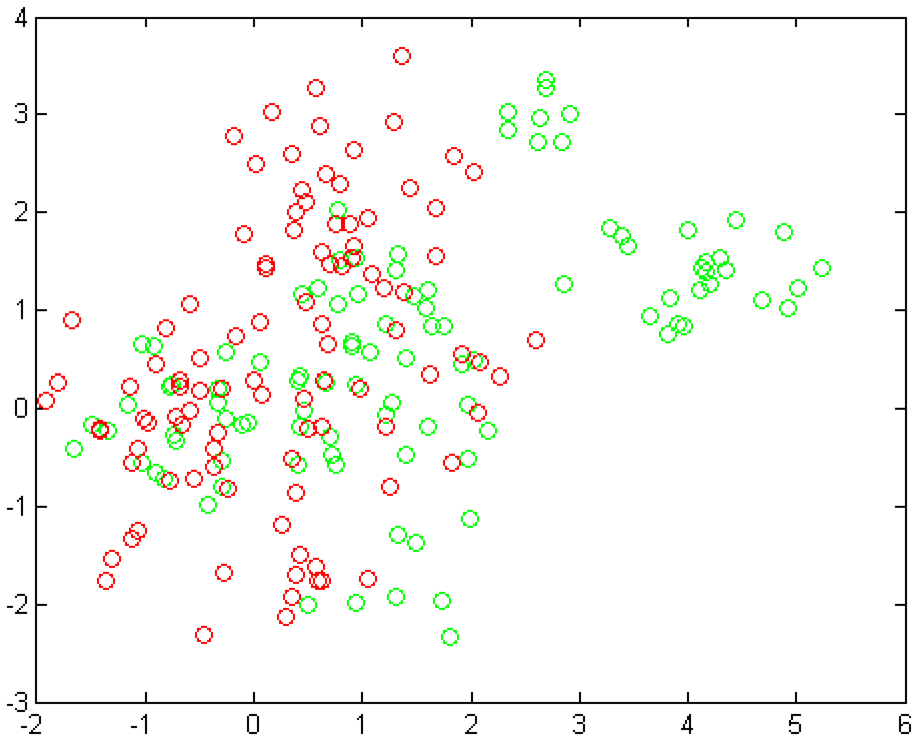
3 Generate the 100 data points of each class:

```
redpts = zeros(100,2);grnpts = redpts;
for i = 1:100
    grnpts(i,:) = mvnrnd(grnpop(randi(10),:),eye(2)*0.2);
    redpts(i,:) = mvnrnd(redpop(randi(10),:),eye(2)*0.2);
end
```

4 View the data points:

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

```
plot(redpts(:,1),redpts(:,2),'ro')
hold off
```

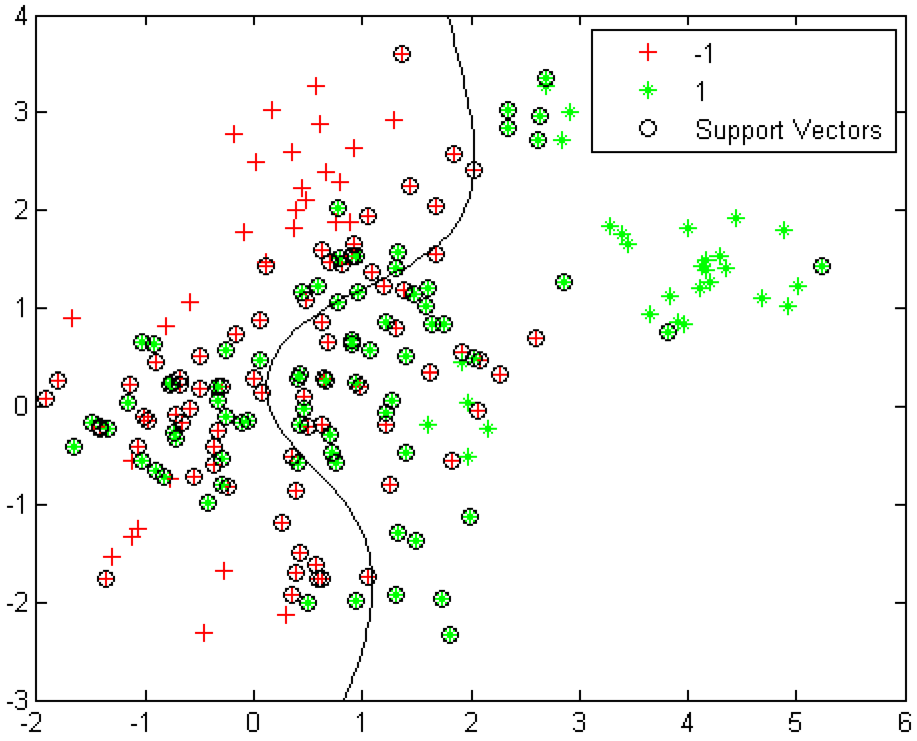


5 Put the data into one matrix, and make a vector `grp` that labels the class of each point:

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

6 Check the basic classification of all the data using the default parameters:

```
svmStruct = svmtrain(cdata,grp,'Kernel_Function','rbf',...
'showplot',true);
```



7 Write a function called `crossfun` to calculate the predicted classification `yfit` from a test vector `xtest`, when the SVM is trained on a sample `xtrain` that has classification `ytrain`. Since you want to find the best parameters `rbf_sigma` and `boxconstraint`, include those in the function.

```
function yfit = ...
    crossfun(xtrain,ytrain,xtest,rbf_sigma,boxconstraint)

% Train the model on xtrain, ytrain,
% and get predictions of class of xtest
svmStruct = svmtrain(xtrain,ytrain,'Kernel_Function','rbf',...
   'rbf_sigma',rbf_sigma,'boxconstraint',boxconstraint);
yfit = svmclassify(svmStruct,xtest);
```

8 Set up a partition for cross validation. This step causes the cross validation to be fixed. Without this step, the cross validation is random, so a minimization procedure can find a spurious local minimum.

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

9 Set up a function that takes an input `z=[rbf_sigma,boxconstraint]`, and returns the cross-validation value of `exp(z)`. The reason to take `exp(z)` is twofold:

- `rbf_sigma` and `boxconstraint` must be positive.
- You should look at points spaced approximately exponentially apart.

This function handle computes the cross validation at parameters `exp([rbf_sigma,boxconstraint])`:

```
minfn = @(z)crossval('mcr',cdata,grp,'Predfun', ...
    @(xtrain,ytrain,xtest)crossfun(xtrain,ytrain,...
    xtest,exp(z(1)),exp(z(2))),'partition',c);
```

10 Search for the best parameters `[rbf_sigma,boxconstraint]` with `fminsearch`, setting looser tolerances than the defaults.

Tip If you have a Global Optimization Toolbox license, use `patternsearch` for faster, more reliable minimization. Give bounds on the components of `z` to keep the optimization in a sensible region, such as [–5,5], and give a relatively loose `TolMesh` tolerance.

```
opts = optimset('TolX',5e-4,'TolFun',5e-4);
[searchmin fval] = fminsearch(minfn,randn(2,1),opts)

searchmin =
    0.9758
   -0.1569
```

```
fval =
    0.3350
```

The best parameters `[rbf_sigma;boxconstraint]` in this run are:

```
z = exp(searchmin)
z =
    2.6534
    0.8548
```

11 Since the result of `fminsearch` can be a local minimum, not a global minimum, try again with a different starting point to check that your result is meaningful:

```
[searchmin fval] = fminsearch(minfn,randn(2,1),opts)

searchmin =
    0.2778
    0.6395

fval =
    0.3100
```

The best parameters `[rbf_sigma;boxconstraint]` in this run are:

```
z = exp(searchmin)
z =
    1.3202
    1.8956
```

12 Try another search:

```
[searchmin fval] = fminsearch(minfn,randn(2,1),opts)

searchmin =
   -0.0749
    0.6085

fval =
    0.2850
```

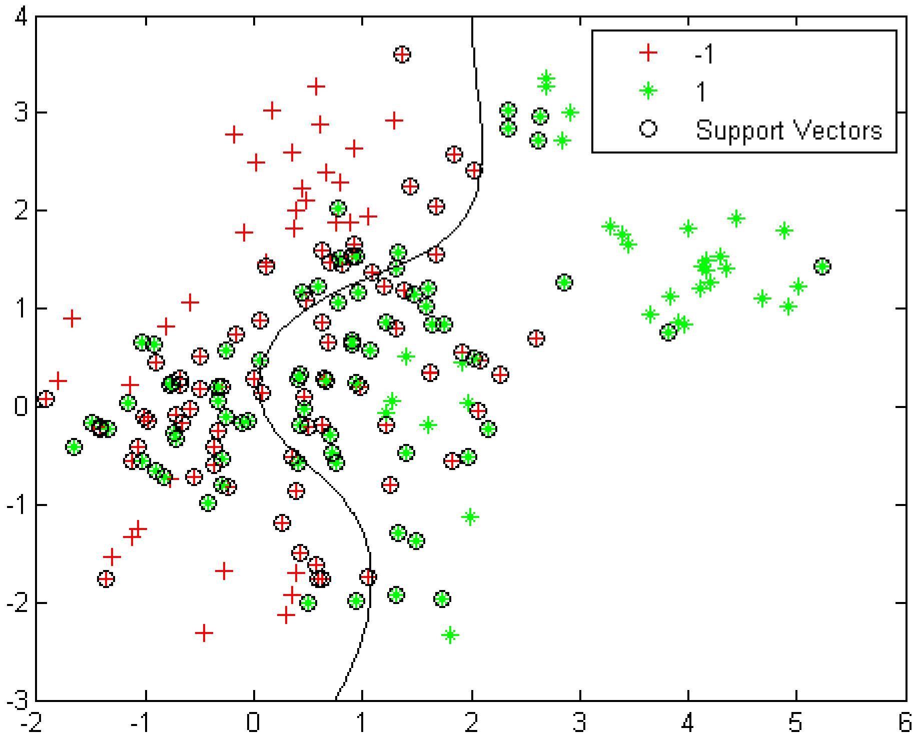
The third search obtains the lowest function value. The final parameters are:

```
z = exp(searchmin)
z =
    0.9278
    1.8376
```

The default parameters `[1,1]` are close to optimal for this data and partition.

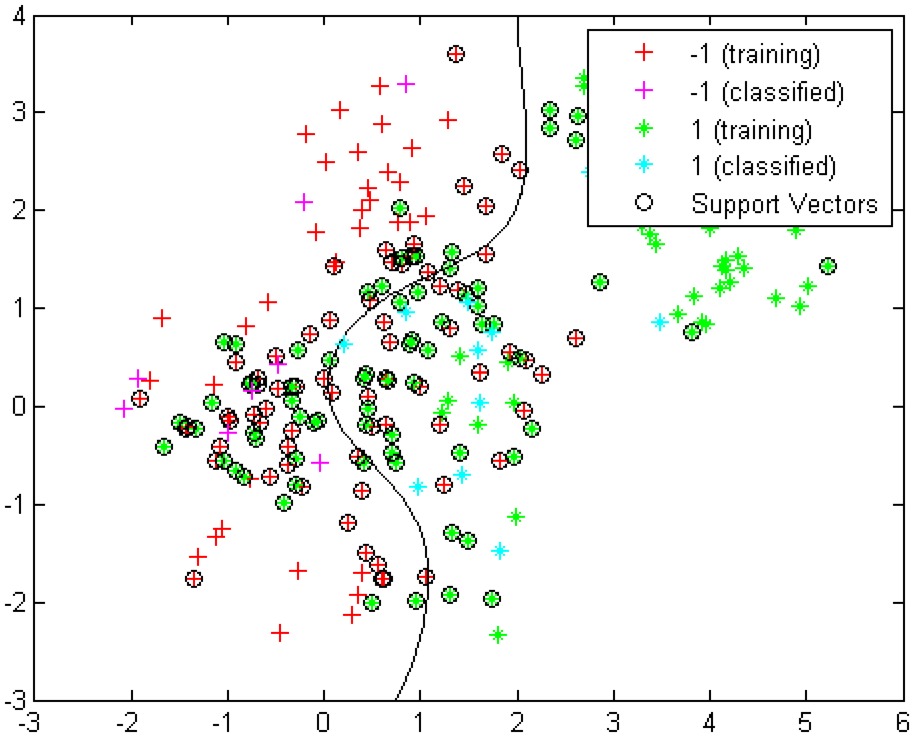
13 Use the `z` parameters to train a new SVM classifier:

```
svmStruct = svmtrain(cdata,grp,'Kernel_Function','rbf',...
'rbf_sigma',z(1),'boxconstraint',z(2),'showplot',true);
```



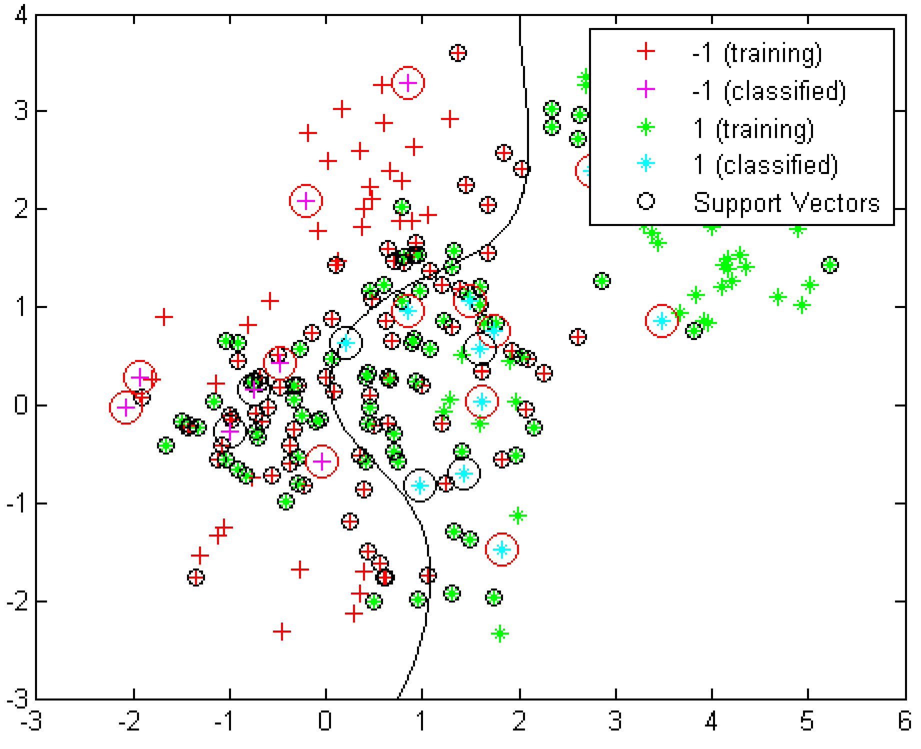
14 Generate and classify some new data points:

```
grnobj = gmdistribution(grnpop,.2*eye(2));
redobj = gmdistribution(redpop,.2*eye(2));

newData = random(grnobj,10);
newData = [newData;random(redobj,10)];
grpData = ones(20,1);
grpData(11:20) = -1; % red = -1

v = svmclassify(svmStruct,newData,'showplot',true);
```



15 See which new data points are correctly classified. Circle the correctly classified points in red, and the incorrectly classified points in black.

```
mydiff = (v == grpData); % classified correctly
hold on
for ii = mydiff % plot red circles around correct pts
    plot(newData(ii,1),newData(ii,2),'ro','MarkerSize',12)
end

for ii = not(mydiff) % plot black circles around incorrect pts
    plot(newData(ii,1),newData(ii,2),'ko','MarkerSize',12)
end
hold off
```



References

[1] Bottou, L., and Chih-Jen Lin. *Support Vector Machine Solvers*. Available at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.64.4209&rep=rep1&type=pdf`.

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

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

[4] Hsu, Chih-Wei, Chih-Chung Chang, and Chih-Jen Lin. *A Practical Guide to Support Vector Classification*. Available at `http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf`.