Midterm Project

Brandon Amaral, Monte Davityan, Nicholas Lombardo, Hongkai Lu

10/19/2022

1 ISLR Chapter 9 Summary

9.1 Maximal Margin Classifier

This section is focus on definition and introduction the concept of an optimal separating hyperplane.

9.1.1 What Is a Hyperplane? A line is a hyperplane in a two dimensions space; a plane is a hyperplane in a three dimensions space; a flat affine subspace of $p-1$ dimension subspace is a hyperplane in a p-dimensional space. The mathematical definition of a hyperplane is defined by the equation as follow:

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

We have below sense for a p-dimensional hyperplane.

1: A point X lies on the hyperplane if $ X = (X_{1},X_{2},...,X_{p})^T$ in p-dimensional space satisfies.

2: A point X lies to one side of the hyperplane if

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

3: A point X lies to other side of the hyperplane if

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

Thus, we can think of the hyperplane as dividing p-dimensional space into two halves.

9.1.2 Classification Using a Separating Hyperplane First, suppose that we have an $n \times p$ data matrix $\mathbf{X}$ that consists of n training observations in a p-dimensinal space. The training observations fall into two groups, where $y_1, \ldots, y_n \in \{-1, 1\}$, -1 represents one class, and 1 the other class. Second, try to construct a separating hyperplane which is a hyperplane that separates the training observations perfectly according to their class labels. That separating hyperplane has the below property:

1:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p > 0 \quad if \quad y_i = 1,$$

2:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p < 0 \quad if \quad y_i = -1,$$

Next, we use the constructed separating hyperplane to construct a very natural classifier: a test observation is assigned a class depending on which side of the hyperplane it is located. The classifier is based on a separating hyperplane which leads to a linear decision boundary.

9.1.3 The Maximal Margin Classifier The goal is to construct a classifier based upon a separating hyperplane. Instinctively, the maximal margin hyperplane is the separating hyperplane that is farthest from the training observations. Then, we can classify a test observation based on which side of the maximal margin hyperplane it lies, which is the maximal margin classifier.

9.1.4 Construction of the Maximal Margin Classifier Suppose we have a set of n training observations $x_1, \ldots, x_n \in \mathbb{R}^p$ and associated class labels $y_1, \ldots, y_n \in \{-1, 1\}$, then the maximal margin hyperplane is to solve the optimization problem as follows:

$$(9.9): \quad \underset{\beta_0, \beta_1, \ldots, \beta_p, M}{maximize} M$$

$$(9.10): \quad subject \quad to \quad \sum_{j=1}^{p} \beta_j^2 = 1$$

$$(9.11): \quad y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip}) \geq M \quad \forall i = 1, \ldots, n)$$

First, use the inequality (9.11) to constrain each observation to the correct side of the hyperplane. Second, both (9.10) and (9.11) ensure that each observation is on the correct side of the hyperplane and at least a distance M from the hyperplane. Last, M represents the margin of our hyperplane, and the optimization problem chooses $\beta_0, \beta_1, \ldots, \beta_p$ to maximize M.

9.1.5 The Non-separable Case If a separating hyperplane exists, we can use the maximal margin classifier to perform classification. But in many cases, there is no separating hyperplane. The next section focuses on how to generalize the maximal margin classifier to the non-separable case, which is known as the support vector classifier.

9.2.1 Overview of the Support Vector Classifier Suppose we have two classes observations. Instead of perfectly separating the two classes, the support vector classifier is more interested in the greater robustness to individual observations and better classification of most of the training observations. In other words, the support vector classifier allows some observations to lie on the incorrect side of the margin or even the incorrect side of the hyperplane.

9.2.2 Details of the Support Vector Classifier Similar to "Construction of the Maximal Margin Classifier", the Support Vector Classifier is solves the optimization problem as follows:

$$(9.12): \quad \underset{\beta_0, \beta_1, \ldots, \beta_p, \epsilon_1, \ldots, \epsilon_n, M}{maximize} M$$

$$(9.13): \quad subject\ to \sum_{j=1}^{p} \beta_j^2 = 1$$

$$(9.14): \quad y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip}) \geq M(1 - \epsilon_i) \quad \forall i = 1, \ldots, n)$$

$$(9.15): \epsilon_i \geq 0, \quad \sum_{i=1}^{n} \epsilon_i \leq C, where\ C\ is\ a\ nonnegative\ tuning\ parameter$$

First, we use the slack variable ϵ_i to locate the ith observation relative to the hyperplane and relative to the margin as follows:

(1): If $\epsilon_i = 0$ then the ith observation is on the correct side of the margin

(2): If $\epsilon_i \geq 0$ then the ith observation is on the wrong side of the margin

(3): If $\epsilon_i \geq 1$ then it is on the wrong side of the hyperplane.

Then, we use the C in (9.15) to be the controller for the bias-variance trade-off of the statistical learning technique. According to the text, we know "When C is small, we seek narrow margins that are rarely violated; this amounts to a classifier that is highly fit to the data, which may have low bias but high variance. On the other hand, when C is larger, the margin is wider and we allow more violations to it; this amounts to fitting the data less hard and obtaining a classifier that is potentially more biased but may have lower variance."

Finally, the observations that lie directly on the margin or on the wrong side of the margin for their class are support vectors, which affect and result the support vector classifier.

9.3 Support Vector Machines

9.3.1 Classification with Non-Linear Decision Boundaries In the two-class setting, the support vector classifier is a natural approach for classification if the boundary between the two classes is linear. For non-linear class boundaries, we could enlarge the feature space using quadratic, cubic, and even higher-order polynomial functions of the predictors to fit a support vector classifier.

Suppose we have a set of n training observations $x_1, \ldots, x_n \in \mathbb{R}^p$ and associated class labels $y_1, \ldots, y_n \in \{-1, 1\}$, we use $2p$ features to fit (9.12) - (9.15)

$$X_1, X_1^2, X_2, X_2^2, \ldots, X_p, X_p^2.$$

Then, we have

$$(9.16): \underset{\beta_0, \beta_{11}, \beta_{12}, \ldots, \beta_{p1}, \beta_{p2}, \epsilon_1, \ldots, \epsilon_n, M}{maximize} M$$

$$subject\ to\ y_i(\beta_0 + \sum_{j=1}^{p} \beta_{j1} x_{ij}^2 + \sum_{j=1}^{p} \beta_{j2} x_{ij}^2) \geq M(1 - \epsilon_i)$$

$$\sum_{i=1}^{n} \epsilon_i \leq C, \quad \epsilon_i \geq 0, \quad \sum_{j=1}^{p} \sum_{k=1}^{2} \beta_{jk}^2 = 1$$

9.3.2 The Support Vector Machine By using kernels, the extension of the support vector classifier that results from enlarging the feature space in a specific way is known as the support vector machine (SVM). Suppose we have a support vector classifier that can be represented as

$$f(x) = \beta_0 + \sum_{i \in S} \alpha_i K\langle x, x_i \rangle$$

(1): If we take a linear kernel

$$K\langle x, x_i \rangle = \sum_{j=1}^{p} x_{ij} x_{i'j}$$

, it is a linear support vector classifier.

(2): If we take a polynomial kernel of degree d

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

, it is a non-linear support vector classifier.

(3): If we take a radial kernel

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

, it is a non-linear support vector classifier.

9.3.3 An Application to the Heart Disease Data In this section, we fit LDA and SVM to the heart disease training data. By using ROC curves, we can see the SVM using a polynomial kernel of degree $d = 1$ is slightly superior to LDA. Also, Figure 9.10 displays ROC curves for SVMs using a radial kernel with three values of γ, which indicates a more flexible method will often produce lower training error rates but may not improve performance on test data.

9.4 SVMs with More than Two Classes

The goal is to extend SVMs to the more general case where we have some arbitrary number of classes. The text introduces two methods which are the one-versus-one and one-versus-all approaches.

9.4.1 One-Versus-One Classification Suppose there are $K > 2$ classes, we would like to perform classification using a one-versus-one or all-pairs approach constructs $\binom{k}{2}$ SVMs. First, classify a test observation using each of the $\binom{k}{2}$ classifiers. Second, tally the number of times that the test observation is assigned to each of the K classes. Last, assign the test observation to the class to which it was most frequently assigned in these $\binom{k}{2}$ pairwise classifications to obtain the final classification.

9.4.2 One-Versus-All Classification In the one-versus-all approach, when we fit K SVMs, we compare one of the K classes to the remaining $K-1$ classes each time.

9.5 Relationship to Logistic Regression

First, below, we have ridge regression and the lasso function:

$$L(X, y, \beta) = \sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2$$

Then, we introduce the loss function of SVM:

$$L(X, y, \beta) = \sum_{i=1}^{n} \max[0, 1 - y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip})]$$

The hinge loss function of SVM is closely related to the loss function used in logistic regression. Due to the similarities, the results of logistic regression and the support vector classifier are often similar. SVMs tend to behave better than logistic regression if the classes are well separated; whereas logistic regression is often preferred if there are more overlapping regimes.

2 Murphy Section 17.3 Summary

17.3.1 Large margin classifiers

Suppose we have a binary classifier with the form $h(x) = sign(f(x))$, then the decision boundary is given by a linear function as follow:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + w_0$$

The goal is to obtain the large margin classifier, which described below:

$$\underset{\boldsymbol{w}, w_0}{max} \frac{1}{||w||} \underset{n=1}{\overset{N_{\mathcal{D}}}{\min}} [\hat{y_n}(\boldsymbol{w}^T\boldsymbol{x} + w_0)]$$

After some transformation, the final objective becomes

$$\underset{\boldsymbol{w}, w_0}{min} \frac{1}{2}||w||^2 \quad s.t. \quad \hat{y_n}(\boldsymbol{w}^T\boldsymbol{x} + w_0) \geq 1, \; n = 1 : N_{\mathcal{D}}$$

17.3.2 The dual problem

The final objective of the large margin classifier is a standard quadratic problem, and we can convert it to a dual problem in convex optimization. The generalized Lagrangian yields the following after plugging $\hat{w} = \sum_{n=1}^{N} \hat{\alpha_n}\tilde{y_n}x_n$ and $0 = \sum_{n=1}^{N} \hat{\alpha_n}\tilde{y_n}$

$$\mathcal{L}(\hat{\boldsymbol{w}}, \hat{w_0}, \alpha) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i\alpha_j\tilde{y_i}\tilde{y_j}x_i^T x_j + \sum_{n=1}^{N} \alpha_n$$

We use the following formula to perform prediction:

$$f(x; \hat{\boldsymbol{w}}, \hat{w_0}) = \hat{\boldsymbol{w}}^T\boldsymbol{x} + \hat{w_0} = \sum_{n \in S} \alpha_n\tilde{y_n}x_n^T x + \hat{w_0}$$

17.3.3 Soft margin classifiers

Slack variables are introduced if the data is not linearly separable. We use the soft margin constraints to replace the ard constraints, then we have the new objective as following:

$$\min_{\boldsymbol{w},w_0,\boldsymbol{\xi}} \frac{1}{2}||w||^2 + C\sum_{n=1}^{N_{\mathcal{D}}} \xi_n \quad s.t. \quad \xi_n \geq 0,\ \hat{y_n}(\boldsymbol{w}^T\boldsymbol{x} + w_0) \geq 1 - \xi_n$$

Correspondingly, we have the Lagrangian for the soft margin classifier after optimizing out $\boldsymbol{w}, w_0,\ and\ \boldsymbol{\xi}$,

$$\mathcal{L}(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i\alpha_j\tilde{y_i}\tilde{y_j}x_i^T x_j$$

17.3.4 The kernel trick

By rewriting the prediction function, we have

$$f(x) = \hat{\boldsymbol{w}}^T\boldsymbol{x} + \hat{w_0} = \sum_{n\in S} \alpha_n\tilde{y_n}x_n^T x + \hat{w_0} = \sum_{n\in S} \alpha_n\tilde{y_n}\mathcal{K}(x_n, x) + \hat{w_0}$$

The bias term is the following after kernelizing

$$\hat{w_0} = \frac{1}{|S|}\sum_{n\in S}(\tilde{y_i} - \hat{\alpha}_j\tilde{y_j}\mathcal{K}(x_j, x_i))$$

17.3.5 Converting SVM outputs into probabilities

The goal is to convert the output of an SVM to a probability. We have the following probability

$$p(y = 1|\boldsymbol{x}, \boldsymbol{\theta}) = \sigma(af(\boldsymbol{x}) + b)$$

where a and b can be estimated by maximum likelihood on a separate validation set.

17.3.6 Connection with logistic regression

For the data points that lie on the correct side of the decision boundary, we have $\xi_n = 0$; for the others, we have $\xi_n = 1 - \tilde{y}_n f(\boldsymbol{x}_n)$. Then, we can rewrite the objective equation of the soft margin classifiers as follows:

$$\mathcal{L}(w) = \sum_{i=1}^{N} \ell_{hinge}(\tilde{y_n}, f(x_n)) + \lambda||w||^2$$

where $\lambda = (2C)^{-1}$ and ℓ_{hinge} is the hinge loss function as below:

$$\ell_{hinge}(\tilde{y_n}, \eta) = max(0, 1 - \tilde{y_n}\eta)$$

Compared to the logistic regression which optimizes:

$$\mathcal{L}(w) = \sum_{i=1}^{N} \ell_{ll}(\tilde{y_n}, f(x_n)) + \lambda||w||^2$$

Where the log loss function is as below:

$$\ell_{ll}(\tilde{y_n}, \eta) = log(1 + e^{-\tilde{y_n}\eta})$$

There are two major difference between the log loss and hinge loss:

1: The hinge loss is piece-wise linear, so we cannot use regular gradient methods to optimize it.

2: The hinge loss has a region where it is strictly 0; this results in sparse estimates

17.3.7 Multi-class classification with SVMs

Similar to ISLR, for multi-class classification, Murphy introduces two methods which are the one-versus-the-rest and one-versus-one approaches.

17.3.8 How to choose the regularizer C

Typically, we apply cross-validation to determinate parameter C. By developing a path following algorithm in the spirit of lars, we are able to choose C efficiently.

17.3.9 Kernel ridge regression

The kernel ridge regression is as follows:

$$f(x; w) = k^T(\boldsymbol{K} + \lambda \boldsymbol{I}_N)^{-1} y$$

where $k = [\mathcal{K}(x, x_1), \ldots, \mathcal{K}(x, x_N)]$

17.3.10 SVMs for regression

The support vector machine regression is described below:

$$f(x) = \hat{w}_0 + \sum_{n:\alpha>0} \alpha_n \mathcal{K}(x_n, x)$$

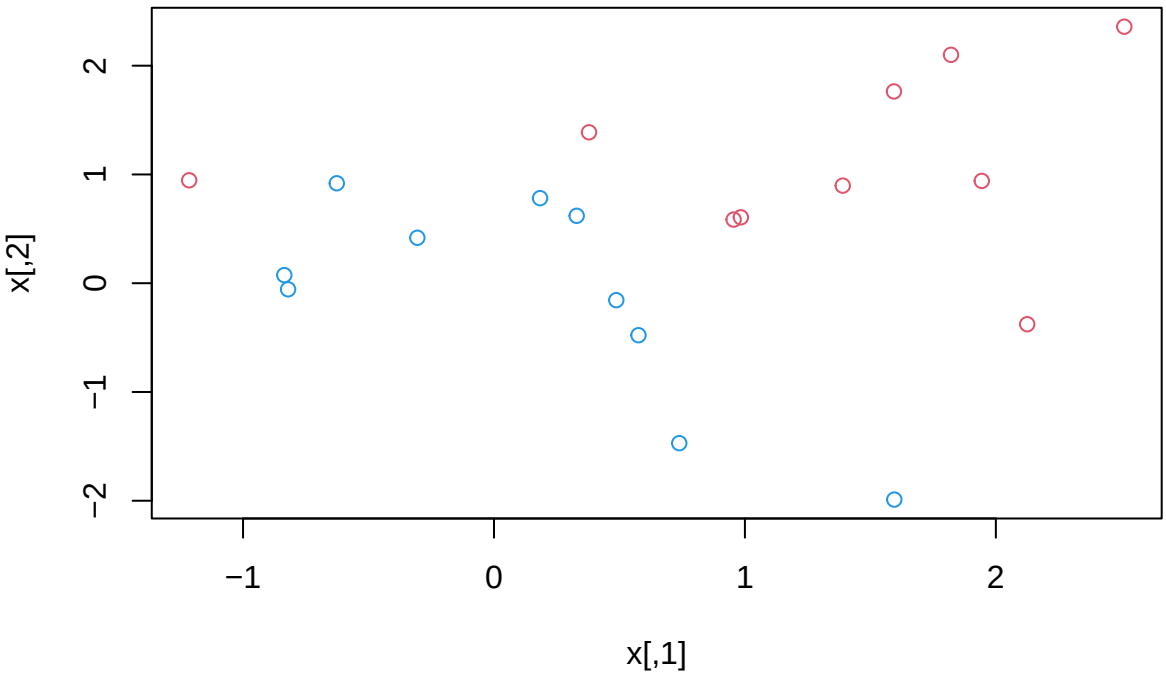
3 ISLR Section 9.6 Lab Report

In this lab, we demonstrate the support vector classifier and the support vector machine (SVM).

9.6.1 Support Vector Classifier

To start, we generate observations belonging to two classes and visually inspect whether they are linearly separable.

```
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, col = (3 - y))
```

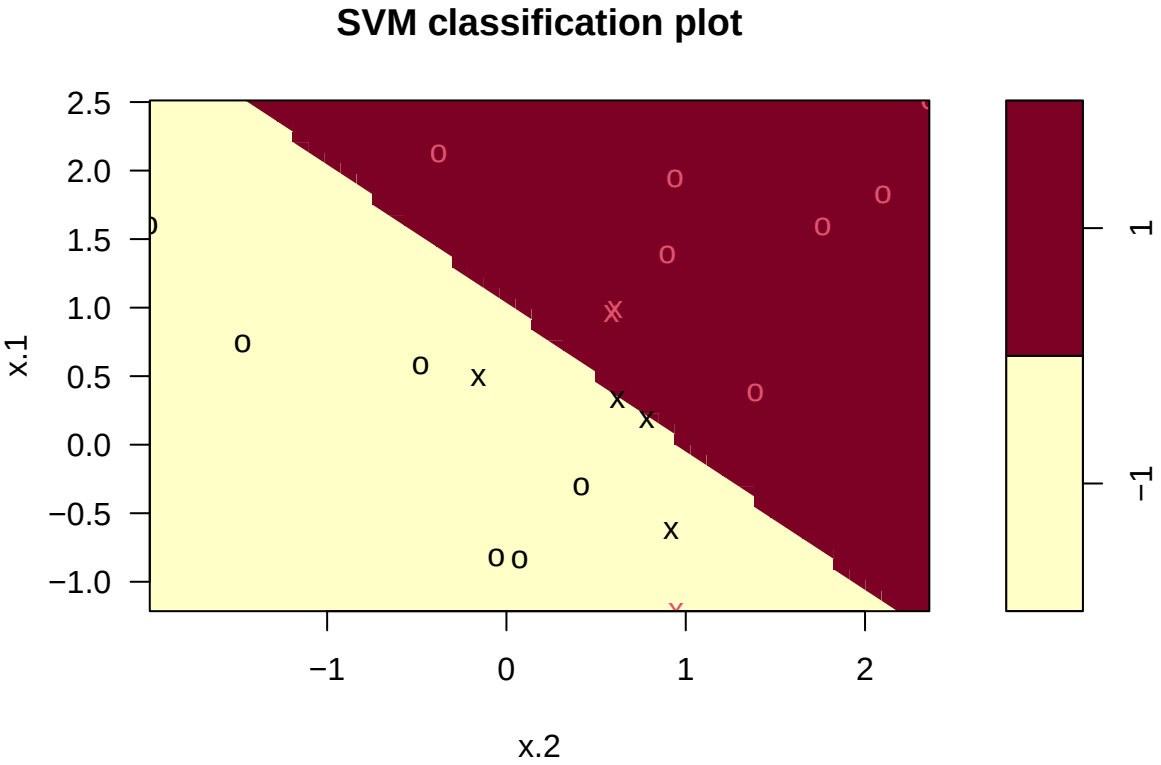


We see they are not linearly separable, so a maximal margin classifier is not usable. Let's set up a support vector classifier. To do so, we need to encode the response as a factor variable. We'll create a data frame with the response coded as a factor. Then we will use the svmfit() function from the e1071 library.

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

Although we chose not to scale the data here, we may choose to do so in some contexts. Let's plot the support vector classifier.

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



We see that the support vectors are plotted as crosses while the other observations are plotted as circles. We can determine the identities of the support vectors using:

```
svmfit$index
```

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

If we want other information about our support vector classifier fit, the summary() command helps us:

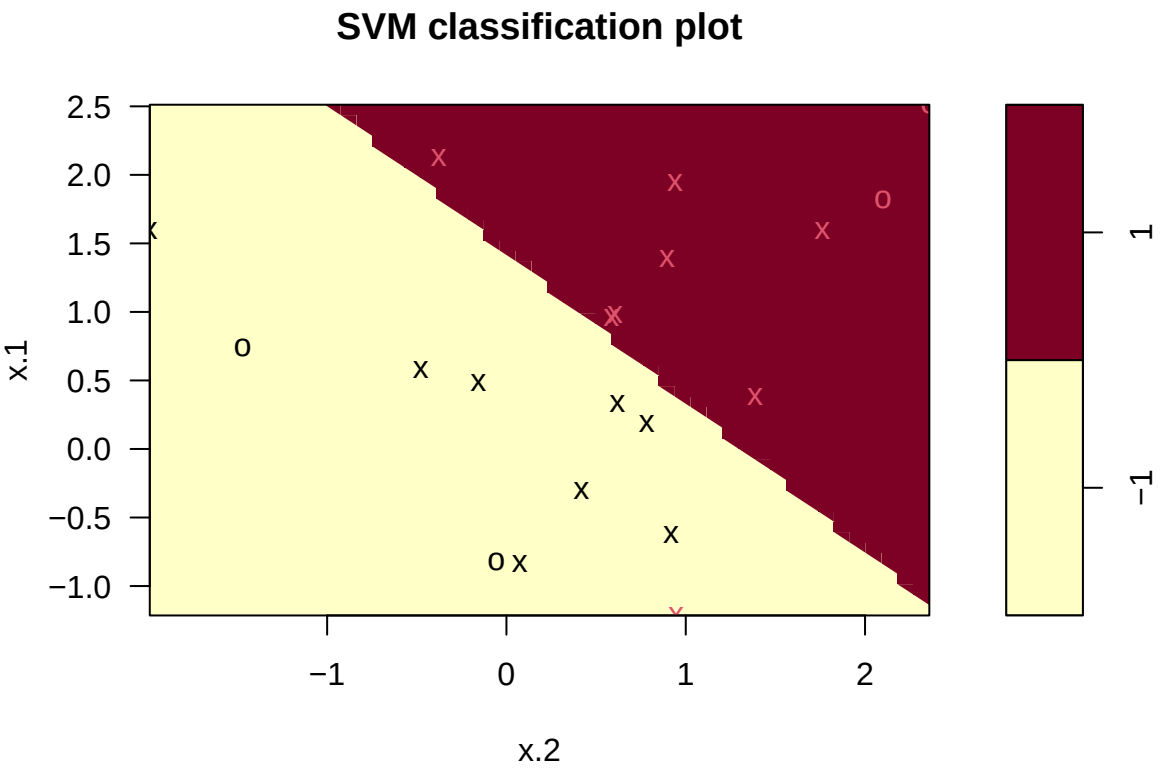
```
summary(svmfit)
```

```
##
## 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
```

Let's see how our fit is affected by using a smaller value of "cost", which reduces the penalty of margin

violations and widens the margins.

```
svmfit <- svm(y ~ ., data = dat, kernel = "linear", cost = 0.1, scale = FALSE)
plot(svmfit, dat)
```




```
svmfit$index
```

```
##  [1]  1  2  3  4  5  7  9 10 12 13 14 15 16 17 18 20
```

Now we have a greater number of support vectors (which outnumber the other observations!) and we see the margins are noticeably wider. The svm() function does not explicitly output the coefficients of the linear decision boundary or width of the margin, so we can't verify these quickly. We can use the tune() function of the e1071 library to perform cross-validation. Let's do this with a range of values for the "cost" parameter.

```
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)))
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
```

The lowest cross-validation error rate was obtained when cost = 0.1. Conveniently, the best model is stored in the tune() function, accessed here:

```
bestmod <- tune.out$best.model
summary(bestmod)
```

```
##
## Call:
## best.tune(method = svm, train.x = y ~ ., data = dat, ranges = list(cost = c(0.001,
##     0.01, 0.1, 1, 5, 10, 100)), kernel = "linear")
##
##
## 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 the predict() function to predict class labels for a set of test observations. We'll starting by generating a test data set again.

```
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))
```

Now we will use the best model from our cross-validation to made predictions on the test observations.

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

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

Using that previously optimized value of cost (0.1), 17 of the test observations were correctly classified. Let's try using a cost of 0.01 instead.

```
svmfit <- svm(y ~ ., data = dat, kernel = "linear", cost = 0.01, scale = FALSE)
ypred <- predict(svmfit, testdat)
```

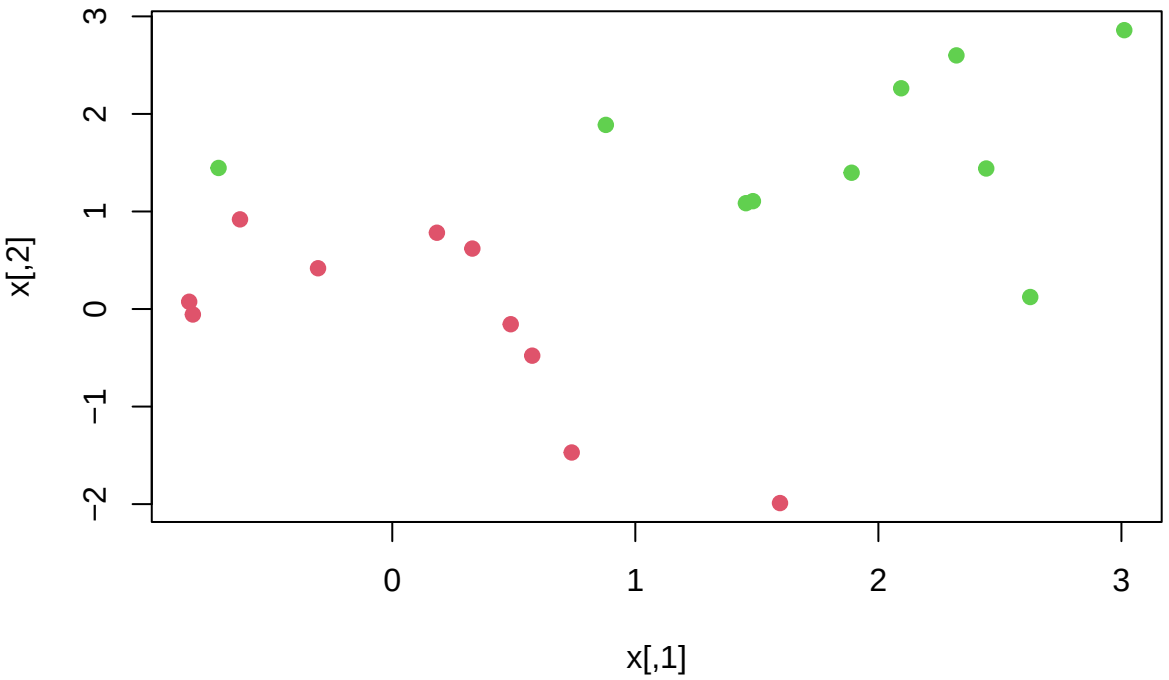
```
table(predict = ypred, truth = testdat$y)
```

```
##        truth
## predict -1  1
##      -1 11  6
##      1   0  3
```

Now there are 3 additional misclassified observations. This is in line with our cross-validation which found a higher error rate with the cost value of 0.01 versus 0.1.

So far, we have only considered data sets where the two classes were not linearly separable. Now let's explore a case where the classes are separable by a hyperplane. We generate such a data set:

```
x[y == 1, ] <- x[y == 1, ] + 0.5
plot(x, col = (y+5)/2, pch = 19)
```



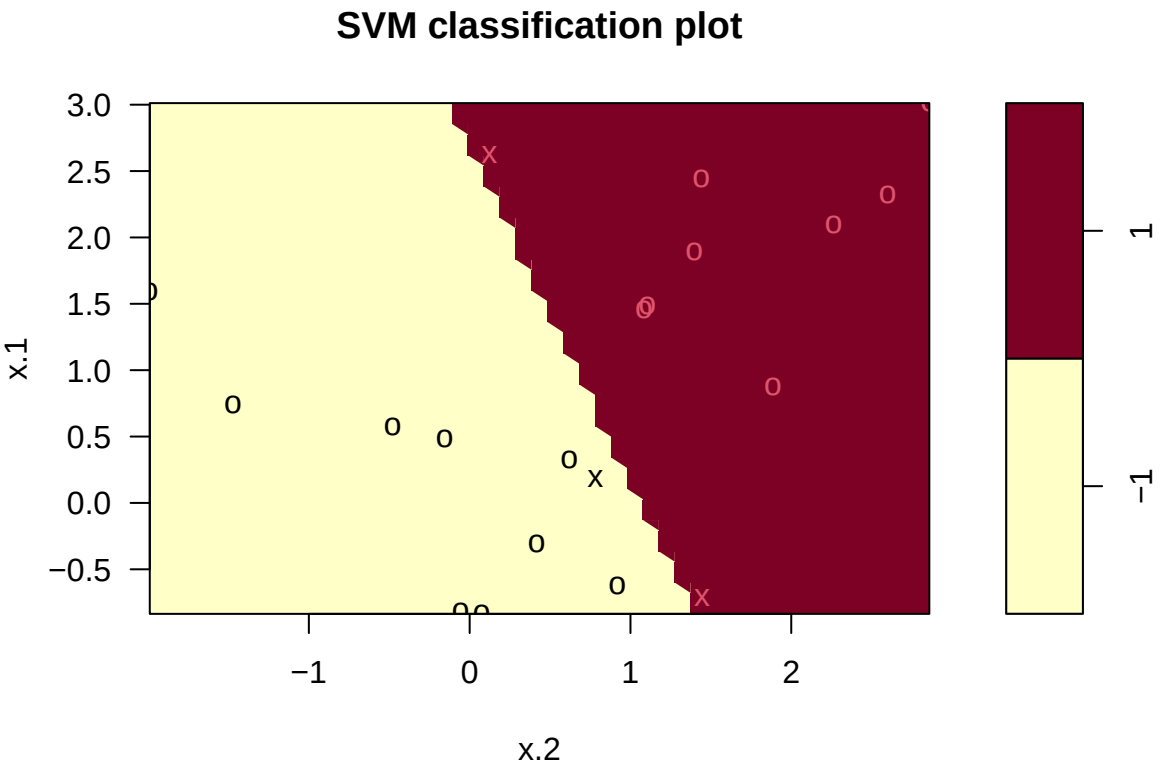
We see that the observations are linearly separable, but just barely! In order to produce strictly separated classes (as in a maximal margin classifier), we can set the "cost" value very high.

```
dat <- data.frame(x = x, y = as.factor(y))
svmfit <- svm(y ~ ., data = dat, kernel = "linear", cost = 1e5)
summary(svmfit)
```

```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 1e+05)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  1e+05
##
## Number of Support Vectors:  3
##
```

```
##  ( 1 2 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

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



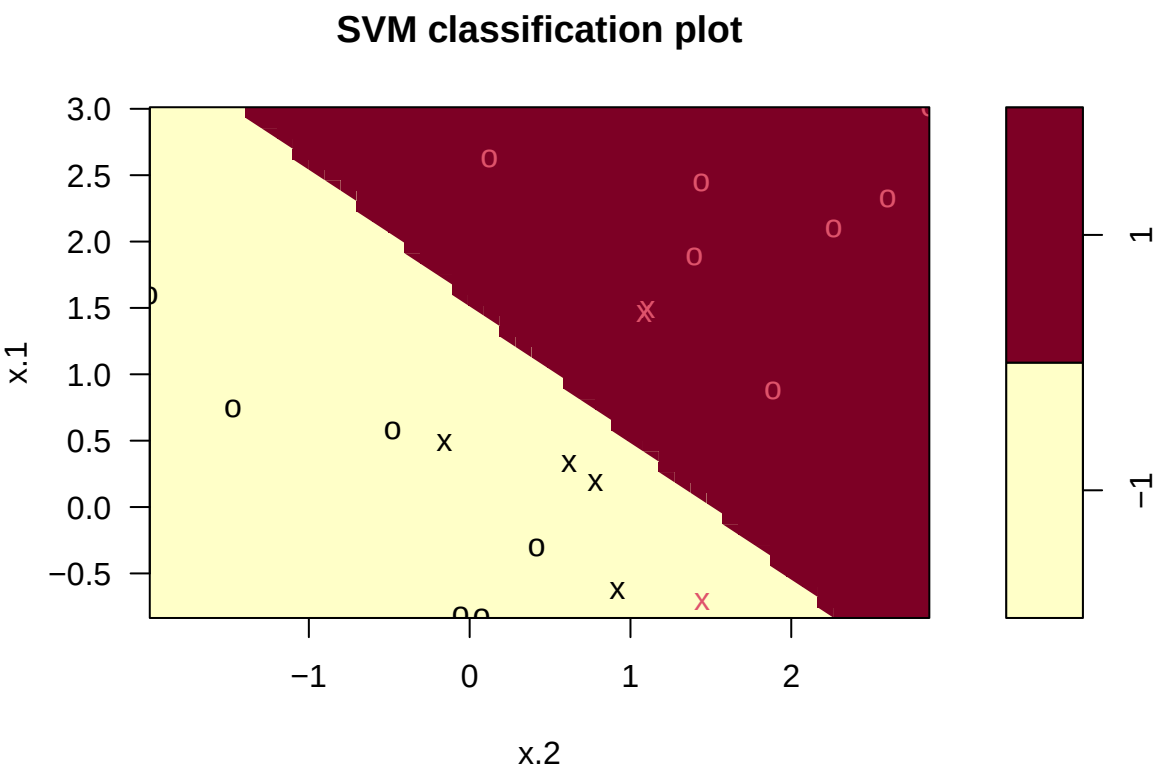
The support vector classifier made no errors on the training data, and the minimum number of support vectors (3) was used. Since we see there are non-support vectors (graphed as circles) near the decision boundary, we know the margin was quite narrow. Because of this, we can expect the model to perform less well on test data. Let's make one more model, this time with a smaller value than cost.

```
svmfit <- svm(y ~ ., data = dat, kernel = "linear", cost = 1)
summary(svmfit)
```

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

```
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

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



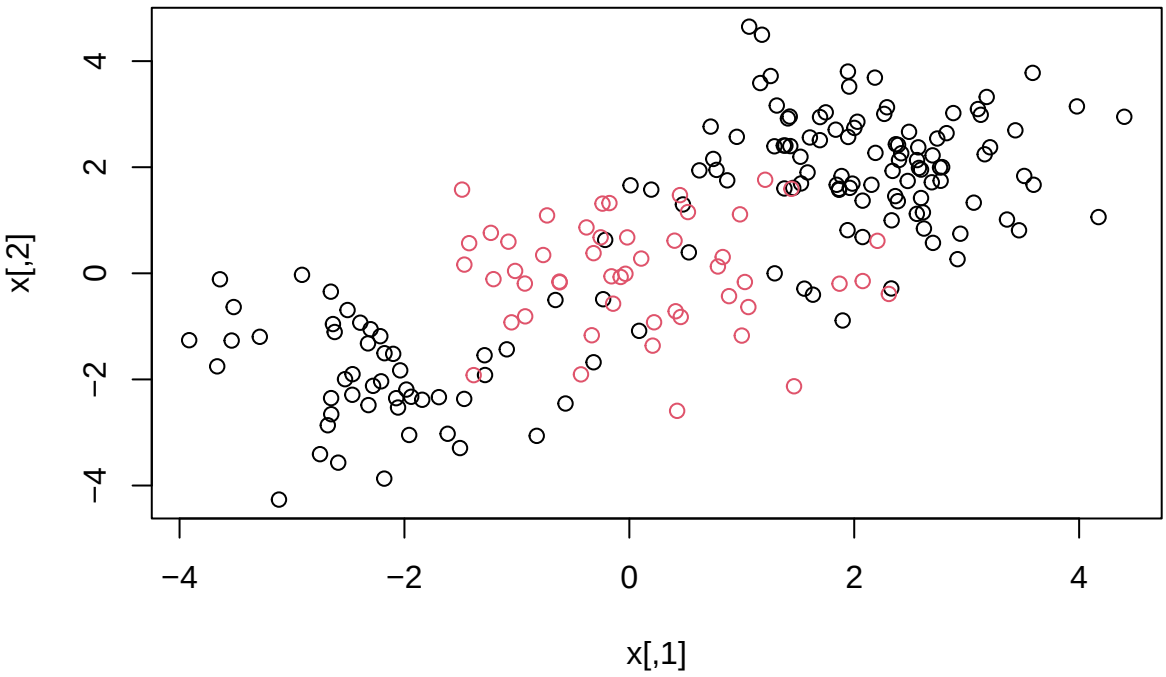
Although our model now has a single misclassified training observation, we can expect it to perform better on test data because of the much wider margin and use of additional support vectors.

9.6.2 Support Vector Machine (SVM)

To use a non-linear kernal in our SVM, we can set the kernel type to other values, such as "polynomial" or "radial". These non-linear kernels also require a second argument; we must specify a "degree" for polynomial" and a "gamma" for "radial".

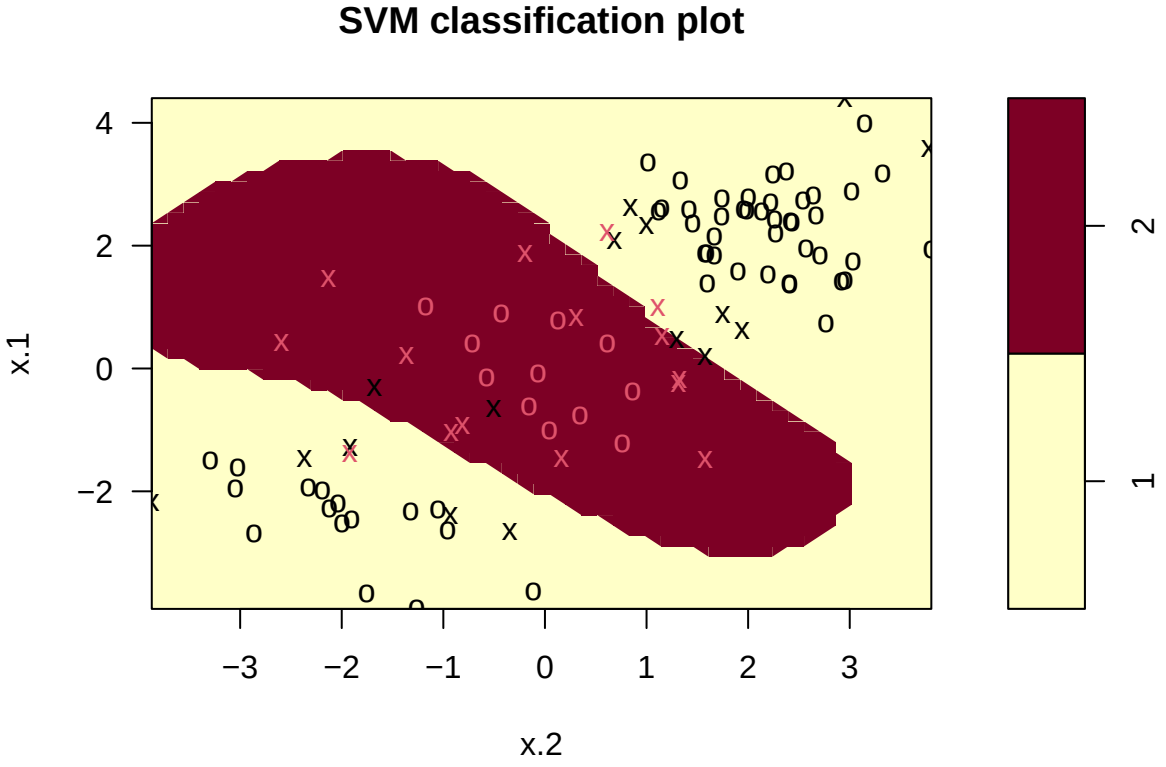
Let's first generate a data set with a non-linear class boundary.

```
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, col = y)
```



Our data is now randomly split with a clearly non-linear boundary. It appears that a radial decision bounday may be a good fit since the observations of class 2 are generally flanked by observations of class 1 to the lower left and upper right. Let's try a radial kernel with a gamma value of 1.

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



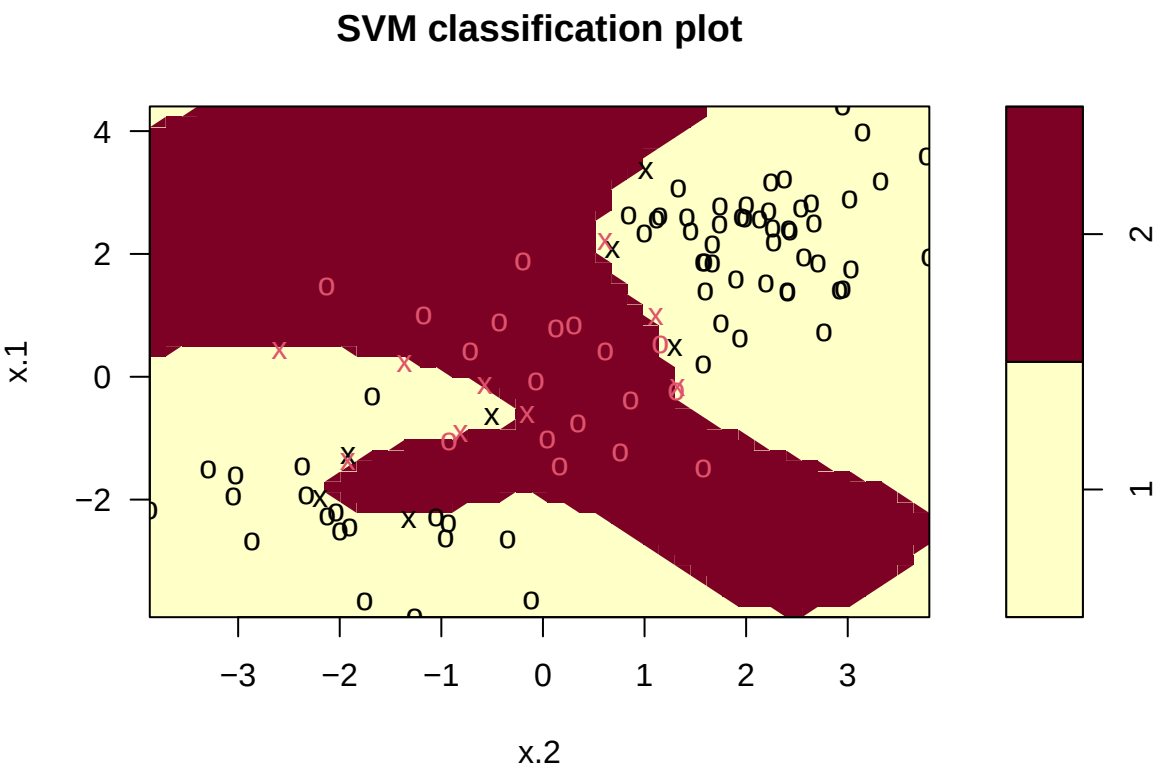
This SVM does indeed have a decision boundary that is clearly not linear; it is shaped like a body of water centered around the "class 2" observations. We can use the summary() function to get the classification counts, but it will unfortunately not count the number of correctly and incorrectly classified training observations for us.

```
summary(svmfit)
```

```
##
## Call:
## svm(formula = y ~ ., data = dat[train, ], kernel = "radial", gamma = 1,
##     cost = 1)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##
## Number of Support Vectors:  31
##
##  ( 16 15 )
##
##
## Number of Classes:  2
##
## Levels:
##  1 2
```

Our SVM with a radial kernal used 31 support vectors, and we see roughly a half dozen misclassified training observations. Again, while we could potentially reduce the number of training errors by increasing the value of cost, we would expect this to produce an over-fit, irregular decision boundary which would likely perform worse with test data. Let's see what that would look like.

```
svmfit <- svm(y ~ ., data = dat[train, ], kernel = "radial",
              gamma = 1, cost = 1e5)
plot(svmfit, dat[train, ])
```



Let's again perform cross-validation using the tune() function to select the best choice training parameters. This time, we will cross-validate across two parameters (gamma and cost) instead of one. This will test a much greater number of potential models, equal to the product of the numbers of parameter values considered.

```
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: 0.07
##
## - Detailed performance results:
##     cost gamma error dispersion
## 1  1e-01   0.5  0.26 0.15776213
## 2  1e+00   0.5  0.07 0.08232726
## 3  1e+01   0.5  0.07 0.08232726
## 4  1e+02   0.5  0.14 0.15055453
## 5  1e+03   0.5  0.11 0.07378648
```

```
## 6  1e-01   1.0  0.22 0.16193277
## 7  1e+00   1.0  0.07 0.08232726
## 8  1e+01   1.0  0.09 0.07378648
## 9  1e+02   1.0  0.12 0.12292726
## 10 1e+03   1.0  0.11 0.11005049
## 11 1e-01   2.0  0.27 0.15670212
## 12 1e+00   2.0  0.07 0.08232726
## 13 1e+01   2.0  0.11 0.07378648
## 14 1e+02   2.0  0.12 0.13165612
## 15 1e+03   2.0  0.16 0.13498971
## 16 1e-01   3.0  0.27 0.15670212
## 17 1e+00   3.0  0.07 0.08232726
## 18 1e+01   3.0  0.08 0.07888106
## 19 1e+02   3.0  0.13 0.14181365
## 20 1e+03   3.0  0.15 0.13540064
## 21 1e-01   4.0  0.27 0.15670212
## 22 1e+00   4.0  0.07 0.08232726
## 23 1e+01   4.0  0.09 0.07378648
## 24 1e+02   4.0  0.13 0.14181365
## 25 1e+03   4.0  0.15 0.13540064
```

The best combination of parameters are cost = 1 and gamma = 0.5. Let's view the accuracy of the test set predictions using a table. We can use "-train" as an index to specify the test data in the dataframe by calling all the observations minus the training data.

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

```
##     pred
## true  1  2
##    1 67 10
##    2  2 21
```

This SVM correctly classified 88 out of 100 test observations, a test accuracy rate of 88%.

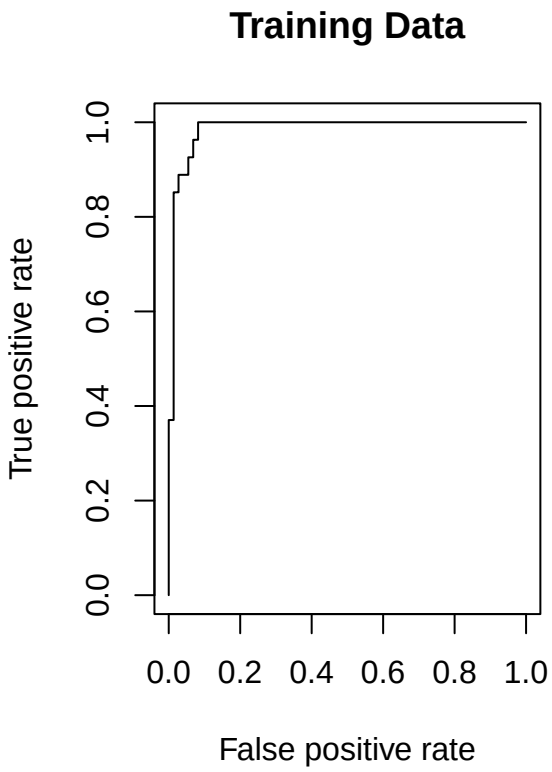
9.6.3 ROC Curves

We can produce Receiver Operator Characteristic (ROC) curves using the ROCR package. Here we create a function to plot an ROC curve given numerical scores and class labels for each observation.

```
library(ROCR)
rocplot <- function(pred, truth, ...) {
  predob <- prediction(pred, truth)
  perf <- performance(predob, "tpr", "fpr")
  plot(perf, ...)
}
```

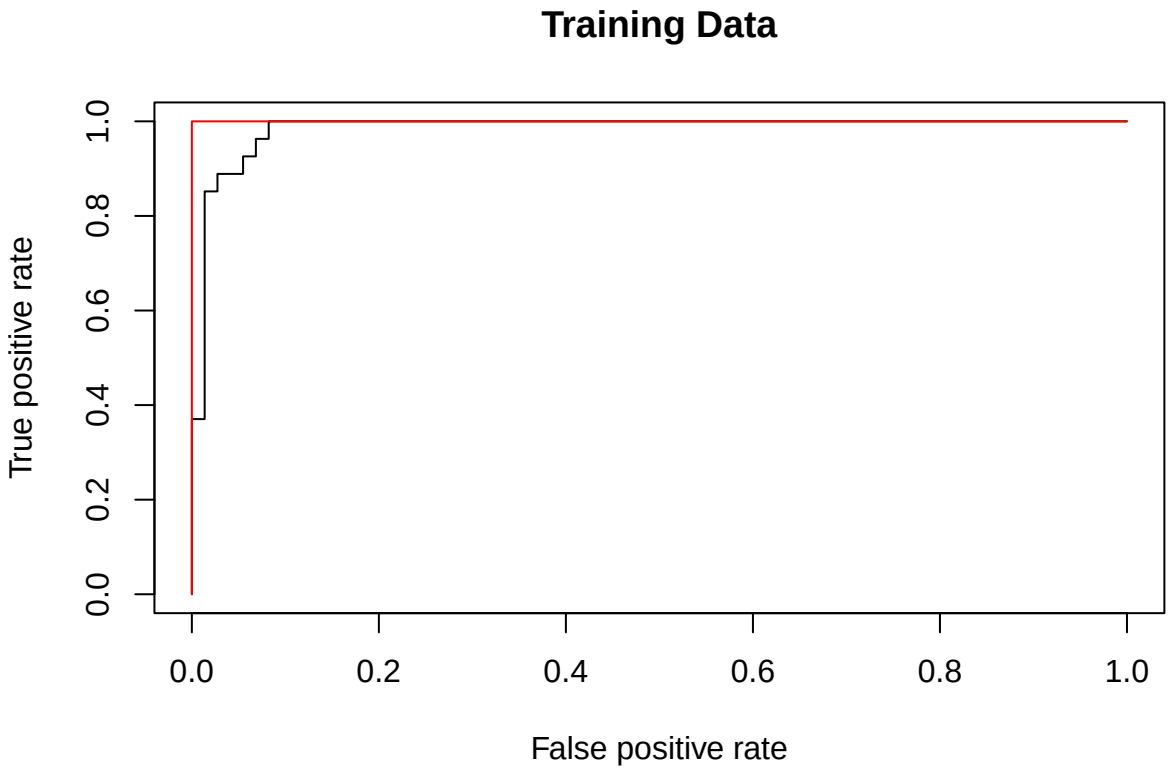
While support vector classifiers (and thus SVMs) output class labels for each observation, we can instead obtain fitted values for each observation. These are the numerical scores used to obtain the class labels. The sign of each fitted value typically decides which side of the decision boundary an observation lies; positives lie on one side, and negatives lie on the other. We can obtain these fitted values directly using the "decision.values = TRUE" parameter when fitting with the svm() function. Then, to plot the ROC curve where negatives correspond to class 1 and positives to class 2, we use the negative of the fitted values.

```
svmfit.opt <- svm(y ~ ., data = dat[train, ],
                  kernel = "radial", gamma = 2, cost = 1, decision.values = T)
fitted1 <- attributes(predict(svmfit.opt, dat[train, ], decision.values = T)
                      )$decision.values
par(mfrow = c(1,2))
rocplot(-fitted1, dat[train, "y"], main = "Training Data")
```



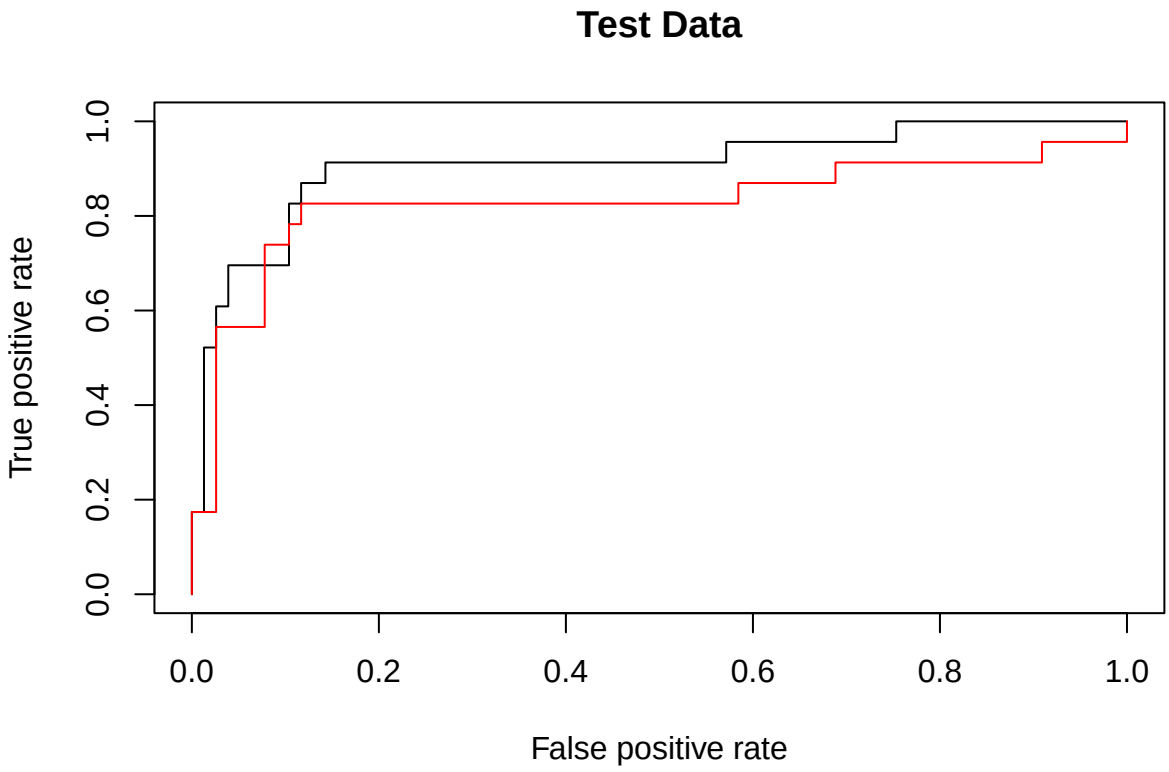
Since we see the ROC curve is hugging the upper left-hand corner of the plot, the infer the SVM is producing generally accurate predictions. If we increase the gamma value, our fit will be more flexible and hopefully more accurate.

```
svmfit.flex <- svm(y ~ ., data = dat[train, ],
                   kernel = "radial", gamma = 50, cost = 1, decision.values = T)
fitted2 <- attributes(
  predict(svmfit.flex, dat[train, ], decision.values = T))$decision.values
rocplot(-fitted1, dat[train, "y"], main = "Training Data")
rocplot(-fitted2, dat[train, "y"], col = "red", add = T)
```



An accuracy rate of 100% on training data usually doesn't bode well for test accuracy. We have probably over-fit the training data with the greater flexibility of gamma = 50. After all, we are really most interested in the level of prediction accuracy on test data. Let's now plot ROC curves for the test data.

```
fittedtest1 <- attributes(
  predict(svmfit.opt, dat[-train, ], decision.values = T))$decision.values
rocplot(-fittedtest1, dat[-train, "y"], main = "Test Data")
fittedtest2 <- attributes(
  predict(svmfit.flex, dat[-train, ], decision.values = T))$decision.values
rocplot(-fittedtest2, dat[-train, "y"], col = "red", add = T)
```

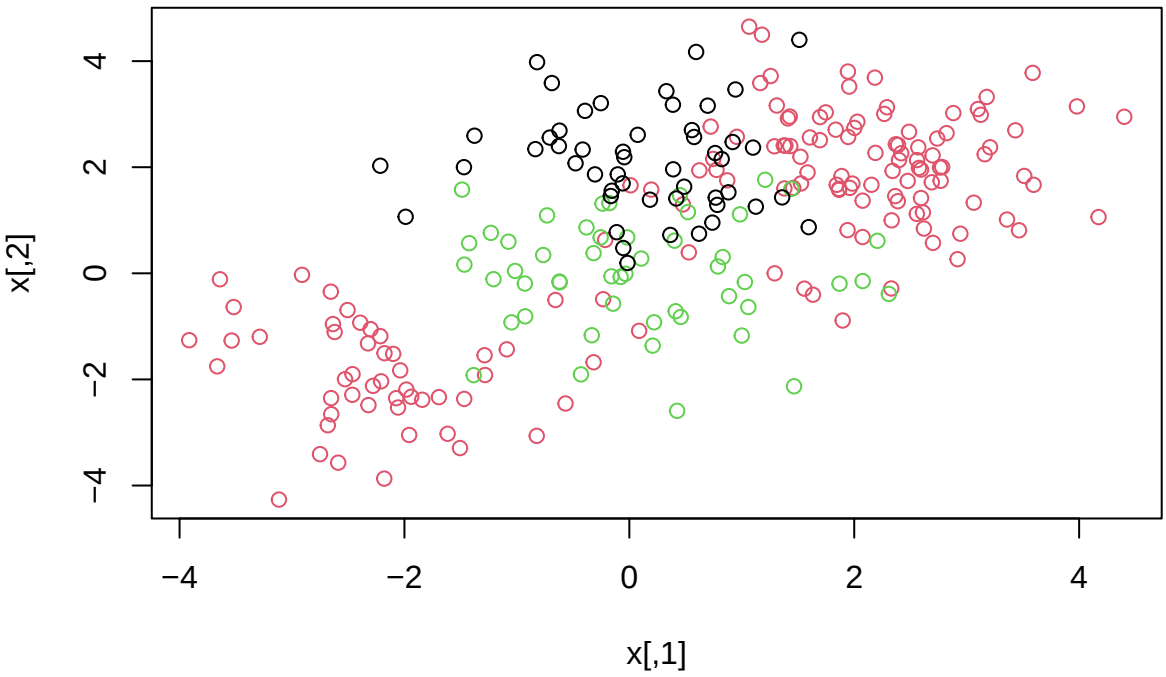


We do indeed see that the less flexible model with gamma = 2 does outperform the overly flexible model in test prediction accuracy.

9.6.4 SVM with Multiple Classes

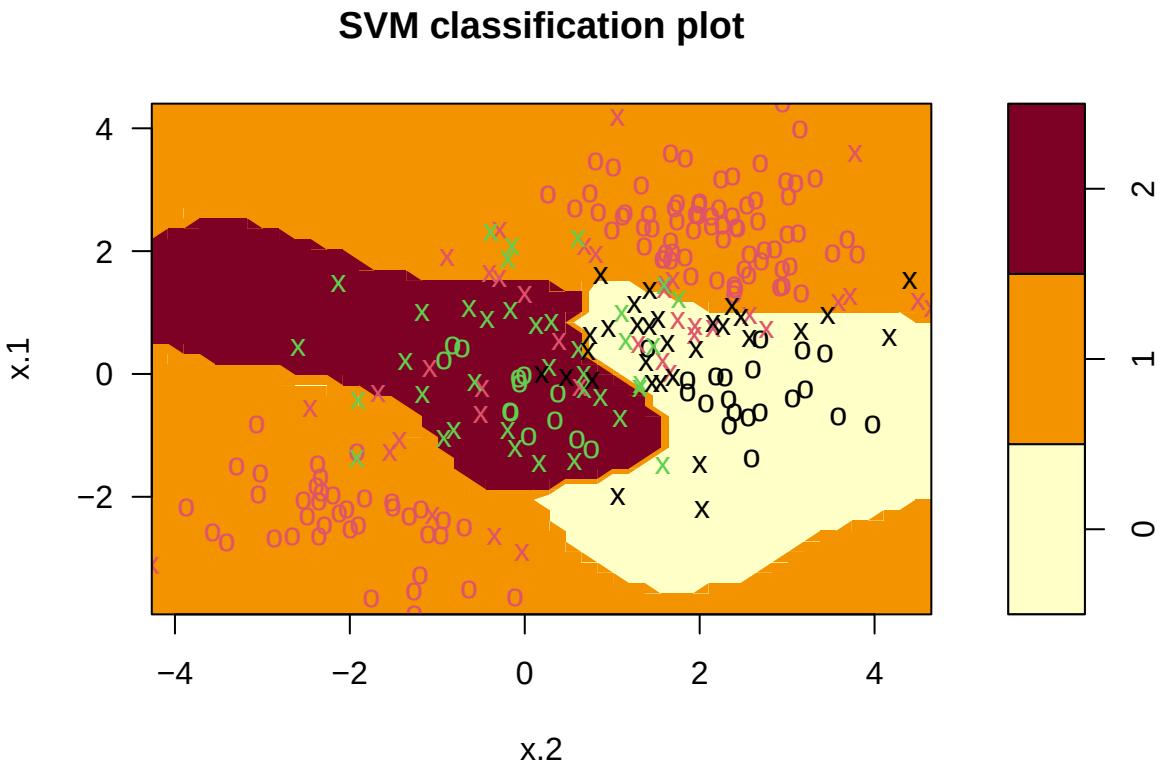
We can use the svm() function in a multi-class classification setting, where it will utilize the one-versus-one method. Let's practice, first by generating a third class of observations in addition to our previous two.

```
set.seed(1)
x <- rbind(x, matrix(rnorm(50*2), ncol = 2))
y <- c(y, rep(0,50))
x[y == 0, 2] <- x[y == 0, 2] + 2
dat <- data.frame(x = x, y = as.factor(y))
par(mfrow = c(1,1))
plot(x, col = (y + 1))
```



The third class of test observations are plotted as black circles. Now, let's fit an SVM to the data:

```
svmfit <- svm(y ~ ., data = dat, kernel = "radial", cost = 10, gamma = 1)
plot(svmfit, dat)
```



Note that we can also use this same e1071 library to perform support vector regression (versus support vector classification) if the response vector passed to svm() is numerical instead of a factor.

9.6.5 Application to Gene Expression Data

Here we test our new SVM tools using the Khan data set. Gene expression measurements were gathered for four types of tumors. We can observe the dimensions of the data and inspect responses.

```
library(ISLR2)
names(Khan)
```

```
## [1] "xtrain" "xtest"  "ytrain" "ytest"
```

```
dim(Khan$xtrain)
```

```
## [1]   63 2308
```

```
dim(Khan$xtest)
```

```
## [1]   20 2308
```

```
length(Khan$ytrain)
```

```
## [1] 63
```

```
length(Khan$ytest)
```

```
## [1] 20
```

```
table(Khan$ytrain)
```

```
##
##  1  2  3  4
##  8 23 12 20
```

```
table(Khan$ytest)
```

```
##
## 1 2 3 4
## 3 6 6 5
```

Let's use a support vector classifier to predict cancer type from the gene expression data. Since there are much greater features than number of observations, we should use a linear kernel; the flexibility of polynomial or radial kernels is not helpful here.

```
dat <- data.frame(
  x = Khan$xtrain,
  y = as.factor(Khan$ytrain)
)
out <- svm(y ~ ., data = dat, kernel = "linear", cost = 10)
summary(out)
```

```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 10)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  10
##
## Number of Support Vectors:  58
##
```

```
##  ( 20 20 11 7 )
##
##
## Number of Classes:  4
##
## Levels:
##  1 2 3 4
```

```
table(out$fitted, dat$y)
```

```
##
##      1  2  3  4
##   1  8  0  0  0
##   2  0 23  0  0
##   3  0  0 12  0
##   4  0  0  0 20
```

Wow! Not a single training error. Since there are so many variables relative to training observations, it can be easy to find hyperplanes which fully separate the classes. However, as always, we are more concerned about the support vector classifier's performance on test observations. Let's see how it fares.

```
dat.te <- data.frame(
  x = Khan$xtest,
  y = as.factor(Khan$ytest)
)
pred.te <- predict(out, newdata = dat.te)
table(pred.te, dat.te$y)
```

```
##
## pred.te 1 2 3 4
##       1 3 0 0 0
##       2 0 6 2 0
##       3 0 0 4 0
##       4 0 0 0 5
```

Our support vector classifier yielded two errors on the test data, both of which it classified a type "2" instead of a type "3". This yields an accuracy rate of 90%.

Reflection

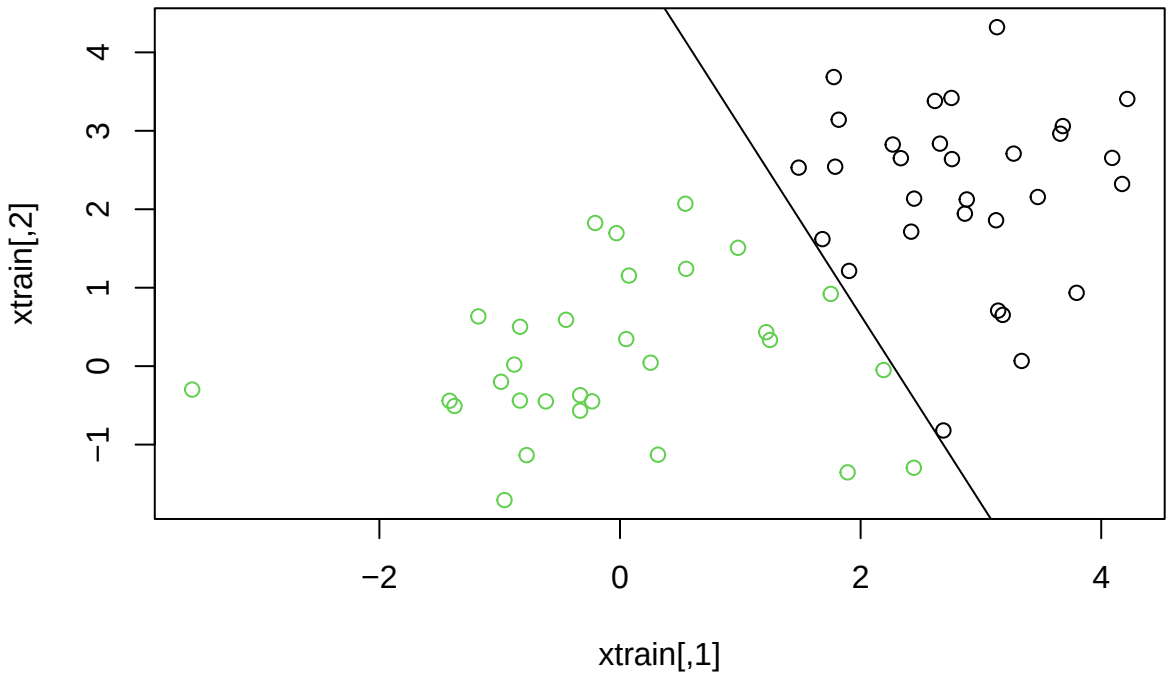
Support vector machines (SVMs) are useful classification models because their varying degrees of flexibility through the use of kernels. The lessons we learned from the regression setting generally apply to the SVM setting. For example, we should being wary of overfitting, staying conscious of the bias-variance trade-off in choosing a value of the "cost" of margin violations. The e1071 library provides intuitive tools for leveraging SVMs, including built-in kernels and cross-validation.

4 ISLR Exercise 9.6

In this exercise, we explore the test performance of a support vector classifier using a range of "cost" values. "Cost" represents the penalty for margin violations on the training set. We are interested to see if support vector classifiers with a large value of cost (which does not misclassify any training observations) will actually perform worse on test data than a support vector classifier with a small value of cost (which misclassifies a few training observations).

a.) First, we generate a data set that is "just barely linearly separable" with a 2-dimensional hyperplane (a line). We plot the observations with color coding matching classification for visibility. We also plot a line separating the two classifications as a visual demonstration of separability.

```r
#Generate a random data set with normalized x1 and x2 values
s <- 17
set.seed(s)
n <- 60
xtrain <- matrix(data = rnorm(n * 2), ncol = 2)

#Assign observations to one of two classifications
ytrain <- c(rep(-1, n/2), rep(1, n/2))

#Displace the observations of one class relative to the other
x1shift = 2.5
x2shift = 2.5
xtrain[1:n/2, 1] <- xtrain[1:n/2, 1] + x1shift
xtrain[1:n/2, 2] <- xtrain[1:n/2, 2] + x2shift

#Plot the observations colored by classification.
plot(xtrain, col = ytrain + 10)

#Plot a boundary line showing the observations are barely linearly separable.
abline(a=5.45, b=-2.4)
```



b.) Now we perform cross-validation on support vector classifiers with a range of cost values. We will calculate error rates, count the number of misclassified training observations, and determine how the two are related.

```r
library(e1071)

#Create data frame of features and responses
traindata <- data.frame(x = xtrain, y = as.factor(ytrain))

#Perform cross-validation on a range of cost values
set.seed(s)
crossvalidation <- tune(svm, y ~ ., data = traindata, kernel = "linear",
                        ranges = list(cost = c(1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2,
```

```
                                                1e3)))

#Inspect the cross-validation error rate for each value of cost
#summary(crossvalidation)

#Generate individual models for each cost value used in the cross-validation
svm.cost0.001 <- svm(y ~ ., data = traindata, kernel = "linear", cost = 0.001)
svm.cost0.01 <- svm(y ~ ., data = traindata, kernel = "linear", cost = 0.01)
svm.cost0.1 <- svm(y ~ ., data = traindata, kernel = "linear", cost = 0.1)
svm.cost1 <- svm(y ~ ., data = traindata, kernel = "linear", cost = 1)
svm.cost10 <- svm(y ~ ., data = traindata, kernel = "linear", cost = 10)
svm.cost100 <- svm(y ~ ., data = traindata, kernel = "linear", cost = 100)
svm.cost1000 <- svm(y ~ ., data = traindata, kernel = "linear", cost = 1000)

#Generate tables of predicted classifications versus actual classifications
Table0.001 = table(Predictions_Cost_0.001 = predict(svm.cost0.001, traindata),
      Actual = traindata$y)
Table0.01 = table(Predictions_Cost_0.01 = predict(svm.cost0.01, traindata),
      Actual = traindata$y)
Table0.1 = table(Predictions_Cost_0.1 = predict(svm.cost0.1, traindata),
      Actual = traindata$y)
Table1 = table(Predictions_Cost_1 = predict(svm.cost1, traindata),
      Actual = traindata$y)
Table10 = table(Predictions_Cost_10 = predict(svm.cost10, traindata),
      Actual = traindata$y)
Table100 = table(Predictions_Cost_100 = predict(svm.cost100, traindata),
      Actual = traindata$y)
Table1000 = table(Predictions_Cost_1000 = predict(svm.cost1000, traindata),
      Actual = traindata$y)

#Count the number of training error misclassifications for each cost
ErrorCount0.001 = Table0.001[2] + Table0.001[3]
ErrorCount0.01 = Table0.01[2] + Table0.01[3]
ErrorCount0.1 = Table0.1[2] + Table0.1[3]
ErrorCount1 = Table1[2] + Table1[3]
ErrorCount10 = Table10[2] + Table10[3]
ErrorCount100 = Table100[2] + Table100[3]
ErrorCount1000 = Table1000[2] + Table1000[3]

#Calculate training misclassification error rates for each cost
ErrorRate0.001 <- ErrorCount0.001 / n
ErrorRate0.01 <- ErrorCount0.01 / n
ErrorRate0.1 <- ErrorCount0.1 / n
ErrorRate1 <- ErrorCount1 / n
ErrorRate10 <- ErrorCount10 / n
ErrorRate100 <- ErrorCount100 / n
ErrorRate1000 <- ErrorCount1000 / n

#Compare Training Error Count, Training Error Rate,
#and Cross Validation error rate
TrainError = c(Cost_0.001 = ErrorCount0.001,
                          Cost_0.01 = ErrorCount0.01, Cost_0.1 = ErrorCount0.1,
                          Cost_1 = ErrorCount1, Cost_10 = ErrorCount10,
```

```
                          Cost_100 = ErrorCount100, Cost_1000 = ErrorCount1000)
TrainErrorRate = c(ErrorRate0.001, ErrorRate0.01, ErrorRate0.1,
                         ErrorRate1, ErrorRate10, ErrorRate100, ErrorRate1000)
CrossValidation = crossvalidation$performance$error

#Display a table comparing training errors with cross-validation error rate
TrainingTable <- cbind(TrainError, TrainErrorRate, CrossValidation)
TrainingTable
```

```
##             TrainError TrainErrorRate CrossValidation
## Cost_0.001          5     0.08333333      0.60000000
## Cost_0.01           4     0.06666667      0.13333333
## Cost_0.1            2     0.03333333      0.03333333
## Cost_1              1     0.01666667      0.01666667
## Cost_10             2     0.03333333      0.06666667
## Cost_100            0     0.00000000      0.01666667
## Cost_1000           0     0.00000000      0.01666667
```

With the above table, we can compare the training misclassification errors with the cross-validation error rate across a range of cost values. When the cost value is small, at 0.001, we have 5 training misclassifications, an error rate of 8%. When cost is high, at 100 or 1000, the error rate falls to zero. The cross-validation results, however, are not monotonic decreasing. Generally, as training error misclassification rate falls, the cross-validation error rate first steeply falls and then modestly rises. The cost value with the minimum cross-validation error is 0.1.

c.) Now that we have examined the performance of our SVMs on training data, let's proceed to work with test data. First, we will generate a test dataset in the same manner as our training dataset. Then we will compute the test errors corresponding to each value of cost we considered previously.

```
#Generate test data in same fashion as training data
set.seed(s+1)
n <- 60
xtest = matrix(rnorm(2*n), ncol = 2)
ytest = c(rep(-1, n/2), rep(1, n/2))
xtest[1:n/2, 1] <- xtest[1:n/2, 1] + x1shift
xtest[1:n/2, 2] <- xtest[1:n/2, 2] + x2shift
#plot(xtest, col = ytest + 11)
testdata <- data.frame(x = xtest, y = as.factor(ytest))

#Compute test errors corresponding to each value of cost considered
TestTable0.001 = table(Predictions_Cost_0.001 = predict(svm.cost0.001,testdata),
      Actual = testdata$y)
TestTable0.01 = table(Predictions_Cost_0.01 = predict(svm.cost0.01, testdata),
      Actual = testdata$y)
TestTable0.1 = table(Predictions_Cost_0.1 = predict(svm.cost0.1, testdata),
      Actual = testdata$y)
TestTable1 = table(Predictions_Cost_1 = predict(svm.cost1, testdata),
      Actual = testdata$y)
TestTable10 = table(Predictions_Cost_10 = predict(svm.cost10, testdata),
      Actual = testdata$y)
TestTable100 = table(Predictions_Cost_100 = predict(svm.cost100, testdata),
      Actual = testdata$y)
TestTable1000 = table(Predictions_Cost_1000 = predict(svm.cost1000, testdata),
      Actual = testdata$y)
```

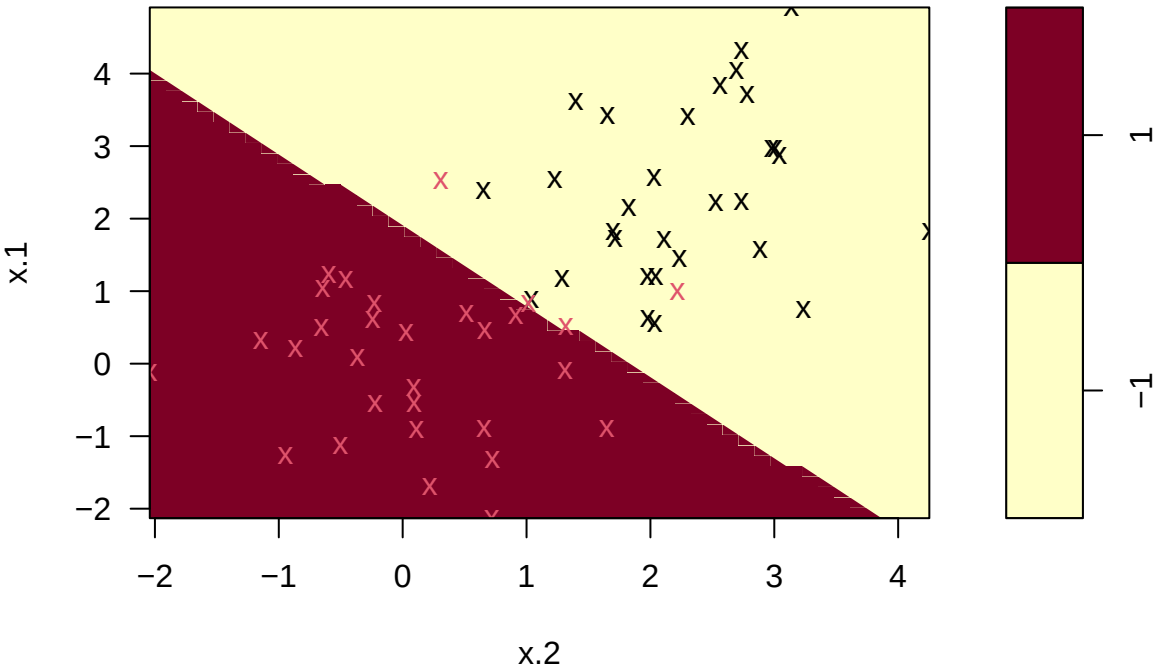
```
#Count the number of test error misclassifications for each cost
TestErrorCount0.001 = TestTable0.001[2] + TestTable0.001[3]
TestErrorCount0.01 = TestTable0.01[2] + TestTable0.01[3]
TestErrorCount0.1 = TestTable0.1[2] + TestTable0.1[3]
TestErrorCount1 = TestTable1[2] + TestTable1[3]
TestErrorCount10 = TestTable10[2] + TestTable10[3]
TestErrorCount100 = TestTable100[2] + TestTable100[3]
TestErrorCount1000 = TestTable1000[2] + TestTable1000[3]

#Calculate test misclassification error rates for each cost
TestErrorRate0.001 <- TestErrorCount0.001 / n
TestErrorRate0.01 <- TestErrorCount0.01 / n
TestErrorRate0.1 <- TestErrorCount0.1 / n
TestErrorRate1 <- TestErrorCount1 / n
TestErrorRate10 <- TestErrorCount10 / n
TestErrorRate100 <- TestErrorCount100 / n
TestErrorRate1000 <- TestErrorCount1000 / n

#Compare test error counts/rate again training error counts/rate
#and cross-validation error rate
TestTable <- cbind(TestError = c(TestErrorCount0.001,
                                 TestErrorCount0.01,
                                 TestErrorCount0.1, TestErrorCount1,
                                 TestErrorCount10, TestErrorCount100,
                                 TestErrorCount1000),
                   TestErrorRate = c(TestErrorRate0.001, TestErrorRate0.01,
                                     TestErrorRate0.1, TestErrorRate1,
                                     TestErrorRate10, TestErrorRate100,
                                     TestErrorRate1000))
CombinedTable <- cbind(TestTable, TrainingTable)
CombinedTable
```

```
##            TestError TestErrorRate TrainError TrainErrorRate CrossValidation
## Cost_0.001         2    0.03333333          5     0.08333333      0.60000000
## Cost_0.01          3    0.05000000          4     0.06666667      0.13333333
## Cost_0.1           6    0.10000000          2     0.03333333      0.03333333
## Cost_1             5    0.08333333          1     0.01666667      0.01666667
## Cost_10            5    0.08333333          2     0.03333333      0.06666667
## Cost_100           8    0.13333333          0     0.00000000      0.01666667
## Cost_1000          8    0.13333333          0     0.00000000      0.01666667
```

The SVM with a cost value of 0.001 produced the fewest test classification errors at 3%. This model was the most flexible of the seven we trained previously. On the other hand, the SVMs with cost values of 100 and 1000 produced the most classification errors on our test data, a 13% test error rate. Notably, these results are in contrast to the training error performance where the relationship between cost and classification error rate was reversed: higher costs had yielded lower training error rates.

We plot the SVM with cost = 0.001 here. Notice that every observation is a support vector.

```
plot(svm.cost0.001, testdata)
```



The cross-validation error rate was exceptionally high at 60% for a cost value of 0.001, yet this cost value produced the lowest test error rate. Our cross-validation suggested a cost-value of 0.1 was optimal, yet this value produced a modestly high test error rate.

d.) The dissonance between classification error rates on the training data and test data highlights the dangers of strictly fitting data which is just barely linearly separable. When a separating hyperplane depends on just a few observations, the support vector classifier is subject to high variance and overfitting. Both the test error rate and cross-validation rates for high cost values support this conclusion.

Interestingly, the test error rate and cross-validation error rate do not support the same cost values, though they both provide evidence that a high cost model is not best. Cross-validation supports a moderate cost of 0.1 while the test error rate supports a very low cost of 0.001. As mentioned before, the cross-validation had a very high error rate of over 50% for the 0.001 cost model. What could be happening here?

We believe part of the discrepancy is due to the relatively small number of observations (n=60) used in both the training and test data. Although utilizing larger data sets would have enabled us to make more robust conclusions, we had to limit ourselves to smaller "sample sizes". Creating randomly generated data sets that were "just barely linearly separable" with large number of observations tended to require the center of each classification of observations to be significantly displaced, by 4 or more standard deviations. These large data sets yielded SVMs whose accuracy were less prone to changes in the hyperplane slope, yielding more homogeneous results for varying degrees of cost.

Let's try now to leverage the law of large numbers by creating a very large test data set to get a better sense of the average power at each cost value. Below, we compare error rates for our models on 10,000 test observations. We hide the code for brevity.

```
##            TestError TestErrorRate TrainError TrainErrorRate CrossValidation
## Cost_0.001       514        0.0514          5     0.08333333      0.60000000
## Cost_0.01        383        0.0383          4     0.06666667      0.13333333
## Cost_0.1         409        0.0409          2     0.03333333      0.03333333
## Cost_1           496        0.0496          1     0.01666667      0.01666667
## Cost_10          506        0.0506          2     0.03333333      0.06666667
```

```
## Cost_100         662         0.0662          0      0.00000000      0.01666667
## Cost_1000        683         0.0683          0      0.00000000      0.01666667
```

On a much larger test data set, we see that SVMs with cost values of 0.1 and 0.01 perform better the 0.001 cost. This outcome is closer to what we would have expected based on our cross-validation which favored a 0.1 cost value. Earlier, the cross-validation results did not correspond as nicely with the test error rate from the smaller test data set. It seems randomness was playing a significant role in measuring the test accuracy of the models on small data sets.

Cross-validation is great tool for informing our choice of model parameters. When paired with other decision-making tools, we can generate a model that best meets our needs without falling into common traps such as overfitting training data. We should be especially be careful in edge cases, such as when data are just barely linearly separable, as we can be misled by performance on these precarious training data.

5 Murphy Chapter 18 Summary

18.1 Classification and Regression Trees

Classification and regression trees recursively partition the input space and define a local model in each resulting region of input space. This can be represented by a tree with a leaf per input region.

18.1.1 Model Definition

A regression tree consists of a set of nested decision rules. At each node i, the feature dimension d_i of the input vector $\boldsymbol{x}$ is compared to a threshold value t_i, after which the input flows down the left or right branch, depending on its comparison to the threshold. The predicted output for an input in that recursively segmented section of the input space is given at the leaves of the tree.

Formally, a regression tree is defined by

$$f(\boldsymbol{x};\boldsymbol{\theta}) = \sum_{j=1}^{J} w_j \mathbb{I}(\boldsymbol{x} \in R_j),$$

where R_j is the region specified by the j'th leaf node, w_j is the predicted output for that node, and $\boldsymbol{\theta} = \{(R_j, w_j) : j = 1 : J\}$, where J is the number of nodes. The regions are defined by the feature dimensions and corresponding thresholds at each split in the tree.

For classification problems, the leaves contain a distribution over the class labels, as opposed to the mean response.

18.1.2 Model Fitting

To fit the model, we minimize

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \ell(y_n, f(\boldsymbol{x}_n;\boldsymbol{\theta})) = \sum_{j=1}^{J} \sum_{\boldsymbol{x}_n \in R_j} \ell(y_n, w_j).$$

The standard way to find the optimal partitioning of the data is to use a greedy procedure, iteratively growing the tree one node at a time.

Suppose we are at node i, let $\mathcal{D}_i = \{(\boldsymbol{x}_n, y_n) \in N_i\}$ be the set of examples that reach this node. If the j'th feature is a real-valued scalar, we partition the data at node i by comparing to a threshold t. The set of possible thresholds $\mathcal{T}_j$ for feature j can be obtained by sorting the unique values of $\{x_{nj}\}$. For each possible threshold, the left and right splits are defined by $\mathcal{D}_i^L(j,t) = \{(\boldsymbol{x}_n, y_n) \in N_i : x_{n,j} \leq t\}$ and $\mathcal{D}_i^R(j,t) = \{(\boldsymbol{x}_n, y_n) \in N_i : x_{n,j} > t\}$, respectively.

If the j'th feature is categorical with K_j possible values, the tree checks whether the feature is equal to each of those values, which defines a set of K_j possible binary splits: $\mathcal{D}_i^L(j,t) = \{(\boldsymbol{x}_n, y_n) \in N_i : x_{n,j} = t\}$ and $\mathcal{D}_i^R(j,t) = \{(\boldsymbol{x}_n, y_n) \in N_i : x_{n,j} \neq t\}$.

After computing $\mathcal{D}_i^L(j,t)$ and $\mathcal{D}_i^R(j,t)$ for each j and t at node i, the best feature j_i and the best value for that feature t_i are chosen as follows:

$$(j_i, t_i) = \arg \min_{j \in \{1,\ldots,D\}} \min_{t \in \mathcal{T}_j} \frac{|\mathcal{D}_i^L(j,t)|}{|\mathcal{D}_i|} c(\mathcal{D}_i^L(j,t)) + \frac{|\mathcal{D}_i^R(j,t)|}{|\mathcal{D}_i|} c(\mathcal{D}_i^R(j,t)),$$

where c can be the mean squared error

$$\frac{1}{|\mathcal{D}|} \sum_{n \in \mathcal{D}_i} (y_n - \bar{y})^2$$

for regression. For classification, we first compute the empirical distribution over class labels for node i:

$$\hat{\pi}_{ic} = \frac{1}{|\mathcal{D}_i|} \sum_{n \in \mathcal{D}_i} \mathbb{I}(y_n = c),$$

and use this to compute the Gini index

$$G_i = \sum_{c=1}^{C} \hat{\pi}_{ic}(1 - \hat{\pi}_{ic}) = 1 - \sum_c \hat{\pi}_{ic}^2,$$

which gives the expected error rate. Note that $\hat{\pi}_{ic}$ is the probability of a random entry in the leaf belonging to class c, and $1 - \hat{\pi}_{ic}$ is the probability it would be misclassified.

We could also define the function c as the entropy or deviance of the node:

$$H_i = -\sum_{c=1}^{C} \hat{\pi}_{ic} \log \hat{\pi}_{ic}.$$

The above equation for (j_i, t_i) can then be used to find the best feature and threshold at each node. The data is then partitioned, and the fitting algorithm is called recursively on each subset of the data.

18.1.3 Regularization

If the tree becomes deep enough, the error rate on the training set tends to 0 by partitioning the input space into small enough regions, which leads to overfitting. There are two main approaches to prevent this: stop the tree growing process according to some heuristic (e.g. reaching some maximum depth), or grow the tree to its maximum depth and prune it back by merging split subtrees back into their parent.

18.1.4 Handling Missing Input Features

The standard heuristic is to look for a series of "backup" variables, which can induce a similar partition to the chosen variable at any given split. These are called surrogate splits.

A simpler approach for categorical variables is to code "missing" as a new value and treat the data as fully observed.

18.1.5 Pros and Cons

Trees are easy to interpret, can easily handle mixed discrete and continuous inputs, can perform automatic variable selection, are relatively robust to outliers, are fast to fit and scale well, can handle missing input features, and there's no need to standardize data.

However, they do not predict accurately compared to other models and they are unstable (i.e. small changes to the input can have large implications later in the tree).

18.2 Ensemble Learning

A simple way to reduce the variance of high variance estimators is to average multiple models. This is ensemble learning. This sort of model has the form

$$f(y|\boldsymbol{x}) = \frac{1}{|\mathcal{M}|} \sum_{m \in \mathcal{M}} f_m(y|\boldsymbol{x}),$$

where f_m is the m'th base model. The ensemble will have similar bias to the base models but lower variance. For regression models, averaging is a good way to combine predictions, and for classifiers it is sometimes better to take a majority vote of the outputs.

18.2.1 Stacking

Alternatively, we can learn to combine the base models by using

$$f(y|\boldsymbol{x}) = \sum_{m \in \mathcal{M}} w_m f_m(y|\boldsymbol{x}),$$

which is called stacking or stacked generalization. The combination weights used by the stacking method need to be trained on a separate dataset, otherwise they put all their mass on the best performing base model.

18.2.2 Ensembling is not Bayes Model Averaging

An ensemble considers a larger hypothesis class of the form

$$p(y|\boldsymbol{x}, \boldsymbol{w}, \boldsymbol{\theta}) = \sum_{m \in \mathcal{M}} w_m p(y|\boldsymbol{x}, \boldsymbol{\theta}_m),$$

whereas Bayes Model Averaging uses

$$p(y|\boldsymbol{x}, \mathcal{D}) = \sum_{m \in \mathcal{M}} p(m|\mathcal{D}) p(y|\boldsymbol{x}, m, \mathcal{D}).$$

The difference is that the weights $p(m|\mathcal{D})$ sum to one and with infinite data, only a single model is chosen (namely the MAP model). However, the ensemble weights w_m are arbitrary and don't collapse to a single model.

18.3 Bagging

Bagging stands for "bootstrap aggregating", a form of ensemble learning in which M different base models are fit to differently randomly sampled (with replacement) versions of the data.

The disadvantage of bootstrap is that each base model only sees 63% of the unique input examples on average. The 37% of the training instances that are not used by a given base model are called out-of-bag instances. The oob instances can be used in prediction as an estimate of test set performance, which is a useful alternative to cross validation.

The main advantage of bootstrap is that it prevents the ensemble from relying too much on any individual training example. This generally increases with the size of the ensemble.

Bagging doesn't always improve performance as it relies on the base models being unstable estimators, which is the case for decision trees, but might not be true for other models.

18.4 Random Forests

The technique of random forests tries to decorrelate the base learners by learning trees based on a randomly chosen subset of input variables, as well as a randomly chosen subset of data cases.

18.5 Boosting

Consider the model of an ensemble of trees

$$f(\boldsymbol{x};\boldsymbol{\theta}) = \sum_{m=1}^{M} \beta_m F_m(\boldsymbol{x};\boldsymbol{\theta}_m)$$

where F_m is the m'th tree and β_m is the corresponding weight. Usually, $\beta_m = 1/M$, but this can be generalized by allowing the F_m functions to be general function approximators, such as neural networks. This is called an additive model, and can be thought of as a linear model with adaptive basis functions. As usual, we want to minimize the empirical loss

$$\mathcal{L}(f) = \sum_{i=1}^{N} \ell(y_i, f(\boldsymbol{x}_i)).$$

Boosting is an algorithm for sequentially fitting additive models where each F_m is a binary classifier that returns $F_m \in \{-1, +1\}$. In particular, after fitting F_1 on the original data, the data samples are weighted by the errors made by F_1, so that misclassified examples get more weight. Then F_2 is fit on to this weighted set, and repeat until the desired number M of components are fit.

As long as each F_m has an accuracy that's better than chance, the final ensemble of classifiers will have higher accuracy than any given component.

Boosting reduces bias by fitting trees that depend on each other, while bagging and random forests reduce variance by fitting independent trees.

18.5.1 Forward Stagewise Additive Modeling

In forward stagewise additive modeling, we sequentially optimize the empirical loss for general, differential loss functions where f is an additive model. That is, at iteration m, we compute

$$(\beta_m, \boldsymbol{\theta}_m) = \arg\min_{\beta,\boldsymbol{\theta}} \sum_{i=1}^{N} \ell(y_i, f_{m-1}(\boldsymbol{x}_i) + \beta F(\boldsymbol{x}_i;\boldsymbol{\theta})),$$

and set

$$f_m(\boldsymbol{x}) = f_{m-1}(\boldsymbol{x}) + \beta_m F(\boldsymbol{x};\boldsymbol{\theta}_m) = f_{m-1}(\boldsymbol{x}) + \beta_m F_m(\boldsymbol{x}).$$

How to perform this optimization depends on the loss function we choose.

18.5.2 Quadratic Loss and Least Squares Boosting

Suppose we use squared error loss $\ell(y, \hat{y}) = (y - \hat{y})^2$. Then the i'th term at step m becomes

$$\ell(y_i, f_{m-1}(\boldsymbol{x}_i) + \beta F(\boldsymbol{x}_i;\boldsymbol{\theta})) = (y_i - f_{m-1}(\boldsymbol{x}_i) - \beta F(\boldsymbol{x}_i;\boldsymbol{\theta}))^2 = (r_{im} - \beta F(\boldsymbol{x}_i;\boldsymbol{\theta}))^2,$$

where $r_{im} = y_i - f_{m-1}(\boldsymbol{x}_i)$ is the residual of the current model on the i'th observation. This can be minimized by simply setting $\beta = 1$ and fitting F to the residual errors. This is least squares boosting.

18.5.3 Exponential Loss and AdaBoost

Suppose we are interested in binary classification, i.e. predicting $\tilde{y}_i \in \{-1, +1\}$. Assume the weak learner computes

$$p(y = 1|\boldsymbol{x}) = \frac{e^{F(\boldsymbol{x})}}{e^{-F(\boldsymbol{x})} + e^{F(\boldsymbol{x})}} = \frac{1}{1 + e^{-2F(\boldsymbol{x})}}$$

The negative log likelihood is given by

$$\ell(\tilde{y}, F(\boldsymbol{x})) = \log(1 + e^{-2\tilde{y}F(\boldsymbol{x})}),$$

which can be minimized by ensuring that the margin $m(\boldsymbol{x}) = \tilde{y}F(\boldsymbol{x})$ is as large as possible. The log loss penalizes negative margins more heavily than positive ones, which is desired since positive margins are already classified correctly.

We could also consider the exponential loss

$$\ell(\tilde{y}, F(\boldsymbol{x})) = \exp(-\tilde{y}F(\boldsymbol{x})).$$

With an infinite sample size, the optimal solution to the exponential loss is the same as for the log loss.

It turns out that the exponential loss is easier to optimize in the boosting setting however. If the base classifier F_m returns a binary class label, the resulting algorithm is called discrete AdaBoost. If F_m returns a probability instead, a modified version, called real AdaBoost, can be used.

At step m, we minimize

$$L_m(F) = \sum_{i=1}^{N} \omega_{i,m} \exp(-\beta \tilde{y}_i F(\boldsymbol{x}_i)),$$

where $\omega_{i,m} = \exp(-\tilde{y}_i f_{m-1}(\boldsymbol{x}_i))$ is a weight applied to datacase i, and $\tilde{y}_i \in \{-1, +1\}$. Solving for the optimal function F_m and the size of the update β, the overall update becomes

$$f_m(\boldsymbol{x}) = f_{m-1}(\boldsymbol{x}) + \beta_m F_m(\boldsymbol{x}),$$

where

$$F_m = \arg\min_F \sum_{i=1}^{N} \omega_{i,m} \mathbb{I}(\tilde{y}_i \neq F(\boldsymbol{x}_i)),$$

$$\beta = \frac{1}{2} \log \frac{1 - \text{err}_m}{\text{err}_m},$$

and

$$\text{err}_m = \frac{\sum_{i=1}^{N} \omega_{i,m} \mathbb{I}(\tilde{y}_i \neq F_m(\boldsymbol{x}_i))}{\sum_{i=1}^{N} \omega_{i,m}}.$$

After updating the strong learner, the weights are computed for the next iteration as

$$\omega_{i,m+1} = \begin{cases} \omega_{i,m} e^{\alpha_m} & \text{if } \tilde{y}_i \neq F_m(\boldsymbol{x}_i) \\ \omega_{i,m} & \text{otherwise} \end{cases},$$

where $\alpha_m = 2\beta_m$.

SAMME stands for stagewise additive modeling using a multiclass exponential loss function is an AdaBoost-like algorithm to minimize a multiclass generalization of exponential loss.

18.5.4 LogitBoost

Exponential loss puts a lot of weight on misclassified examples, which makes the method sensitive to outliers. Since $e^{-\tilde{y}f}$ is not the logarithm of a pmf for binary variables $\tilde{y} \in \{-1, +1\}$, we can't recover probability estimates from $f(\boldsymbol{x})$.

Log loss only punishes mistakes linearly, which means that we can extract probabilities using

$$p(y = 1|\boldsymbol{x}) = \frac{1}{1 + e^{-2f(\boldsymbol{x})}}.$$

The expected log-loss is given by

$$L_m(F) = \sum_{i=1}^{N} \log[1 + \exp(-2\tilde{y}_i(f_{m-1}(\boldsymbol{x}) + F(\boldsymbol{x}_i)))],$$

which can be optimized using a Newton update. This is known as logitBoost. The key subroutine is the ability of the weak learner F to solve a weighted least squares problem.

18.5.5 Gradient Boosting

It's possible to derive a generic version of boosting for different loss functions, known as gradient boosting. Suppose we want to solve $\hat{\boldsymbol{f}} = \arg\min_{\boldsymbol{f}} \mathcal{L}(\boldsymbol{f})$ by performing gradient descent in the space of functions. At step m, let $\boldsymbol{g}_m$ be the gradient of $\mathcal{L}(\boldsymbol{f})$ evaluated at $\boldsymbol{f} = \boldsymbol{f}_{m-1}$:

$$g_{im} = \left[\frac{\partial \ell(y_i, f(\boldsymbol{x}_i))}{\partial f(\boldsymbol{x}_i)}\right]_{f=f_{m-1}}.$$

Then, we make the update

$$\boldsymbol{f}_m = \boldsymbol{f}_{m-1} - \beta_m \boldsymbol{g}_m,$$

where β_m is the step length and

$$\beta_m = \arg\min_{\beta} \mathcal{L}(\boldsymbol{f}_{m-1} - \beta \boldsymbol{g}_m).$$

Fitting a weak learner to approximate the negative gradient signal, i.e.

$$F_m = \arg\min_{F} \sum_{i=1}^{N} (g_{im} - F(\boldsymbol{x}_i))^2$$

allows us to learn a function that can generalize.

For classification, we can use log-loss, which leads to an algorithm called BinomialBoost, which does not need to be able to do weighted fitting.

With large datasets, we can use a stochastic variant which subsamples a random fraction of the data to pass to the regression tree at each iteration, which is called stochastic gradient boosting.

18.5.5.1 Gradient Tree Boosting

Gradient boosting nearly always assumes that the weak learner is a regression tree of the form

$$F_m(\boldsymbol{x}) = \sum_{j=1}^{J_m} w_{jm} \mathbb{I}(\boldsymbol{x} \in R_{jm}),$$

where w_{jm} is the predicted output for region R_{jm}. This is called gradient boosted regression trees or gradient tree boosting.

To use this, we find good regions R_{jm} for tree m using standard regression tree learning on the residuals and re-solve for the weights at each leaf by

$$\hat{w}_{jm} = \arg\min_{w} \sum_{\boldsymbol{x}_i \in R_{jm}} \ell(y_i, f_{m-1}(\boldsymbol{x}_i) + w).$$

18.5.5.2 XGBoost

XGBoost stands for extreme gradient boosting, and is an efficient implementation of gradient boosted trees. It adds a regularizer on the tree complexity, uses a second order approximation of the loss instead of a linear approximation, samples features at internal nodes, and uses other computer science methods to ensure scalability.

XGBoost optimizes the following

$$\mathcal{L}(f) = \sum_{i=1}^{N} \ell(y_i, f(\boldsymbol{x}_i)) + \Omega(f),$$

where

$$\Omega(f) = \gamma J + \frac{1}{2}\lambda \sum_{j=1}^{J} w_j^2$$

is the regularizer, where J is the number of leaves, and $\gamma \geq 0$ and $\lambda \geq 0$ are regularization coefficients.

18.6 Interpreting Tree Ensembles

Ensembles of trees lose the property of being interpretable, but there are some simple methods to interprete what function has been learned.

18.6.1 Feature Importance

A measure for feature importance of feature k is

$$R_k(T) = \sum_{j=1}^{J-1} G_j \mathbb{I}(v_j = k),$$

where the sum is over all non-leaf nodes, G_j is the gain in accuracy at node j, and $v_j = k$ if node j uses feature k. Averaging over all trees in the ensemble gives a more reliable estimate

$$R_k = \frac{1}{M} \sum_{m=1}^{M} R_k(T_m).$$

18.6.2 Partial Dependency Plots

A partial dependency plot for feature k is a plot of

$$\overline{f}_k(x_k) = \frac{1}{N} \sum_{n=1}^{N} f(\boldsymbol{x}_{n,-k}, x_k)$$

vs x_k. All features are marginalized out except k. For a binary classifier, convert to log odds $\log p(y = 1|x_k)/p(y = 0|x_k)$ before plotting.

Interaction effects between features j and k can be captured by computing

$$\overline{f}_{jk}(x_j, x_k) = \frac{1}{N} \sum_{n=1}^{N} f(\boldsymbol{x}_{n,-jk}, x_j, x_k).$$

6 Alzheimer's data Problem

```
# Libraries
library(tidymodels)
library(tidyverse)
library(hash)
library(doParallel)
library(ranger)
library(baguette)
library(xgboost)
library(mltools)
library(ggplot2)
```

```
# Read in data
alz <- read.csv("alzheimer_data.csv")
```

```
# Get Count Of Unique Results (if the count is low its probably a factor)
# (This is used below in the data cleaning section)

# The reason for this is because there are many variables that are integers
# that should be factors and I am too lazy to manually convert them to factors
# so this does it for me based on the threshold of unique values
```

```r
countResults <- hash()
for (var in colnames(alz)) {
  countResults[[var]] <- nrow(unique(alz[var]))
}

# Note from the results below it seems having unique elements less than 10
# is a good enough threshold to determine if a variable is a factor or not
print(countResults)
```

```
## <hash> containing 57 key-value pair(s).
##   age : 74
##   agitsev : 4
##   animals : 41
##   anxsev : 4
##   apasev : 4
##   appsev : 4
##   bills : 5
##   bpdias : 63
##   bpsys : 115
##   cdrglob : 5
##   csfvol : 2668
##   delsev : 4
##   depdsev : 4
##   diagnosis : 3
##   digif : 13
##   disnsev : 4
##   educ : 26
##   elatsev : 4
##   events : 5
##   female : 2
##   frcort : 2630
##   games : 5
##   hallsev : 4
##   height : 190
##   hrate : 73
##   id : 2700
##   irrsev : 4
##   lcac : 1924
##   lent : 1879
##   lhippo : 1740
##   lparcort : 2525
##   lparhip : 1750
##   lposcin : 1849
##   ltempcor : 2556
##   mealprep : 5
##   memunits : 26
##   motsev : 4
##   naccgds : 15
##   naccicv : 2237
##   naccmmse : 30
##   nitesev : 4
##   payattn : 5
##   rcac : 1800
##   remdates : 5
```

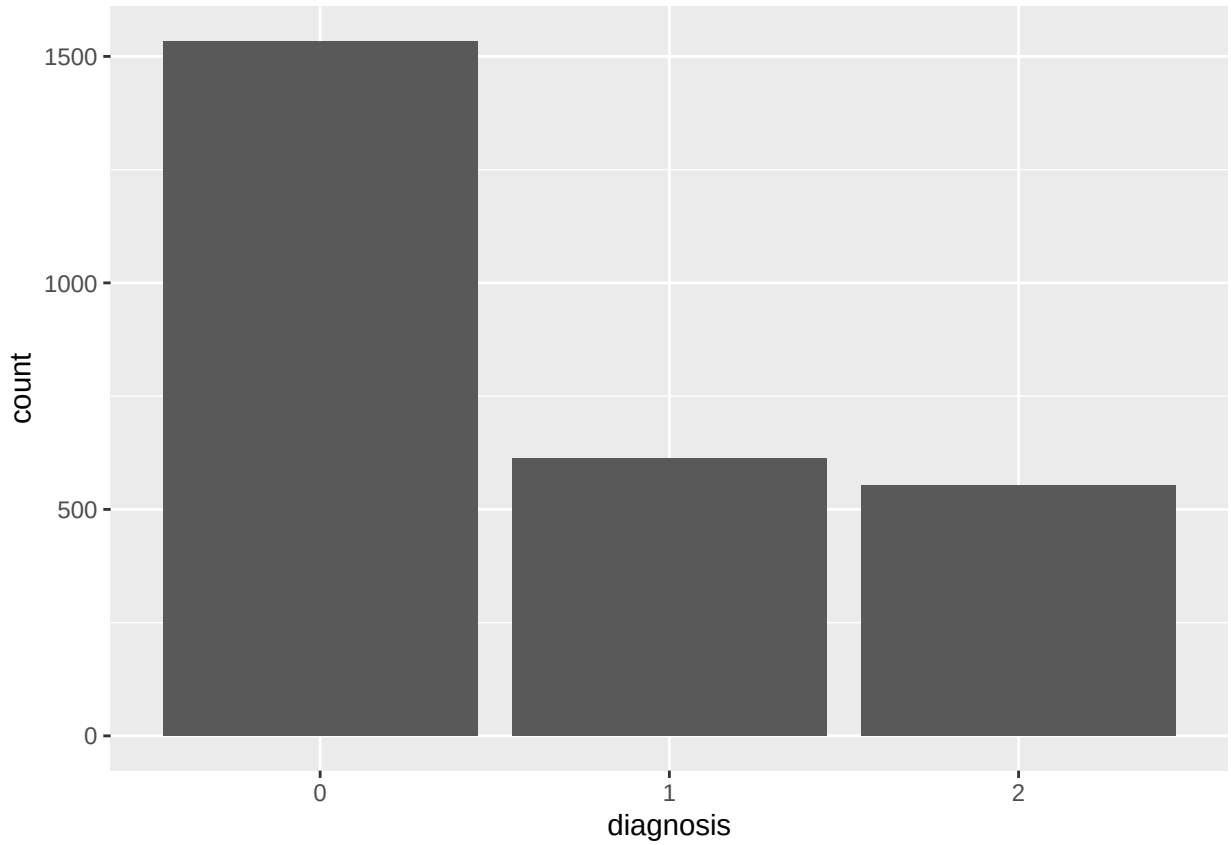
```
##   rent : 1862
##   rhippo : 1728
##   rparcort : 2520
##   rparhip : 1770
##   rposcin : 1864
##   rtempcor : 2527
##   shopping : 5
##   stove : 5
##   taxes : 5
##   traila : 130
##   trailb : 247
##   travel : 5
##   weight : 196
```

```
factoredVars <- character()
for (var in colnames(alz)) {
  if(nrow(unique(alz[var])) <= 10) {
    factoredVars <- append(factoredVars, var)
  }
}

print(factoredVars)
```

```
##  [1] "diagnosis" "female"    "cdrglob"   "delsev"    "hallsev"   "agitsev"
##  [7] "depdsev"   "anxsev"    "elatsev"   "apasev"    "disnsev"   "irrsev"
## [13] "motsev"    "nitesev"   "appsev"    "bills"     "taxes"     "shopping"
## [19] "games"     "stove"     "mealprep"  "events"    "payattn"   "remdates"
## [25] "travel"
```

```
# Data cleaning (mainly just converting ints to factors)
alz <- alz %>%
  select(-id) %>%
  mutate_at(factoredVars, factor)
```

```
### Some basic data exploration on a few predictors

#### Distribution of diagnosis
ggplot(data = alz, mapping = aes(x = diagnosis)) +
  geom_bar()
```

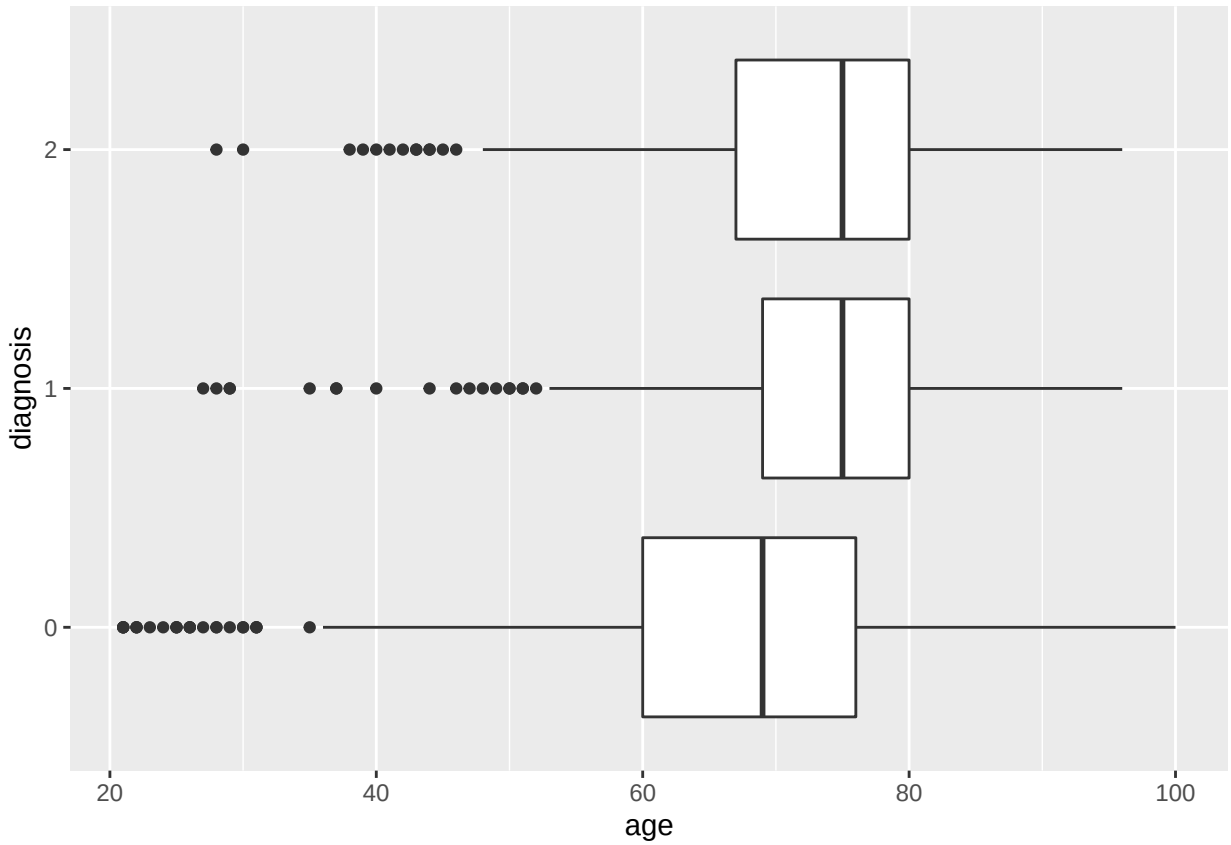



```
alz %>% group_by(diagnosis) %>% summarise(prop = n() / nrow(alz))
```

```
## # A tibble: 3 x 2
##   diagnosis  prop
##   <fct>     <dbl>
## 1 0         0.568
## 2 1         0.227
## 3 2         0.205
```

A normal (0) diagnosis is slightly more skewed in terms of frequency of occurrence then the other two diagnosis types.

```
# Effect of age on diagnosis
ggplot(data = alz, mapping = aes(x = age, y = diagnosis)) +
  geom_boxplot()
```

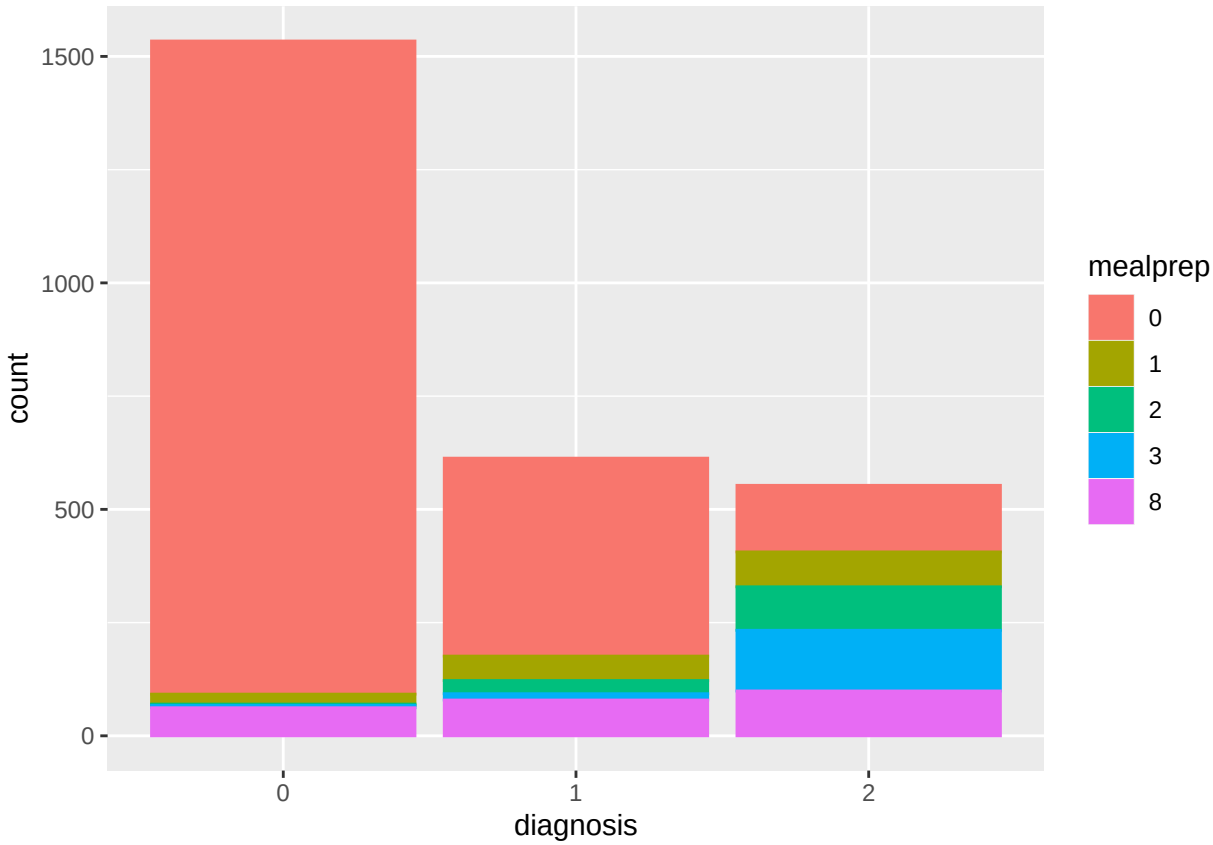



```
alz %>% group_by(diagnosis) %>%
  summarise(mean_age = mean(age), med_age = median(age), sd_age = sd(age))
```

```
## # A tibble: 3 x 4
##   diagnosis mean_age med_age sd_age
##   <fct>        <dbl>   <dbl>  <dbl>
## 1 0             67.6      69  12.0
## 2 1             73.7      75   9.97
## 3 2             72.9      75  10.3
```

Seems a diagnosis of normal (0) with regards to age is slightly skewed towards being younger which makes since alzhemiers is a progressive degenerative disease. This is probably a significant predictor.

```
# Effect of mealprep on diagnosis
ggplot(alz, aes(x = diagnosis, color = mealprep, fill = mealprep)) +
  geom_bar()
```



Mealprep of 0 (normal meal prep without assistance) is heavily skewed towards a diagnosis of normal (0) and the frequency of progressively worse meal prep scores increases as the diagnosis worsens implying mealprep may be a significant predictor.

```
# Split into training and testing
set.seed(123)
split <- initial_split(alz)
train_df <- training(split)
test_df <- testing(split)
```

Random Forest

```
# Random Forest

# The reason for not tuning trees is because it takes too long

# Define the specification for classification for random forest for
# tidymodels where mtry and min_n will be tuned
spec <- rand_forest(
  mtry = tune(),
  trees = 500,
  min_n = tune()) %>%
  set_mode("classification") %>%
  set_engine("ranger")

# Define a recipe for tidymodels
recipe <- recipe(diagnosis ~ ., data = train_df)
```

```r
# Define the workflow for tidymodels
wf <- workflow() %>%
  add_recipe(recipe) %>%
  add_model(spec)

# Define the folds for cross validated tuning
folds <- vfold_cv(train_df, v = 10)
```

```r
# Register parallel computation
registerDoParallel()

# Tune the model using the folds
tune_res <- tune_grid(
  wf,
  resamples = folds,
  grid = 10
)
```

```r
# Select the best model configuration based on roc area under curve
best_auc <- select_best(tune_res, "roc_auc")

# Finalize the best model using the best hyperparameters
final_rf <- finalize_model(
  spec,
  best_auc
)

final_rf
```

```
## Random Forest Model Specification (classification)
##
## Main Arguments:
##   mtry = 9
##   trees = 500
##   min_n = 17
##
## Computational engine: ranger
```

```r
set.seed(123)
# Finalize the workflow
final_wf <- workflow() %>%
  add_recipe(recipe) %>%
  add_model(final_rf)

# Finalize the fit
final_res <- final_wf %>%
  last_fit(split)

# Output accuracy and roc_auc
final_res %>%
  collect_metrics()
```

```
## # A tibble: 2 x 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
```

```
## 1 accuracy multiclass     0.836 Preprocessor1_Model1
## 2 roc_auc  hand_till      0.921 Preprocessor1_Model1
```

The tuned random forest model produces test misclassification rate of 0.1540741.

Bagging

```
# Bagging

# Define the specification for classification for bagging for tidymodels
# where min_n, tree_depth and cost complexity will be tuned
spec_bagging <- bag_tree(
  mode = "classification",
  cost_complexity = tune(),
  tree_depth = tune(),
  min_n = tune()) %>%
  set_engine("rpart")

# Add the recipe from above and the bagging specification
wf_bagging <- workflow() %>%
  add_recipe(recipe) %>%
  add_model(spec_bagging)
```

```
# Enable parrallel compute
registerDoParallel()

# Tune the hyperparameters for the bagging model
tune_res_bagging <- tune_grid(
  wf_bagging,
  resamples = folds,
  grid = 10
)
```

```
# Select the best bagging model based on roc area under curve
best_auc_bagging <- select_best(tune_res_bagging, "roc_auc")

# Finalize the model based on the best hyperparameters
final_bag <- finalize_model(
  spec_bagging,
  best_auc_bagging
)

final_bag
```

```
## Bagged Decision Tree Model Specification (classification)
##
## Main Arguments:
##   cost_complexity = 0.000114794375957125
##   tree_depth = 13
##   min_n = 27
##
## Computational engine: rpart
```

```
set.seed(123)
# Finalize the workflow
final_wf_bag <- workflow() %>%
```

```
  add_recipe(recipe) %>%
  add_model(final_bag)

# Finalize the fit
final_res_bag <- final_wf_bag %>%
  last_fit(split)

# Output accuracy and roc_auc
final_res_bag %>%
  collect_metrics()
```

```
## # A tibble: 2 x 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy multiclass     0.825 Preprocessor1_Model1
## 2 roc_auc  hand_till      0.911 Preprocessor1_Model1
```

The tuned bagging model produces test misclassification rate of 0.1688889.

Boosting

```
# Boosting

# The reason for not tuning trees is because it takes too long

# Define the specification for classification for boosting for tidymodels
# where mtry and min_n will be tuned
spec_boosting <- boost_tree(
  mode = "classification",
  mtry = tune(),
  trees = 500,
  min_n = tune()) %>%
  set_engine("xgboost")

# Since boosting (specifically xgboost) cant handle factors, convert the
# categorical variables into onehot encoded values
onehot_alz <- data.table::data.table(alz) %>%
  one_hot(cols = factoredVars[2:length(factoredVars)])

# Split into training and testing again with the onehot encoded data
set.seed(123)
split <- initial_split(onehot_alz)
train_df <- training(split)
test_df <- testing(split)

# Redefine folds, recipe and workflow
folds <- vfold_cv(train_df, v = 10)

recipe_boost <- recipe(diagnosis ~ ., data = train_df)

wf_boosting <- workflow() %>%
  add_recipe(recipe_boost) %>%
  add_model(spec_boosting)
```

```r
# Enable parallel compute
registerDoParallel()

# Tune the boosting model
tune_res_boosting <- tune_grid(
  wf_boosting,
  resamples = folds,
  grid = 10
)
```

```r
# Select the best boosting hyperparameters based on roc area under curve
best_auc_boosting <- select_best(tune_res_boosting, "roc_auc")

# Finalaize the model
final_boost <- finalize_model(
  spec_boosting,
  best_auc_boosting
)

final_boost
```

```
## Boosted Tree Model Specification (classification)
##
## Main Arguments:
##   mtry = 31
##   trees = 500
##   min_n = 6
##
## Computational engine: xgboost
```

```r
set.seed(123)
# Finalize the workflow
final_wf_boost <- workflow() %>%
  add_recipe(recipe_boost) %>%
  add_model(final_boost)

# Finalize the fit
final_res_boost <- final_wf_boost %>%
  last_fit(split)

# Output accuracy and roc_auc
final_res_boost %>%
  collect_metrics()
```

```
## # A tibble: 2 x 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy multiclass     0.844 Preprocessor1_Model1
## 2 roc_auc  hand_till      0.919 Preprocessor1_Model1
```

The tuned boosting model produces test misclassification rate of 0.157037

Comparing the three models: Random forest, bagging and boosting had misclassification rate of 0.1540741, 0.1688889, and 0.157037 respectively. Therefore, the Random Forest was the best model when comparing misclassification rate which means about 15.4% of predictions made were incorrect on the test set.