Statistical Learning

Lecture 9a - Support Vector Machines

ANU - RSFAS

Last Updated: Tue May 3 12:42:00 2022

Support Vector Machines

- Here we approach the two-class classification problem in a direct way:
  - We try and find a plane that separates the classes in feature space.
- If we cannot, we get creative in two ways:
  - We soften what we mean by “separates”.
  - We enrich and enlarge the feature space so that separation is possible.

What is a Hyperplane?

- A hyperplane in p dimensions is a flat (does not have to go through the origin) subspace of dimension $p-1$.
- In 2 dimensions, the equation for a hyperplane has the form:

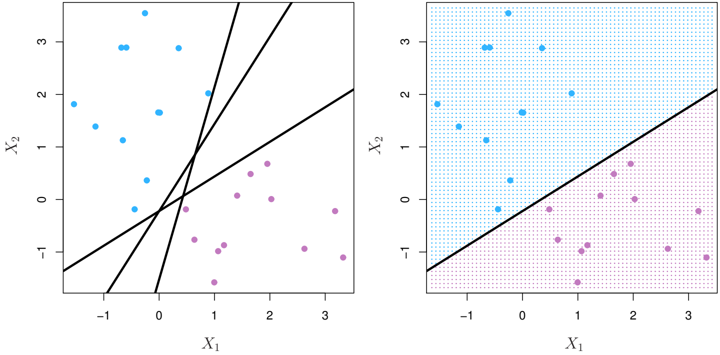
$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 = 0$$

- So for $p = 2$ dimensions a hyperplane is a line.
- More generally we have:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p = 0$$

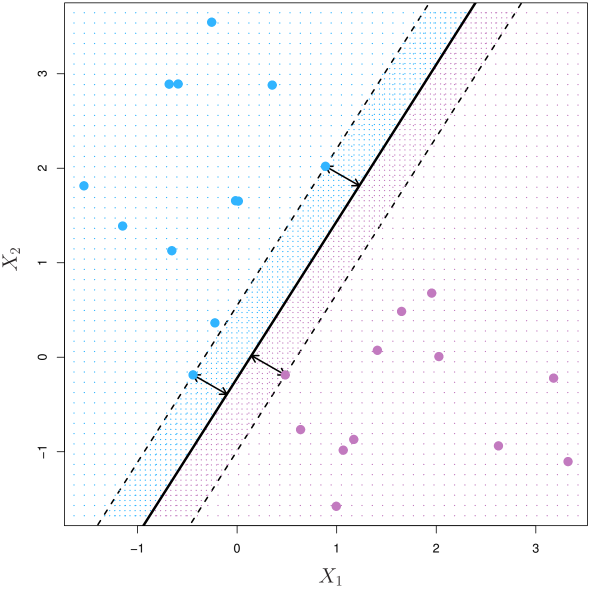
- If $\beta_0 = 0$, the hyperplane goes through the origin, otherwise not.
- The vector $\boldsymbol{\beta}' = (\beta_1, \beta_2, \ldots, \beta_p)$ is called the normal vector — it points in a direction orthogonal to the surface of a hyperplane.

Separating Hyperplanes



- If $f(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2$:
  - Then $f(X) > 0$ for points on one side and $f(X) < 0$ for points on the otherside.
- Code points: $Y_i = +1$ for blue, and $Y_i = -1$ for mauve, then $Y_i \times f(X) > 0 \;\; \forall \; i$.
- $f(X) = 0$ defines a separating hyperplane.

Maximal Margin Classifier

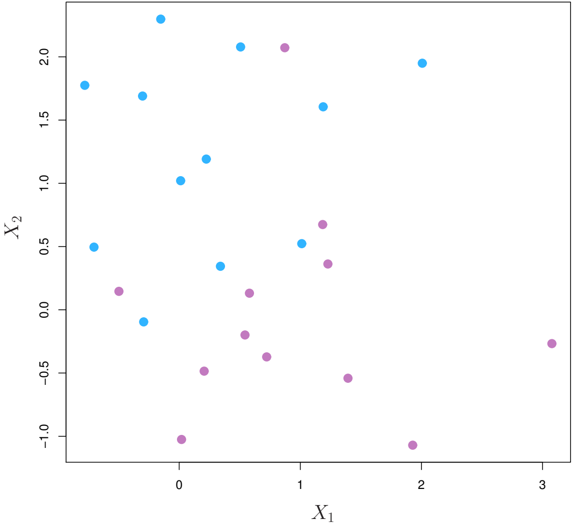


- Constrained optimization problem:

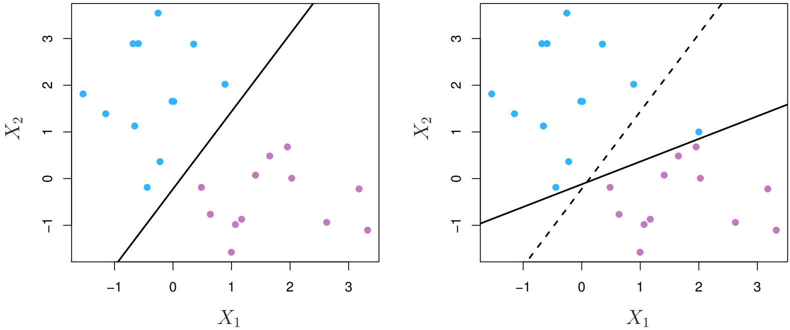
$$
\begin{aligned}
\underset{\beta_0,\beta_1,\ldots,\beta_p}{\text{maximize}} \quad & M \\
\text{Subject to} \quad & \sum_{j=1}^{p} \beta_j^2 = 1 \\
& y_i(\beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \cdots + \beta_p x_{i,p}) \geq M \\
\forall \quad & i = 1, \ldots, n
\end{aligned}
$$

- This can be rephrased as a convex quadratic program, and solved efficiently. The function svm() in package e1071 solves this problem.

Non-Separable Data



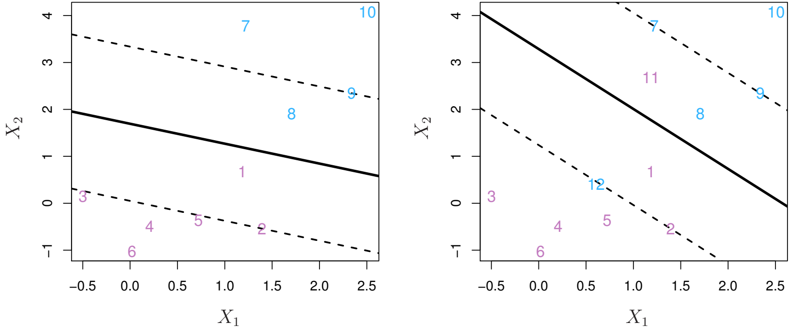
Noisy Data



- Sometimes the data are separable, but noisy. This can lead to a poor solution for the maximal-margin classifier.
- The support vector classifier maximizes a soft margin.

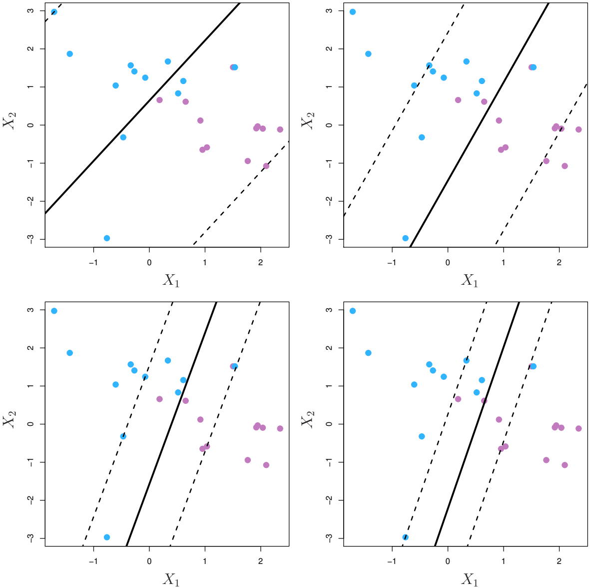
Support Vector Classifier

$$
\begin{aligned}
\underset{\beta_0,\beta_1,\ldots,\beta_p,\epsilon_1,\epsilon_2,\ldots,\epsilon_n}{\text{maximize}} \quad & M \\
\text{Subject to} \quad & \sum_{j=1}^{p} \beta_j^2 = 1 \\
& y_i(\beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \cdots + \beta_p x_{i,p}) \geq M(1 - \epsilon_i) \\
\epsilon_i \geq 0 \quad & \sum_{i=1}^{n} \epsilon_i \leq C
\end{aligned}
$$

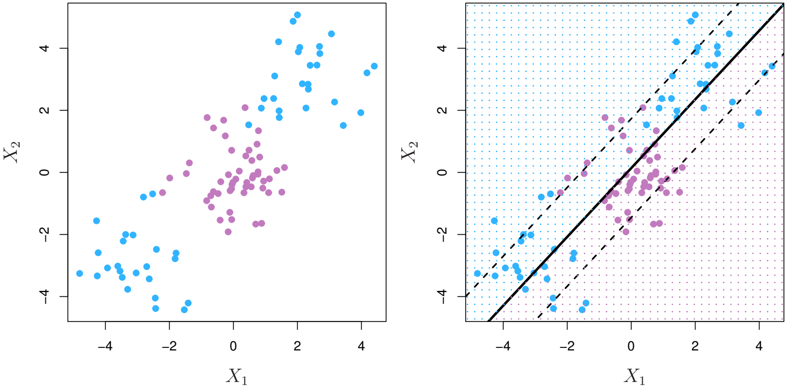


4
3
2
1
0
−1
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
X_1
X_2
1
2
3
4
5
6
7
8
9
10
11
12

C is a Regularization Parameter



A Linear Boundary Can Fail



- Sometime a linear boundary simply won't work, no matter what value of C.

As Usual - Feature Expansion

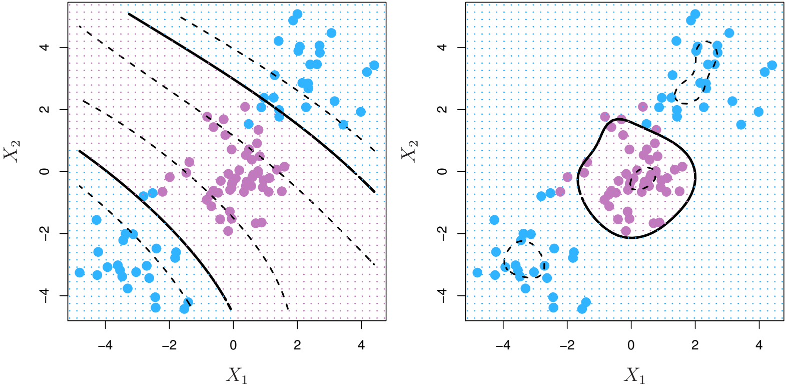
- Enlarge the space of features by including transformations:

$$X_1^2, X_1^3, X_1 X_2, X_1 X_2^2, \ldots$$

- Fit a support-vector classifier in the enlarged space.
- This results in non-linear decision boundaries in the original space.
- Eg.

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1^2 + \beta_4 X_2^2 + \beta_5 X_1 X_2 = 0$$

Cubic Polynomials



$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1^2 + \beta_4 X_2^2 + \beta_5 X_1 X_2 + \beta_6 X_1^3 + \beta_7 X_2^3 + \beta_8 X_1^2 X_2 + \beta_9 X_1 X_2^2 = 0$$

Nonlinearities and Kernels

- Polynomials (especially high-dimensional ones) get wild rather fast.
- There is a more elegant and controlled way to introduce nonlinearities in support-vector classifiers — through the use of kernels.
- Before we discuss these, we must understand the role of inner products in support-vector classifiers.

Inner Products and Support Vectors

$$\langle x_i, x_{i'} \rangle = \sum_{j=1}^{p} x_{ij} x_{i'j} \quad - \text{ inner product}$$

- The linear support vector classifier can be represented as:

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i \langle x, x_i \rangle \quad - \text{ n parameters}$$

- To estimate the parameters: $\alpha_1, \ldots, \alpha_n$ and β_0, all we need are the $\binom{n}{2}$ inner products $x_i, x_{i'}$.

- It turns out that only the support vectors are non-zero, so most $\hat{\alpha}_i$ are zero.

$$f(x) = \beta_0 + \sum_{i \in \mathcal{S}} \hat{\alpha}_i \langle x, x_i \rangle \quad - \text{ n parameters}$$

- $\mathcal{S}$ is the support set of indices i such that $\hat{\alpha}_i > 0$.

Kernels and Support Vector Machines

- If we can compute inner-products between observations, we can fit a SV classifier. Can be quite abstract!

- Some special kernel functions can do this for us:

$$K(x_i, x_{i'}) = \left(1 + \sum_{j=1}^{p} x_{ij} x_{i'j}\right)^d$$

computes the inner-products needed for d dimensional polynomials — $\binom{p+d}{d}$ basis functions!

- Try this for $p = 2$ and $d = 2$. What do you get?

- The solution has the form:

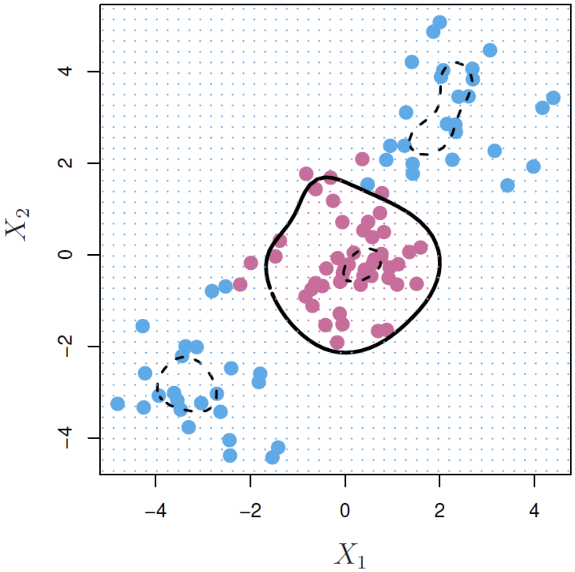
$$f(x) = \beta_0 + \sum_{i \in \mathcal{S}} \hat{\alpha}_i K(x_i, x_{i'})$$

Radial Kernel

$$K(x_i, x_{i'}) = exp\left(-\gamma \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2\right)$$

$$f(x) = \beta_0 + \sum_{i \in \mathcal{S}} \hat{\alpha}_i K(x_i, x_{i'})$$

- Implicit feature space: very high dimensional.
- Controls variance by squashing down most dimensions severely.


X_1
X_2
−4
−2
0
2
4
−4
−2
0
2
4

SVMs: More than 2 classes?

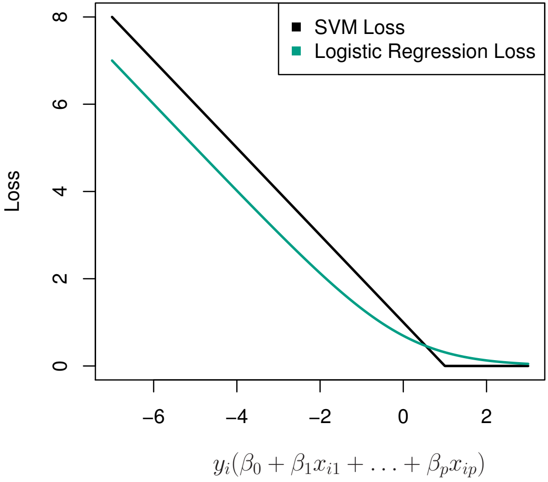
- The SVM as defined works for $K = 2$ classes. What do we do if we have $K > 2$ classes?
- OVA One versus All:
  - Fit K different 2-class SVM classifiers $\hat{f}_k(x)$ for $k = 1, \ldots, K$; each class versus the rest.
  - Classify x^* to the class for which $\hat{f}_k(x^*)$ is the largest.
- OVO One versus One:
  - Fit all $\binom{K}{2}$ pairwise classifiers $\hat{f}_{kl}(x)$.
  - Classify x^* to the class that wins the most pairwise competitions.
- Which to choose? If K is not too large, use OVO.

Support Vector versus Logistic Regression?

- We can rephrase the SV optimization problem as:

$$\underset{\beta_0,\beta_1,\ldots,\beta_p}{\text{minimize}} \left\{ \sum_{i=1}^{n} \max\left[0, 1 - y_i f(x_i)\right] + \lambda \sum_{j=1}^{p} \beta_j^2 \right\}$$

- This has the form — loss plus penalty.
- The loss is known as the hinge loss.
- Very similar to "loss" in logistic regression (negative log-likelihood).


SVM Loss
Logistic Regression Loss
Loss
8
6
4
2
0
−6
−4
−2
0
2
$y_i(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip})$

Which to Use: SVM or Logistic Regression

- When classes are (nearly) separable, SVM does better than LR. So does LDA.
- When not, LR (with ridge penalty) and SVM very similar.
- If you wish to estimate probabilities, LR is the choice.
- For nonlinear boundaries, kernel SVMs are popular. Can use kernels with LR and LDA as well, but computations are more expensive - also more difficult to interpret!

Some Fun

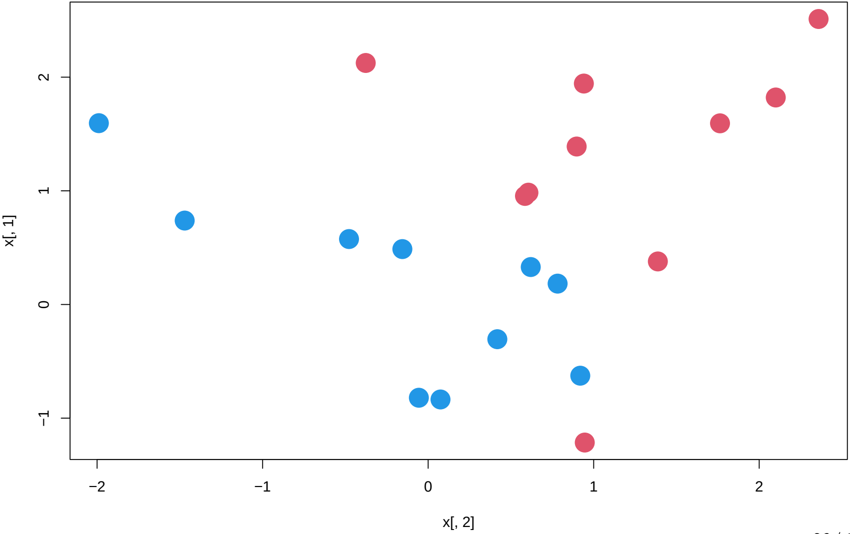
- Let's generate some data.

```
set.seed (1)
x <- matrix(rnorm (20*2), ncol =2)
y <- c(rep (-1,10), rep (1 ,10))
x[y==1,] <- x[y==1,] + 1

##
plot(x[,2], x[,1], col=(3-y), pch=16, cex=3)
```

Some Fun

- Let's generate some data.



- We can see that the two classes are not linearly separable. Let's try to classify based on **support vector machines**.
- A cost argument allows us to specify the cost of a violation to the margin. When the cost argument is small, then the margins will be wide and many support vectors will be on the margin or will violate the margin.
- When the cost argument is large, then the margins will be narrow and there will be few support vectors on the margin or violating the margin.

```r
dat <- data.frame(x=x, y=as.factor(y))
library(e1071)
svmfit <- svm(y ~ ., data=dat, kernel="linear",
              cost=10, scale=FALSE)
```

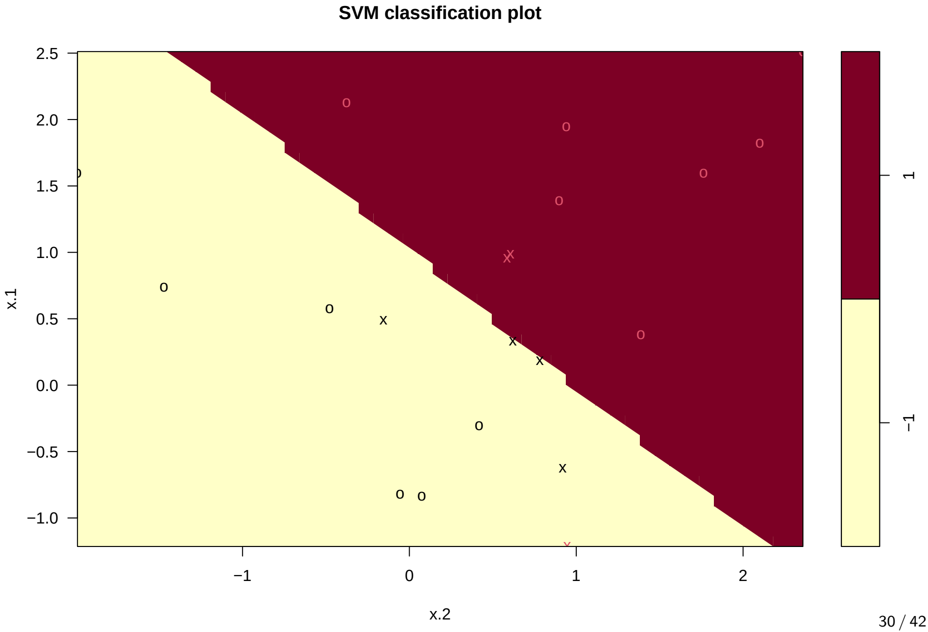
- The argument scale=FALSE tells the svm() function not to scale each feature to have mean zero or standard deviation one; depending on the application, one might prefer to use scale=TRUE.

- Let's get some summary information.

```
summary(svmfit)
```

```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 10, scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  10
##
## Number of Support Vectors:  7
##
##  ( 4 3 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

```r
plot(svmfit, dat)
```



- The support vectors (cases)

```
svmfit$index
```

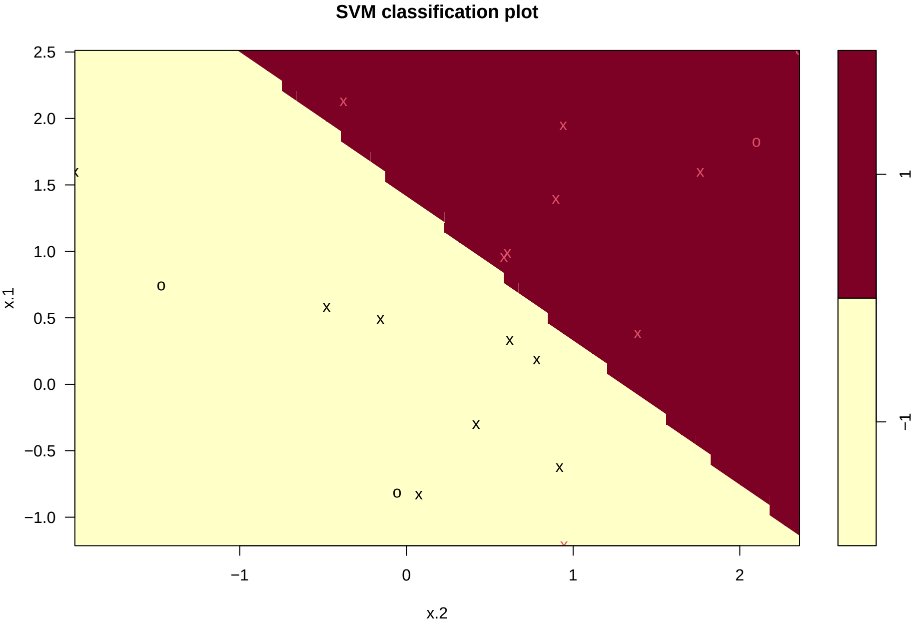
```
## [1]  1  2  5  7 14 16 17
```

```
dat[svmfit$index,]
```

```
##           x.1        x.2  y
## 1  -0.6264538  0.9189774 -1
## 2   0.1836433  0.7821363 -1
## 5   0.3295078  0.6198257 -1
## 7   0.4874291 -0.1557955 -1
## 14 -1.2146999  0.9461950  1
## 16  0.9550664  0.5850054  1
## 17  0.9838097  0.6057100  1
```

- Let's change the cost. Now the **cost** is cheap so we have more lee-way with our **error budget**.

```
svmfit <- svm(y ~ ., data=dat,
              kernel="linear", cost=0.10, scale=FALSE)
plot(svmfit , dat)
```


SVM classification plot
x.1
x.2
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1
0
1
2
1
−1

```
summary(svmfit)
```

```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 0.1, scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.1
##
## Number of Support Vectors:  16
##
##  ( 8 8 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

- Let's use 10-fold cross-validation to **tune** the cost.

```r
set.seed(1)
tune.out <- tune(svm, y ~ ., data=dat, kernel ="linear",
                    ranges=list(cost=c(0.001, 0.01, 0.1, 1, 5, 10, 100) ))
bestmod <- tune.out$best.model
summary(tune.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##   0.1
##
## - best performance: 0.05
##
## - Detailed performance results:
##    cost error dispersion
## 1 1e-03  0.55  0.4377975
## 2 1e-02  0.55  0.4377975
## 3 1e-01  0.05  0.1581139
## 4 1e+00  0.15  0.2415229
## 5 5e+00  0.15  0.2415229
## 6 1e+01  0.15  0.2415229
## 7 1e+02  0.15  0.2415229
```

- Now let's consider some predictions.

```
xtest <- matrix(rnorm (20*2), ncol=2)
ytest <- sample(c(-1,1) , 20, rep=TRUE)
xtest[ytest==1,] = xtest[ytest==1,] + 1
testdat <- data.frame(x=xtest, y=as.factor(ytest))



##
ypred <- predict(bestmod, testdat)
tab <- table(predict=ypred, truth=testdat$y); tab
```

```
##          truth
## predict -1 1
##      -1  9 1
##      1   2 8
```

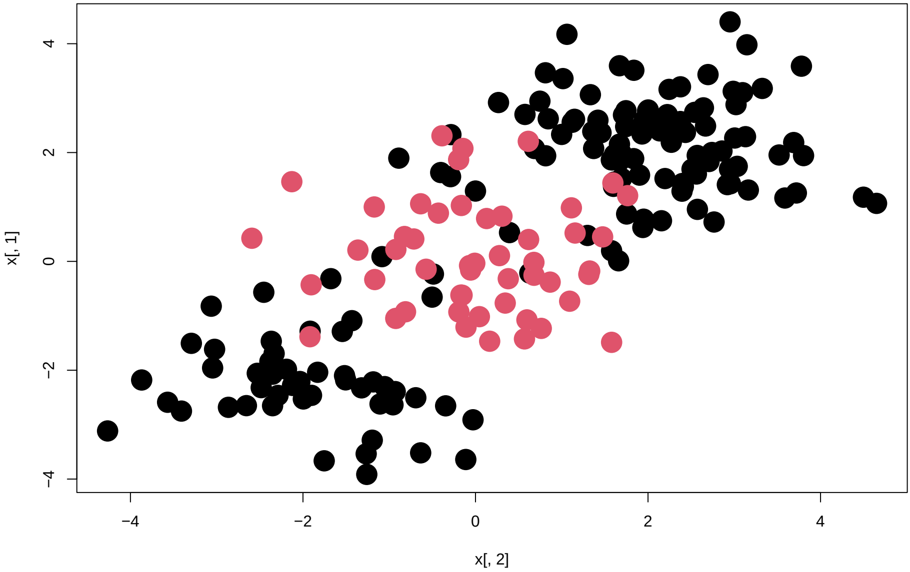
```
sum(diag(tab))/sum(tab)
```

```
## [1] 0.85
```

More Data

```
set.seed(1)
x <- matrix(rnorm (200*2), ncol=2)
x[1:100,] <- x[1:100,] + 2
x[101:150,] <-  x[101:150,] - 2
y <- c(rep(1,150), rep (2,50))
dat <- data.frame(x=x,y=as.factor(y))

##
plot(x[,2], x[,1], col=y, pch=16, cex=3)
```

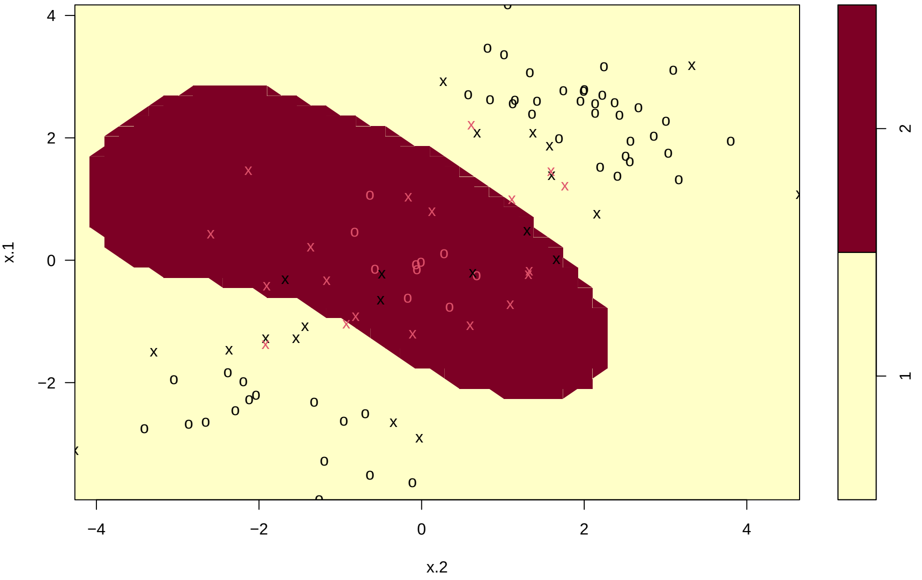


4
2
0
−2
−4
x[, 1]
−4
−2
0
2
4
x[, 2]

- Let's randomly split into training and testing.

```
set.seed(1)
train <- sample(200, 100)
svmfit <- svm(y ~ ., data=dat[train,],
              kernel="radial", gamma=1, cost=1)
plot(svmfit, dat[train,])
```

SVM classification plot



- We now have **two** tuning parameters.

```
set.seed (1)
tune.out <- tune(svm, y ~., data=dat[train,], kernel="radial",
      ranges=list(cost=c(0.1, 1, 10, 100, 1000), gamma=c(0.5,1,2,3,4) ))
summary(tune.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost gamma
##     1   0.5
##
## - best performance: 0.12
##
## - Detailed performance results:
##     cost gamma error dispersion
## 1  1e-01   0.5  0.28 0.15491933
## 2  1e+00   0.5  0.12 0.07888106
## 3  1e+01   0.5  0.15 0.10801234
## 4  1e+02   0.5  0.17 0.11595018
## 5  1e+03   0.5  0.23 0.14944341
## 6  1e-01   1.0  0.25 0.13540064
## 7  1e+00   1.0  0.14 0.09660918
## 8  1e+01   1.0  0.16 0.10749677
## 9  1e+02   1.0  0.21 0.15238839
## 10 1e+03   1.0  0.20 0.14142136
## 11 1e-01   2.0  0.28 0.14757296
## 12 1e+00   2.0  0.15 0.10801234
## 13 1e+01   2.0  0.19 0.15238839
## 14 1e+02   2.0  0.18 0.14757296
## 15 1e+03   2.0  0.23 0.12516656
## 16 1e-01   3.0  0.28 0.15491933
```

- Let's do some prediction.

```
tab <- table(true=dat[-train,"y"],
             pred=predict(tune.out$best.model,
                          newx=dat[-train,]))
tab
```

```
##      pred
## true  1  2
##     1 52 27
##     2 17  4
```

```
sum(diag(tab))/sum(tab)
```

```
## [1] 0.56
```