CHAPTER 7: BALANCE

SVM AND ENSEMBLE LEARNING

I. Overview

Chapter 6 is about "**balance**". As in Chapter 4 we have introduced the concept of overfitting, here, we further expand the issue of overfitting and introduce two famous models that provide two different approaches to address the overfitting issue. The two methods are the Support Vector Machine (SVM) and Ensemble Learning that includes RF as a particular case. The solutions provided in both methods to control the risk of overfitting are architectural, rather than some technical adjustments or implementation tricks.

II. Support Vector Machine

II.1 Rationale and Formulation

As we have learned that, the complexity of the final model should match the complexity of the signal embed in the noise. Overfitting happens when the model not only fits the signal part of the data, but also the noise part.

Overfitting could lead to promising results on the training data since the model actually "memorizes" the training data rather than generalizing the training data in the form as a model. Since the noise in the training data won't reappear in future unseen data, it is sure that the overfitted model won't perform well on future unseen data.

Thus, a question that appears in every practice of data analytics is, what model should I use? Is the model too simple? Or too complex?

Let's give this question a nice and specific context. Consider the classification problem that uses linear model to represent the decision boundary, as shown in below.

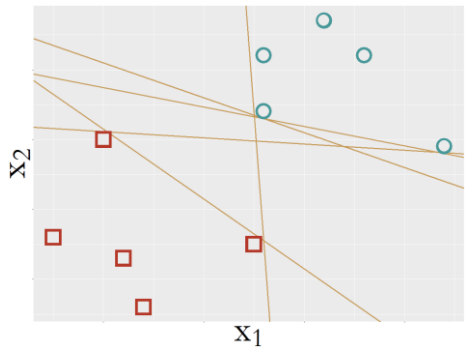


Figure 7.1: Which model (e.g., which line) should we use as our classification model to separate the two classes of data points?

From Figure 7.1, we can see that, if we only consider the classification error on the given data points of the two classes, it seems that all the models (represented as the lines) could achieve perfect classification. Thus, classification error is not sufficient in this case for us to decide on the optimal model. What else should we bring into the thought process?

As we have mentioned that, the objective of the model is to predict on future unseen data, now we may turn our attention from the given training data shown in Figure 7.1 to future unseen data. What could the future unseen data look like? Will all the models shown in Figure 7.1 perform equally well on the future unseen data?

It seems that, the lines that are close to the data points may bear a risk of performing bad on future unseen data. This is because that it seems to be very plausible that future red data points may allocate a little bit outside of the current region of the red data points, and thus, if we pick up the line that is close to red data points, the model may just misclassify the new red data points.

In other words, the lines that are too close to either side of the data points lack a safe **margin**. To reduce risk, we like to have the margin as large as possible. This is in the same spirit of risk management. This gave birth to the idea of SVM, as shown in Figure 7.2, where the model SVM suggests is the one that has the **maximum margin**.

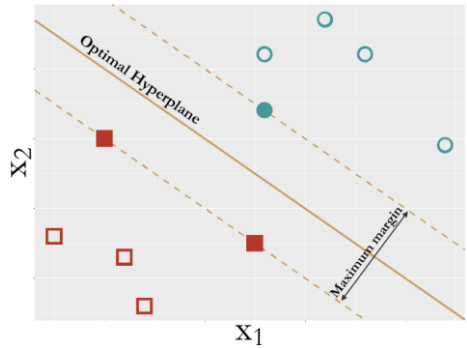


Figure 7.2: The model that has the maximum margin – the basic idea of SVM

II.2 Theory/Method

Derivation of the SVM formulation: Denote the training data points as $\{(\boldsymbol{x}_n, y_n), n = 1,2, \dots, N\}$. We are now ready to derive the mathematical framework corresponding to the idea shown in Figure 7.2. The goal is to identify a model, $\boldsymbol{w}^T\boldsymbol{x} + b$, using which we can make binary classification:

If $\boldsymbol{w}^T\boldsymbol{x} + b > 0$, then $y = 1$;

Otherwise, $y = -1$;

As we aim to maximize the margin, first, we need to be able to denote the margin mathematically in terms of the model parameters $\boldsymbol{w}$ and b.

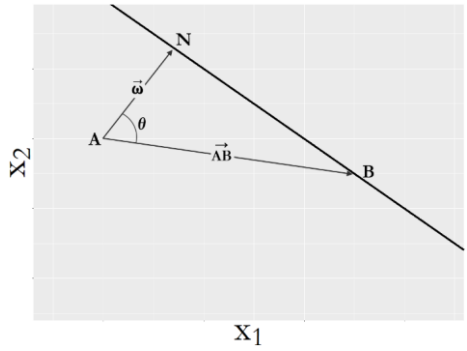


Figure 7.3: Illustration of how to derive the margin

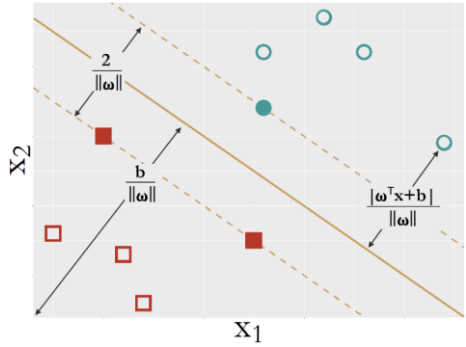


Figure 7.4: Formulation of the idea of maximum margin

Note that, as shown in Figure 7.3, for a data point A, its perpendicular distance to the line $\boldsymbol{w}^T\boldsymbol{x} + b = 0$ can be derive as:

$$\|AN\| = \|AB\| \cos\theta = \|AB\| \frac{\overrightarrow{AB}\cdot\overrightarrow{\boldsymbol{w}}}{\|AB\|\|\boldsymbol{w}\|} = \frac{\overrightarrow{AB}\cdot\overrightarrow{\boldsymbol{w}}}{\|\boldsymbol{w}\|}.$$

Denote the coordinates of the two data points A and B as $\boldsymbol{x}_a$ and $\boldsymbol{x}_b$, respectively. Then, we can see that

$$\frac{\overrightarrow{AB}\cdot\overrightarrow{\boldsymbol{w}}}{\|\boldsymbol{w}\|} = \frac{\boldsymbol{w}^T(\boldsymbol{x}_a - \boldsymbol{x}_b)}{\|\boldsymbol{w}\|}.$$

As the data point B is an arbitrary data point on the line $\boldsymbol{w}^T\boldsymbol{x} + b = 0$, it means that $\boldsymbol{w}^T\boldsymbol{x}_b = -b$. Thus, we can further derive that

$$\|AN\| = \frac{\boldsymbol{w}^T(\boldsymbol{x}_a - \boldsymbol{x}_b)}{\|\boldsymbol{w}\|} = \frac{\boldsymbol{w}^T\boldsymbol{x}_a + b}{\|\boldsymbol{w}\|}.$$

Now we can lay this derivation on Figure 7.2 to obtain Figure 7.4.

As shown in Figure 7.4, with a clear characterization of the margin of the data points to the decision line in terms of the model parameters $\boldsymbol{w}$ and $\boldsymbol{b}$, we still need more to write up the objective function of SVM that can maximizes the margin. We know that, as shown in Figure 7.2, the margin is only determined by $\boldsymbol{w}$, but it seems that in our derivation the margin also depends on the data points. Actually, this indicates that there is a numerical dimension to fix, as currently the whole formulation is underdetermined.

Thus, in the formulation of SVM, it was suggested to fix numerical scale of the model with a constraint:

$$|\boldsymbol{w}^T\boldsymbol{x}_n + b| = 1 \text{ for any } \boldsymbol{x}_n \text{ that is on the margin.}$$

With this fix, now we are ready to derive the margin in the SVM model as $\frac{2}{\|\boldsymbol{w}\|}$. To maximize the margin of the model is equivalent to minimize $\|\boldsymbol{w}\|$. This gives us the objective function of the SVM model.

Now let's derive the constraints of the SVM model. To derive the model from these training data points, obviously, we need to make sure the model can perform correctly on the training data. As the data points on the margin satisfy $|\boldsymbol{w}^T\boldsymbol{x}_n + b| = 1$, the data points that are beyond the margin will satisfy $|\boldsymbol{w}^T\boldsymbol{x}_n + b| > 1$.

Thus, the final SVM formulation is:

$$\min_{\boldsymbol{w}} \frac{1}{2}\|\boldsymbol{w}\|,$$

$$\text{Subject to: } y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \geq 1 \text{ for } n = 1,2,\dots,N.$$

To solve this problem, first, we can use the method of lagrange multiplier:

$$L(\boldsymbol{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\|\boldsymbol{w}\| - \sum_{n=1}^{N} \alpha_n[y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) - 1].$$

This could be rewritten as

$$L(\boldsymbol{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} - \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{w}^T \boldsymbol{x}_n - b\sum_{n=1}^{N} \alpha_n y_n + \sum_{n=1}^{N} \alpha_n.$$

Differentiating $L(\boldsymbol{w}, b, \boldsymbol{\alpha})$ with respect to $\boldsymbol{w}$ and b, and setting to zero yields:

$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n,$$
$$\sum_{n=1}^{N} \alpha_n y_n = 0.$$

Then, we can rewrite $L(\boldsymbol{w}, b, \boldsymbol{\alpha})$ as

$$L(\boldsymbol{w}, b, \boldsymbol{\alpha}) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m \boldsymbol{x}_n^T \boldsymbol{x}_m.$$

This is because that:

$$\frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} = \frac{1}{2}\boldsymbol{w}^T \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n = \frac{1}{2}\sum_{n=1}^{N} \alpha_n y_n \boldsymbol{w}^T \boldsymbol{x}_n =$$
$$\frac{1}{2}\sum_{n=1}^{N} \alpha_n y_n (\sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n)^T \boldsymbol{x}_n = \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m \boldsymbol{x}_n^T \boldsymbol{x}_m.$$

Then, finally, we can derive the model of SVM by solving its dual form problem:

$$\max_{\boldsymbol{\alpha}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m \boldsymbol{x}_n^T \boldsymbol{x}_m,$$

Subject to: $\alpha_n \geq 0$ for $n = 1,2,\ldots,N$ and $\sum_{n=1}^{N} \alpha_n y_n = 0$.

This is a **quadratic programming** problem that can be solved using many existing packages.

Note that, the learned model parameters could be represented as:

$\widehat{\boldsymbol{w}} = \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n$ and $\hat{b} = 1 - \widehat{\boldsymbol{w}}^T \boldsymbol{x}_n$ for any $\boldsymbol{x}_n$ whose $\alpha_n > 0$.

And we know that, based on the KKT condition of the SVM formulation, the following equations must hold:

$$\alpha_n[y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) - 1] = 0 \text{ for } n = 1,2,\ldots,N.$$

Thus, for any data point, e.g., the nth data point, it is either

$$\alpha_n = 0 \text{ or } y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) - 1 = 0.$$

Support vectors: Actually, this leads to the following interesting phenomenon, which leads to the definition of the "**support vectors**" as shown in Figure 7.5. The support vectors are what have been taken by the learning algorithm to constitute its decision function, which thus hold crucial implications for the SVM model. First, based on some theoretical

evidences, the number of support vectors is usually a metric that can indicate the healthiness of the model, i.e., the smaller the better. Second, it also reveals that the main statistical information the SVM model uses is from the support vectors. Thus, some works have been inspired by this aspect to accelerate the computation of SVM model training by discarding potentially non-support-vectors.

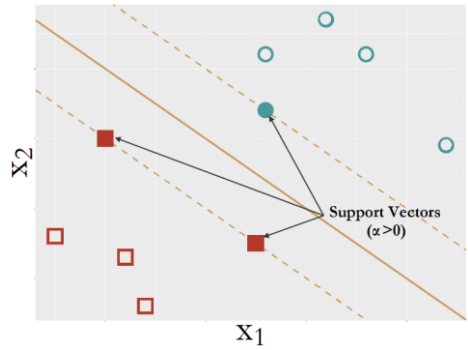


Figure 7.5: Support vectors of SVM are the data points that are on the margins

Extension to nonseparable cases: Note that, we have assumed that the two classes are separable. It is easy to relax this assumption. Ideally, in SVM, we hope that all the data points are either on or beyond the margin. We could relax this idealism and allow some data points to be within the margins or even on the wrong side of the decision line. To do so, we introduce the slack variables:

$$y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \geq 1 - \xi_n \text{ for } n = 1,2,\dots,N.$$

As shown in Figure 7.6, the data points that are within the margins will have the corresponding slack variables as $0 \leq \xi_n \leq 1$, and the data points that are on the wrong side of the decision line have the corresponding slack variables as $\xi_n > 1$.

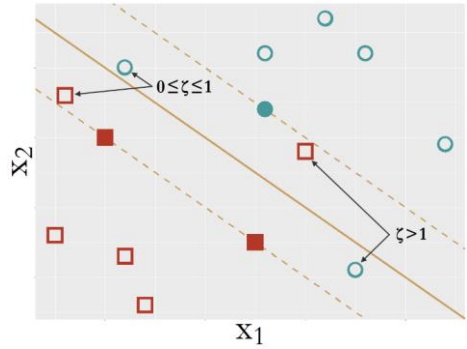
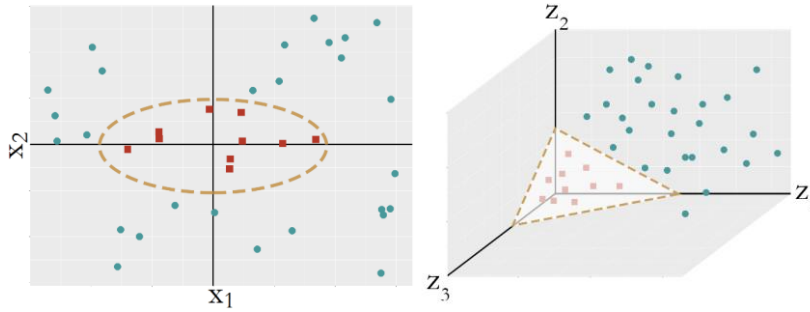


Figure 7.6: Behaviors of the slack variables



(a) (b)

Figure 7.7: (a) A linearly inseparable dataset; (b) with transformation (a) becomes separable

The corresponding formulation of the SVM model becomes:

$$\min_{\boldsymbol{w}} \frac{1}{2}\|\boldsymbol{w}\| + C\sum_{n=1}^{N}\xi_n,$$

Subject to: $y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \geq 1 - \xi_n$ and $\xi_n \geq 0$, for $n = 1,2,\ldots,N$.

Here, C is a user-specified parameter to control how much tolerance we can assign for the slack variables.

Extension to nonlinear SVM: So far we have presented SVM in linear models. Sometimes, the decision boundary could not be characterized as linear models, as shown in Figure 7.7 (a).

To create a nonlinear model within the framework of linear model, we could conduct transformation of the original variables. Here, we conduct the following transformation from $\boldsymbol{x}$ to $\boldsymbol{z}$:

$$z_1 = x_1^2,$$
$$z_2 = \sqrt{2}x_1x_2,$$
$$z_3 = x_2^2.$$

Then, in the new coordinates system, as shown in Figure 7.7 (b), the data points of the two classes become separable. This is the approach we often use in regression models as well, to create explicit transformation that asks us to write up how the features $\boldsymbol{z}$ could be represented as $\boldsymbol{x}$.

A remarkable thing about SVM is that, its formulation allows implicit transformation. This implicit transformation could be done by the use of kernel function. The dual formulation of SVM on the transformed variables is:

$$\max_{\boldsymbol{\alpha}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} \alpha_n\alpha_m y_n y_m \boldsymbol{z}_n^T\boldsymbol{z}_m,$$

Subject to: $0 \le \alpha_n \le C$ for $n = 1,2,\dots,N$ and $\sum_{n=1}^{N} \alpha_n y_n = 0$.

It can be seen that, the dual formulation of SVM shown above doesn't really need the information of individual $\boldsymbol{z}_n$. Rather, only the inner product of $\boldsymbol{z}_n^T\boldsymbol{z}_m$ is needed. As $\boldsymbol{z}$ is essentially functional of $\boldsymbol{x}$, i.e., $\boldsymbol{z} = \boldsymbol{\phi}(\boldsymbol{x})$, it can be seen that $\boldsymbol{z}_n^T\boldsymbol{z}_m$ is essentially function of $\boldsymbol{x}_n$ and $\boldsymbol{x}_m$. Thus, we can write it up as $\boldsymbol{z}_n^T\boldsymbol{z}_m = K(\boldsymbol{x}_n, \boldsymbol{x}_m)$. This is called the "**kernel function**". A kernel function is a function that theoretically entails a transformation $\boldsymbol{z} = \boldsymbol{\phi}(\boldsymbol{x})$ such that $K(\boldsymbol{x}_n, \boldsymbol{x}_m)$ implies that it can be written as an inner product $K(\boldsymbol{x}_n, \boldsymbol{x}_m) = \boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{\phi}(\boldsymbol{x})$. In other words, our effort now is not to seek explicit transformations that may be tedious and difficult, rather, we seek kernel functions that entail such transformations.

Nowadays we have had many such kernel functions to use. For example, the Gaussian radial basis kernel function is defined as

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = e^{-\gamma\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2},$$

where the transformation $\boldsymbol{z} = \boldsymbol{\phi}(\boldsymbol{x})$ is implicit and infinitely long to represent any smooth function.

The polynomial kernel function is defined as

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i^T \boldsymbol{x}_j + 1)^q.$$

Also, the linear kernel function is defined as

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i^T \boldsymbol{x}_j.$$

And there are still many new kernel functions to be developed to enrich our capacity of representing nonlinear decision boundaries in real-world applications. With a given kernel function, SVM learns the model by solving the following optimization problem:

$$\max_{\boldsymbol{\alpha}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m K(\boldsymbol{x}_n, \boldsymbol{x}_m),$$

Subject to: $0 \leq \alpha_n \leq C$ for $n = 1,2, \ldots, N$ and $\sum_{n=1}^{N} \alpha_n y_n = 0$.

However, in the kernel space, it will no longer to possible to write up the parameter $\boldsymbol{w}$ the same way as in linear models.

For any new data point, denoted as $\boldsymbol{x}_*$, the learned SVM model predict on it as

$$\text{If } \sum_{n=1}^{N} \alpha_n y_n K(\boldsymbol{x}_n, \boldsymbol{x}_*) + b > 0, \text{ then } y = 1;$$
$$\text{Otherwise, } y = -1.$$

II.3 R Lab

Let's try on an example. Consider a dataset:

$$\boldsymbol{x}_1 = (-1, -1), y_1 = -1;$$
$$\boldsymbol{x}_2 = (-1, +1), y_2 = +1;$$
$$\boldsymbol{x}_3 = (+1, -1), y_3 = +1;$$
$$\boldsymbol{x}_4 = (+1, +1), y_4 = -1.$$

We could use R to visualize this dataset. The R code is shown in below:

```
# Package installation
# pkgs <- c( 'ggplot2', 'kernlab', 'ROCR' )
# install.packages( pkgs )
# source( 'http://bioconductor.org/biocLite.R' )
# biocLite( 'ALL' )
```

```
# For the toy problem
x = matrix(c(-1,-1,1,1,-1,1,-1,1), nrow = 4, ncol = 2)
y = c(-1,1,1,-1)
linear.train <- data.frame(x,y)

# Visualize the distribution of data points of two classes
require( 'ggplot2' )

p <- qplot( data=linear.train, X1, X2, colour=factor(y),xlim = c
(-1.5,1.5), ylim = c(-1.5,1.5))
p <- p + labs(title = "Scatterplot of data points of two classes")
print(p)
```

The dataset is visualized in Figure 7.8. It is clear that the dataset presents a linearly inseparable problem, calling for the use of kernel to build nonlinear classification boundary.

Figure 7.8: A linearly inseparable dataset

Now, consider the kernel function, $K(\boldsymbol{x}_n, \boldsymbol{x}_m) = (\boldsymbol{x}_n^T \boldsymbol{x}_m + 1)^2$, which corresponds to the transformation:

$$\phi(\boldsymbol{x}_n) = \left[1, \sqrt{2}x_{n,1}, \sqrt{2}x_{n,2}, \sqrt{2}x_{n,1}x_{n,2}, x_{n,1}^2, x_{n,2}^2\right]^T.$$

The objective function becomes:

$$\max_{\boldsymbol{\alpha}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2}\sum_{n=1}^{4}\sum_{m=1}^{4} \alpha_n \alpha_m y_n y_m K(\boldsymbol{x}_n, \boldsymbol{x}_m),$$

Subject to: $\alpha_n \geq 0$ for $n = 1,2,\ldots,N$ and $\sum_{n=1}^{N} \alpha_n y_n = 0$.

We can calculate the kernel matrix as

$$\boldsymbol{K} = \begin{bmatrix} 9 & 1 & 1 & 1 \\ 1 & 9 & 1 & 1 \\ 1 & 1 & 9 & 1 \\ 1 & 1 & 1 & 9 \end{bmatrix}.$$

Then, we can solve the quadratic programming problem and get that

$$\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = 0.125.$$

In this particular case, as we can write up the transformation explicitly, we can write up $\widehat{\boldsymbol{w}}$ explicitly as:

$$\widehat{\boldsymbol{w}} = \sum_{n=1}^{4} \alpha_n y_n \phi(\boldsymbol{x}_n) = \left[0,0,0,1/\sqrt{2},0,0\right]^T.$$

Then, we can write up the decision function explicitly as:

$$f(\boldsymbol{x}_*) = \widehat{\boldsymbol{w}}^T \phi(\boldsymbol{x}_*) = x_{*,1} x_{*,2}.$$

As you can see, this is the decision boundary for a typical XOR problem. On the other hand, we can use R to build the SVM model on this dataset and see if our results could be reproduced in R. To do so, we use the R package "**kernlab**" and its function ksvm. The R code is shown in below:

```
# Train a linear SVM
x <- cbind(1, poly(x, degree = 2, raw = TRUE))
coefs = c(1,sqrt(2),sqrt(2),sqrt(2),1,1)
x <- x * t(matrix(rep(coefs,4),nrow=6,ncol=4))
linear.train <- data.frame(x,y)
require( 'kernlab' )

linear.svm <- ksvm(y ~ ., data=linear.train, type='C-svc', kernel
='vanilladot', C=10, scale=c())
```

The function **alpha()** returns the values of α_n for $n = 1,2,\ldots,N$. Here, note that, the function actually scaled the vector $\boldsymbol{\alpha}$. Thus, our manual results are consistent with the results obtained by using R.

```
alpha(linear.svm) #scaled alpha vector

## [[1]]
## [1] 0.2499619 0.2499619 0.2499873 0.2499873
```

Now let's do more examples. First, let's generate a dataset with linearly separable boundary.

```
# Generate a dataset with linear boundary
n <- 200
p <- 2
```

```
n.pos <- n/2
x.pos <-  matrix(rnorm( n*p, mean=0, sd=1 ),n.pos, p)
x.neg <-  matrix(rnorm( n*p, mean=2, sd=1), n-n.pos, p)
y <- c(rep(1, n.pos), rep(-1, n-n.pos))
n.train <- floor( 0.8 * n )
idx.train <- sample( n, n.train )
is.train <- rep( 0, n )
is.train[idx.train] <- 1
linear.data <- data.frame( x=rbind( x.pos, x.neg ), y=y, train=is.
train )
# Extract train and test subsets of the dataset
linear.train <- linear.data[linear.data$train==1, ]
linear.train <- subset( linear.train, select=-train )
linear.test <- linear.data[linear.data$train==0, ]
linear.test <- subset( linear.test, select=-train )
str(linear.train)

## 'data.frame':    160 obs. of  3 variables:
##  $ x.1: num  0.707 -0.92 0.87 -0.621 2.101 ...
##  $ x.2: num  -2.959 1.37 -0.526 0.989 0.833 ...
##  $ y  : num  1 1 1 1 1 1 1 1 1 ...

str(linear.test)

## 'data.frame':    40 obs. of  3 variables:
##  $ x.1: num  -1.857 -1.703 0.923 1.448 -0.723 ...
##  $ x.2: num  1.7934 0.0927 1.4485 1.1649 -0.1459 ...
##  $ y  : num  1 1 1 1 1 1 1 1 1 ...
```

We can visualize the data as shown in Figure 7.9.

```
# Visualize the distribution of data points of two classes
require( 'ggplot2' )
p <- qplot( data=linear.data, x.1, x.2, colour=factor(y) )
p <- p + labs(title = "Scatterplot of data points of two classes")
print(p)
```

We then use the `ksvm()` function to build the SVM model:

```
# Train a linear SVM
require( 'kernlab' )
linear.svm <- ksvm(y ~ ., data=linear.train, type='C-svc', kernel
='vanilladot', C=10, scale=c())
```

By typing in `linear.svm`, we can see more details of the built model. In this analysis, out of 200 data points, only 7 data points are needed to be the support vectors to define the linear boundary to separate two classes. This

is a good and healthy sign of the generalizability of the model to achieve robust success on unseen future data if the unseen future data would come from the same distribution of the training data.

We can also visualize the built model using the following R code, as shown in Figure 7.10. The black points shown in Figure 7.10 are the support vectors.

```
# Plot the model
plot( linear.svm, data=linear.train )
```

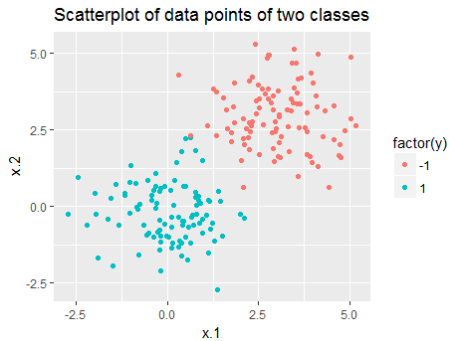


Figure 7.9: A randomly generated dataset with linearly separable boundary

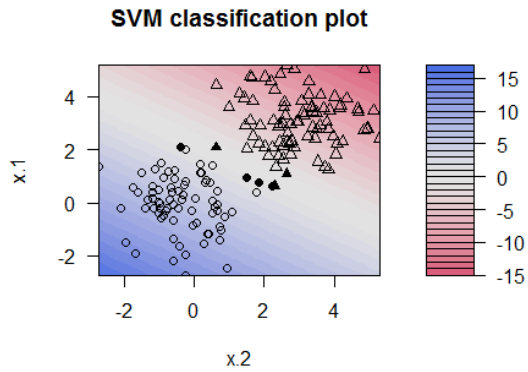


Figure 7.10: Visualization of the linear SVM model built for the simulated dataset

To verify our hypothesis that the model would obtain robust performance on unseen testing data, here, we present the ROC curve of the linear SVM model on the testing data that is not used in training the model.

```
# Generate the ROC curve using the testing data
# Prediction scores
linear.prediction.score <- predict(linear.svm, linear.test, type=
'decision')
# Compute ROC and Precision-Recall curves
require( 'ROCR')

linear.roc.curve <- performance( prediction( linear.prediction.sc
ore, linear.test$y ),
                                 measure='tpr', x.measure='fpr' )
plot(linear.roc.curve,  lwd = 2, col = "orange3",
     main = "Validation of the linear SVM model using testing dat
a")
```

As shown in Figure 7.11, indeed, the ROC curve shows the linear SVM model predicts well on the testing data.

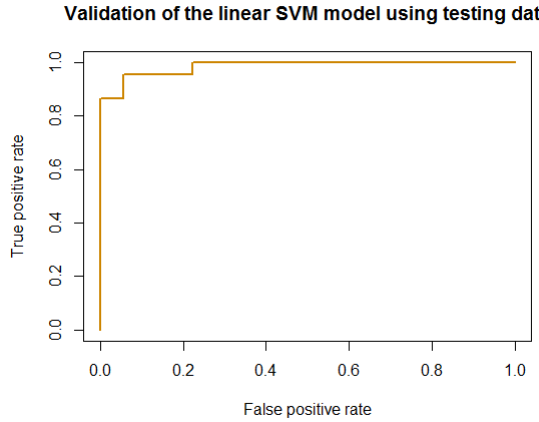


Figure 7.11: The ROC curve of the linear SVM model on testing data

On the other hand, similarly as what we have discussed in Chapter 5, in practice we will not use a testing data to guide the model selection. Thus,

the performance of the trained model has to be evaluated using the training data.

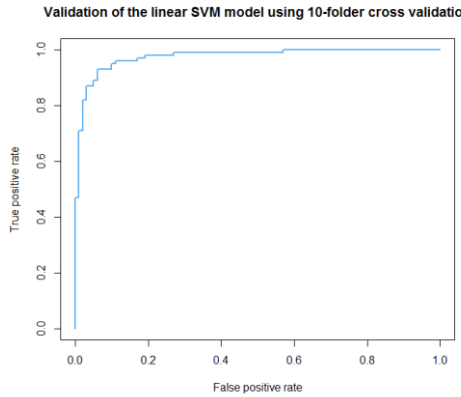


Figure 7.12: The ROC curve of the linear SVM model by 10-folder cross-validation

To achieve this, the cross-validation has been shown in Chapter 5 that can approximate the testing performance of the model. Here, again, we use this example to show that indeed the cross-validation provides such an effective way. The R code is shown in below:

```
# Generate the ROC curve using 10-folder cross validation
n <- nrow(linear.data)
n.folds=10
idx <- split(sample(seq(n)), seq(n.folds))
scores <- rep(0, n)
for(i in seq(n.folds)) {
  model <- ksvm(y ~ ., data=linear.data[-idx[[i]], ], kernel='van
illadot', C=100 )
  scores[idx[[i]]] <- predict( model, linear.data[idx[[i]],], typ
e='decision' )
}

plot(performance(prediction(scores, linear.data$y), measure='tpr',
 x.measure='fpr' ),
     lwd = 2, col = "steelblue2",
     main = "Validation of the linear SVM model using 10-folder c
ross validation")
```

It can be observed that the ROC curve presented in Figure 7.12 approximates the ROC curve presented in Figure 7.11 well, showing that the ROC curve by 10-folder cross-validation is a good estimation of the ROC curve obtained on a testing dataset.

Since the 10-folder cross-validation could be used to obtain the prediction performance of any given model, it gives rise to the possibility that we could use it to compare different model formulations and decide on which model is the best. To do so, the R package "**caret**" could be used that has an automatic procedure dedicated for this.

```
# Cross-validation using caret pacakge
# install.packages("caret")
# install.packages("pROC")
# Training SVM Models
require(caret)

require(kernlab)       # support vector machine
require(pROC)          # plot the ROC curves

# Setup for cross validation
ctrl <- trainControl(method="repeatedcv",   # 10fold cross valida
tion
                     repeats=5,         # do 5 repititions of cv
                     summaryFunction=twoClassSummary,   # Use AUC
 to pick the best model
                     classProbs=TRUE)

#Train and Tune the SVM
linear.train <- data.frame(linear.train)
trainX <- linear.train[,1:2]
trainy= linear.train[,3]
trainy[which(trainy==1)] = rep("T",length(which(trainy==1)))
trainy[which(trainy==-1)] = rep("F",length(which(trainy==-1)))
svm.tune <- train(x = trainX,
                  y = trainy,
                  method = "svmLinear",   # Linear kernel
                  tuneLength = 9,                    # 9 values of
 the cost function
                  preProc = c("center","scale"),  # Center and sc
ale data
                  metric="ROC",
                  trControl=ctrl)

svm.tune
```

Then we can obtain that:

```
svm.tune

## Support Vector Machines with Linear Kernel
##
## 160 samples
##   2 predictor
##   2 classes: 'F', 'T'
##
## Pre-processing: centered (2), scaled (2)
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 144, 144, 144, 145, 143, 144, ...
## Resampling results:
##
##   ROC        Sens       Spec
##   0.9625198  0.8964286  0.9055556
##
## Tuning parameter 'C' was held constant at a value of 1
```

While "`caret`" provides an automatic but sealed process to help us directly arrive the final end, the R package "`manipulate`" could be used to visualize the intermediate process. Here, we take some snapshots of this dynamic and interactive process and present these snapshots in Figures 7.13 and 7.14. It can be seen that, with larger value of C, a tighter margin could be obtained with less support vectors.

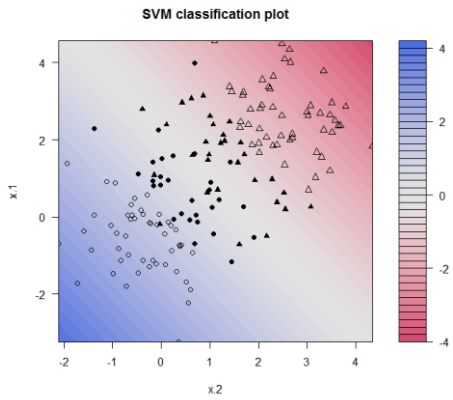


Figure 7.13: Visualization of the linear SVM model with C = 0.01.

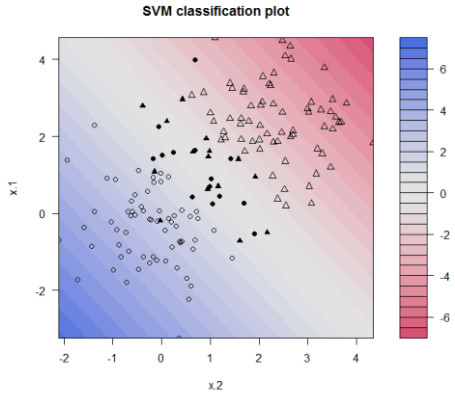


Figure 7.14: Visualization of the linear SVM model with C = 10.

Let's further consider a nonlinear dataset.

```
# Generate a dataset with nonlinear boundary
n = 100
p = 2
bottom.left <- matrix(rnorm( n*p, mean=0, sd=1 ),n, p)
upper.right <- matrix(rnorm( n*p, mean=4, sd=1 ),n, p)
tmp1 <- matrix(rnorm( n*p, mean=0, sd=1 ),n, p)
tmp2 <- matrix(rnorm( n*p, mean=4, sd=1 ),n, p)
upper.left <- cbind( tmp1[,1], tmp2[,2] )
bottom.right <- cbind( tmp2[,1], tmp1[,2] )
y <- c( rep( 1, 2 * n ), rep( -1, 2 * n ) )
idx.train <- sample( 4 * n, floor( 3.5 * n ) )
is.train <- rep( 0, 4 * n )
is.train[idx.train] <- 1
nonlinear.data <- data.frame( x=rbind( bottom.left, upper.right,
upper.left, bottom.right ), y=y, train=is.train )

# Visualize the distribution of data points of two classes
require( 'ggplot2' )
p <- qplot( data=nonlinear.data, x.1, x.2, colour=factor(y) )
p <- p + labs(title = "Scatterplot of data points of two classes")
print(p)
```

As shown in Figure 7.15, this nonlinear dataset is similar to the XOR problem we have shown in the beginning of this section, in the sense that a similar style of classification boundary is needed.

Figure 7.15: A randomly generated dataset with nonlinear boundary

Then, let's use the `kernlab` with Gaussian kernel (denoted as "`svmRadial`") and the 10-folder cross-validation procedure in "**`caret`**" to train a nonlinear SVM model.

```
# Use cross-validation to choose C
# install.packages("caret")
# install.packages("pROC")
# Training SVM Models
require(caret)
require(kernlab)        # support vector machine
require(pROC)           # plot the ROC curves
# Setup for cross validation
ctrl <- trainControl(method="repeatedcv",   # 10fold cross valida
tion
                     repeats=1,         # do 5 repititions of cv
                     summaryFunction=twoClassSummary,   # Use AUC
 to pick the best model
                     classProbs=TRUE)

#Train and Tune the SVM
nonlinear.train <- data.frame(nonlinear.train)
trainX <- nonlinear.train[,1:2]
trainy= nonlinear.train[,3]
trainy[which(trainy==1)] = rep("T",length(which(trainy==1)))
trainy[which(trainy==-1)] = rep("F",length(which(trainy==-1)))
svm.tune <- train(x = trainX,
```

```
                  y = trainy,
                  method = "svmRadial",   # Radial kernel
                  tuneLength = 9,                    # 9 values of
 the cost function
                  preProc = c("center","scale"),  # Center and sc
ale data
                  metric="ROC",
                  trControl=ctrl)

svm.tune
```

Details of the model tuning by the cross-validation process is shown in below:

```
## Support Vector Machines with Radial Basis Function Kernel
##
## 350 samples
##   2 predictor
##   2 classes: 'F', 'T'
##
## Pre-processing: centered (2), scaled (2)
## Resampling: Cross-Validated (10 fold, repeated 1 times)
## Summary of sample sizes: 315, 315, 315, 315, 315, 315, ...
## Resampling results across tuning parameters:
##
##   C      ROC        Sens       Spec
##    0.25  0.9878913  0.9584559  0.9502924
##    0.50  0.9869174  0.9584559  0.9502924
##    1.00  0.9872463  0.9643382  0.9502924
##    2.00  0.9826561  0.9525735  0.9558480
##    4.00  0.9797171  0.9584559  0.9502924
##    8.00  0.9754708  0.9584559  0.9558480
##   16.00  0.9735144  0.9525735  0.9447368
##   32.00  0.9709086  0.9466912  0.9502924
##   64.00  0.9659980  0.9290441  0.9391813
##
## Tuning parameter 'sigma' was held constant at a value of 1.834
54
## ROC was used to select the optimal model using  the largest va
lue.
## The final values used for the model were sigma = 1.83454 and C
 = 0.25.
```

Again, we can use the package "**manipulate**" to see how the model changes according to the parameters such as C and even kernel types.

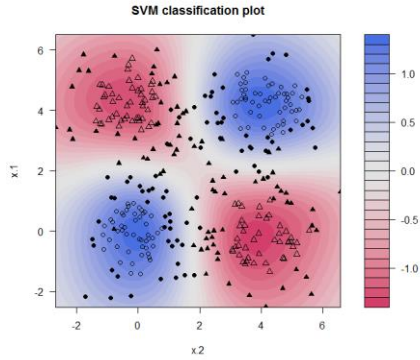


Figure 7.16: SVM model with C = 0.01 and Gaussian kernel

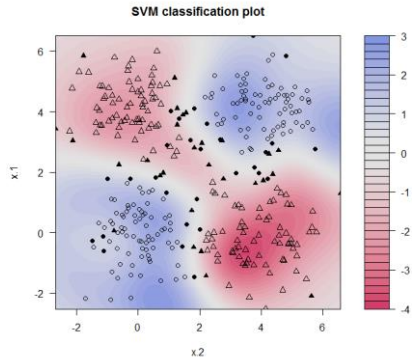


Figure 7.17: SVM model with C = 10 and Gaussian kernel

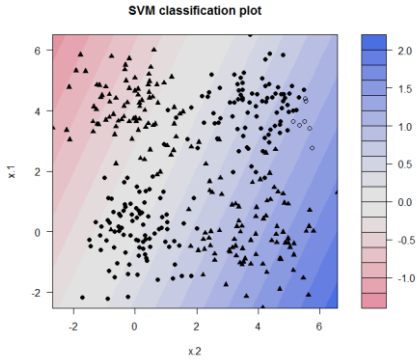


Figure 7.18: SVM model with C = 0.1 and Laplacian kernel

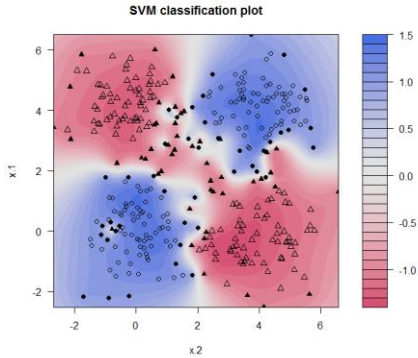


Figure 7.19: SVM model with C = 10 and Laplacian kernel

Finally, let's implement the above process on the AD dataset.

```
#### Dataset of Alzheimer's Disease
#### Objective: prediction of diagnosis
# filename
AD <- read.csv('AD_bl.csv', header = TRUE)
str(AD)

#Train and Tune the SVM
n = dim(AD)[1]
n.train <- floor(0.8 * n)
idx.train <- sample(n, n.train)
AD[which(AD[,1]==0),1] = rep("Normal",length(which(AD[,1]==0)))
AD[which(AD[,1]==1),1] = rep("Diseased",length(which(AD[,1]==1)))
AD.train <- AD[idx.train,c(1:16)]
AD.test <- AD[-idx.train,c(1:16)]
trainX <- AD.train[,c(2:16)]
trainy= AD.train[,1]

# Setup for cross validation
ctrl <- trainControl(method="repeatedcv",   # 10fold cross valida
tion
                     repeats=1,         # do 5 repititions of cv
                     summaryFunction=twoClassSummary,   # Use AUC
 to pick the best model
                     classProbs=TRUE)

# Use the expand.grid to specify the search space
grid <- expand.grid(sigma = c(0.002, 0.005, 0.01, 0.012, 0.015),
                    C = c(0.3,0.4,0.5,0.6)
)
```

```
svm.tune <- train(x = trainX,
                  y = trainy,
                  method = "svmRadial",   # Radial kernel
                  tuneLength = 9,                   # 9 values of
 the cost function
                  preProc = c("center","scale"),  # Center and sc
ale data
                  metric="ROC",
                  tuneGrid = grid,
                  trControl=ctrl)

svm.tune
```

Then we can obtain the following results.

```
## Support Vector Machines with Radial Basis Function Kernel
##
## 413 samples
##  15 predictor
##   2 classes: 'Diseased', 'Normal'
##
## Pre-processing: centered (15), scaled (15)
## Resampling: Cross-Validated (10 fold, repeated 1 times)
## Summary of sample sizes: 371, 372, 372, 371, 372, 372, ...
## Resampling results across tuning parameters:
##
##   sigma  C    ROC        Sens       Spec
##   0.002  0.3  0.8929523  0.9121053  0.5932900
##   0.002  0.4  0.8927130  0.8757895  0.6619048
##   0.002  0.5  0.8956402  0.8452632  0.7627706
##   0.002  0.6  0.8953759  0.8192105  0.7991342
##   0.005  0.3  0.8965129  0.8036842  0.8036797
##   0.005  0.4  0.8996565  0.7989474  0.8357143
##   0.005  0.5  0.9020830  0.7936842  0.8448052
##   0.005  0.6  0.9032422  0.7836842  0.8450216
##   0.010  0.3  0.9030514  0.7889474  0.8541126
##   0.010  0.4  0.9058248  0.7886842  0.8495671
##   0.010  0.5  0.9060999  0.8044737  0.8541126
##   0.010  0.6  0.9077848  0.8094737  0.8450216
##   0.012  0.3  0.9032308  0.7781579  0.8538961
##   0.012  0.4  0.9049043  0.7989474  0.8538961
##   0.012  0.5  0.9063505  0.8094737  0.8495671
##   0.012  0.6  0.9104511  0.8042105  0.8586580
##   0.015  0.3  0.9060412  0.7886842  0.8493506
##   0.015  0.4  0.9068165  0.8094737  0.8495671
##   0.015  0.5  0.9109051  0.8042105  0.8541126
##   0.015  0.6  0.9118615  0.8042105  0.8632035
```

```
##
## ROC was used to select the optimal model using  the largest va
lue.
## The final values used for the model were sigma = 0.015 and C =
 0.6.
```

It can be seen that, by 10-folder cross-validation, the best model parameters are `sigma = 0.015` and `C = 0.6`, which achieve a prediction performance as 90.49%.

II.4 Remarks

Is SVM a more complex model? Here, we need to look at the term "complexity" carefully. What is a complex model? Comparing with linear regression model or logistic regression model, the idea of SVM and formulation of SVM indeed seems more complex. This is probably true. The inventor of SVM, Vladimir Vapnik, once said in the preface of his seminar book[1] that delineates the theory of SVM, that he often heard peers talked in conferences that complex models don't work but simple models work. He thought, SVM, commonly perceived as a more complex model, is essentially simpler than some simple model. This is true, since some models that look simple are only because they presuppose stronger conditions, which make them essentially more complex!

Thus, a model is more complex than another model doesn't necessary stem from the fact that the more complex one employs a more sophisticated mathematical representation. At least, in our current context, it doesn't mean in this way when we say a model is complex. Here, we say that a model is more complex if it provides more capacity to represent the statistical phenomena in the training data. In other words, a more complex model is more flexible of responding to any patterns in the data by adjusting itself. Now, look at Figure 7.20, which model is simpler?

[1] *Vapnik, V. The nature of statistical learning theory. Springer, 2000.*

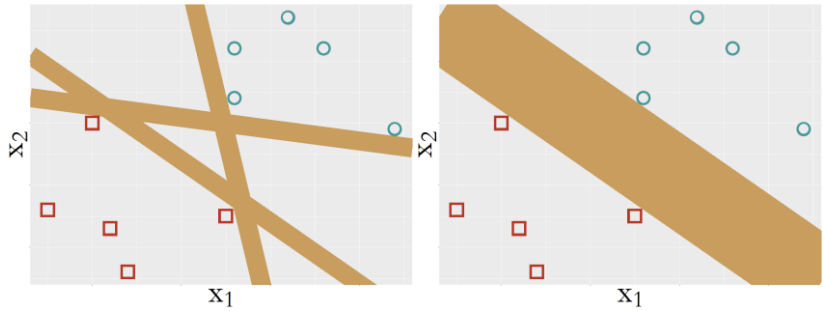


Figure 7.20: SVM is actually a simpler model

Is SVM a neural network model? Another interesting fact about SVM is that, when it was developed, it was named as "support vector network". In other words, it has a connection with the artificial neural network. This is revealed in the Figure 7.21. Readers who know neural network models are encouraged to write up the mathematical model of the SVM model following the neural network format as shown in Figure 7.21.

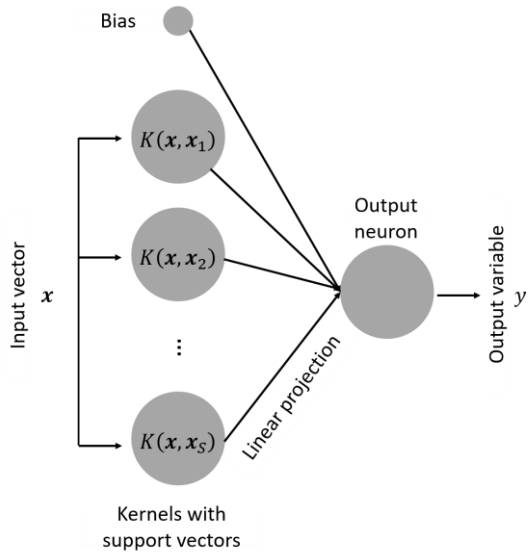


Figure 7.21: SVM as a neural network model

III. Ensemble Learning

III.1 Rationale and Formulation

As we mentioned in the begining of this chapter, the random forest model is a particular case of the more general category of models that are called ensemble models. Ensemble models consist of multiple base models, denoted as, $h_1, h_2, \ldots, h_K$, where K is the total number of base models. Each model can be considered as a **hypothesis** in the space of $\mathcal{H}$ that includes all the possible models. A general framework of ensemble learning is illustrated in Figure 7.22. Each model is built on a sample that is created from the original dataset. Recall that, in random forest models, each tree is built on an independently bootstrapped sample (also referred to as bagging), but this is not the only approach we can create a new dataset from the original dataset. For instance, in Adaboosting, the sample for a base model is not independently created. Rather, it also depends on the error rates from previous base models. In other words, it takes an adapative and sequential approach to grow its base models, while later models more focus on the hard data points that present challenges for previous base models to achieve good prediction performance.

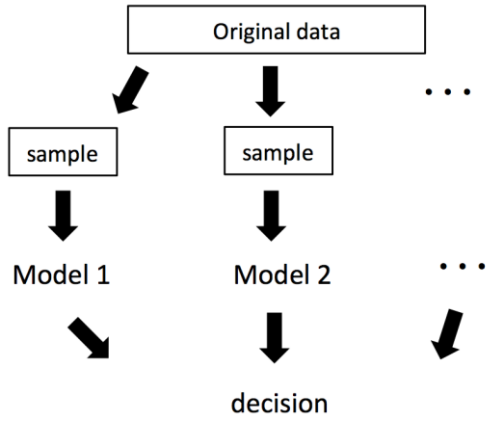


Figure 7.22: A general framework of ensemble learning

III.2 Theory/Method

As shown in Figure 7.22, the ensemble learning is very flexible as different approaches could be combined to create new samples and build new base models. It has been known that ensemble learning is very powerful in practices and has been reported as the winner methods in numerous data science competitions. Just like SVM, it is another main approach to handle the risk of overfitting in practice. Here, we use the framework proposed by Dietterich (2000)[1], where three perspecitves (statistical, computational, and representational) were articulated to explain why ensemble methods could lead to excellent accuracy performance. Each perspecitve is described in detail below.

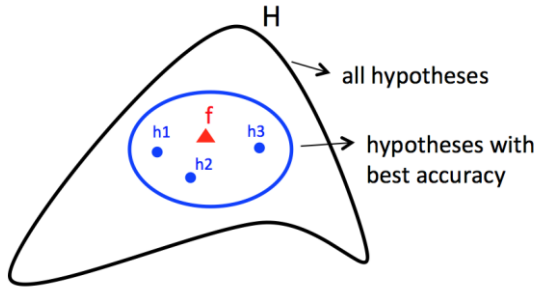


Figure 7.23: Ensemble learning approximates the true model with a combination of good models (statistical perspective)

Statistical perspective: The statistical reason is illustrated in Figure 7.23. $\mathcal{H}$ is the hypothesis space where a learning algorithm searches for the best one guided by the training data. f is the true function. The statistical problem occurs when there are limited data, and there are multiple best hypotheses. In other words, multiple models can achieve the best accuracy on the training data. This is illustrated by the inner circle in Figure 7.23. By

[1] *Dietterrich, T.G. Ensemble methods in machine learning. Multiple classifier systems, 2000.*

building an ensemble of multiple learners, e.g., h_1, h_2, and h_3, the average of the hypotheses is a good approximation to the true hypothesis f. Therefore, the average of the multiple hypotheses essentially approximates a solution with the minimum variance in the inner circle.

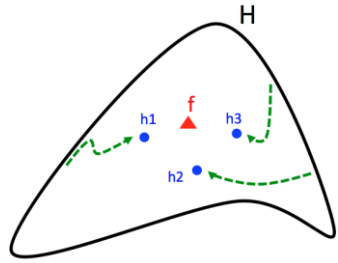


Figure 7.24: Ensemble learning approximates the true model with a combination of good models (computational perspective)

Computational perspective: A computational perpective is shown in Figure 7.24. This is related to the way we build base models, which is usually a greedy approach such as the recursive splitting procedure we have shown in decision tree models. Many machine learning models are complex models that present NP-hard optimization problems in training these models including neural networks and decision tree models. E.g., in decision trees, at each node, the node is split according to the maximum information gain. However, only the current node is evaluated, and it may result in suboptimal situations for further splitting of descendant nodes. Growing an optimal tree model is thus NP-hard and computationally expensive. This computational prospective is shown in Figure 7.24, while the learning algorithm of greedy and heuristic nature with a certain parameter setting searches for the best hypothesis in the hypothesis space. The search paths of three hypotheses are illustrated in Figure 7.24. The differences of the three hypotheses can be attributed to different parameter settings or different input data (e.g., bootstrap samples). However, growing

multiple learning algorithms and averaging them in joint decision makings, may reasonably approximate the true hypothesis f.

Representational perspective: Due to the size of the dataset or the limitations of a learning algorithm, sometimes the hypothesis space $\mathcal{H}$ does not cover the true function, as shown in Figure 7.25. For example, linear models cannot learn non-linear patterns, and decision trees with limited data have difficulty learning linear patterns. Using a weighted sum of the outcomes from the base learners may be able to approximate a function outside $\mathcal{H}$. This is shown in Figure 7.25, while the true function f is outside the space, but a combination of h_1, h_2 and h_3 can approximate the true function.

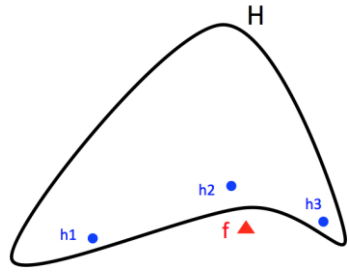


Figure 7.25: Ensemble learning approximates the true model with a combination of good models (representional perspective)

Now we discuss three methods, single decision trees, random forests, and AdaBoost (adaptive boosting) under the framework of ensemble learning.

Single decision tree: A single decision tree lacks capability to overcome overfitting in terms of all the three perspectives. From the statistical perspective, a decision tree algorithm constructs each node using the maximum information gain and is sensitive to the training data. While the training dataset is limited, the possible models achieving the best

accuracy can be large. Consequently, the learning algorithm may end up with any one of these models, which maybe far away from the true hypothesis.

Single decision trees also have the computational issue. Decision trees are greedy implementations, seeking for maximum impurity gain at each node. Decisions made in upstream nodes would affect downstream nodes very much. And it is NP-hard to find an optimal decision tree which achieves the maximum impurity gain at all nodes.

Representational perspective also shows limitations of the decision tree model. Although decision trees can approximate a wide range of functions, it needs sufficient data to achieve an accurate approximation. Given limited training data, the possible hypothesis space of a single decision tree may not be able to cover the true function, e.g., if the true function is a linear or any smooth function.

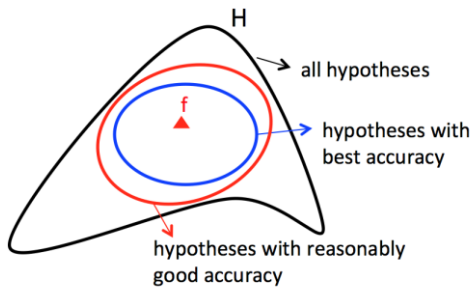


Figure 7.26: Analysis of the random forest in terms of the statistical perspective

Random forests: Random forests construct multiple hypotheses by two ways. First, each tree in random forests is built on a different bootstrapped sample. Actually, this framework that builds each base learner based on a bootstrapped sample is referred to as bagging in general. Second, at each node, a subset of variables is randomly selected and the best variable from the subset is used for splitting. It addresses the statistical issue. Note that, each tree is injected with randomness, and therefore, is not necessarily in

the best-accuracy hypothesis circle, but lies in the hypothesis space with reasonably good accuracy. Averaging the outcomes from all trees (or hypotheses) would achieve the minimum variance in the reasonably-good-accuracy space. Assuming the best-accuracy space has a similar shape with the reasonably-good space just with a smaller size, the solution may also achieve a reasonably small variance in the best-accuracy space. This is illustrated in Figure 7.26.

Random forests can also address the computational issue. As shown in Figure 7.27, the inner circle represents the space that is computationally difficult to reach. However, averaging multiple hypotheses could get into the inner space.

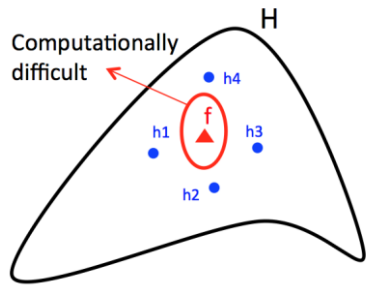


Figure 7.27: Analysis of the random forest in terms of the computational perspective

Random forests do not necessarily, or actively, solve the representational issue. If the true function lies outside $\mathcal{H}$, averaging the outcomes from all trees won't necessarily approximate the true function. Figure 7.28 shows that random forests would construct multiple hypotheses that randomly spread over the $\mathcal{H}$ space. Averaging them won't reach the true function f.

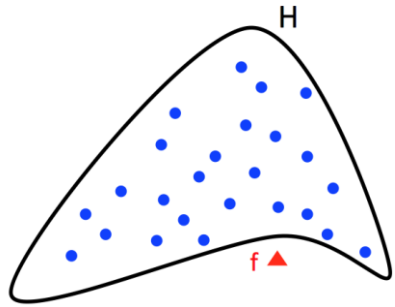


Figure 7.28: Analysis of the random forest in terms of the representional perspective

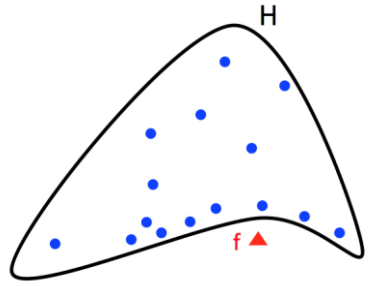


Figure 7.29: Analysis of the AdaBoost in terms of the representional perspective

AdaBoost: Unlike random forests that build each tree independently, AdaBoost builds trees sequentially. For each tree, the training dataset is sampled not by bootstrap, but by a weight determined by the error rates from previous trees. Data points that are difficult to be correctly predicted by the previous trees will be given more weights in the new training dataset for new trees. When all the base learners are trained, the aggregation of these models in predicting on a data instance $\boldsymbol{x}$ is a weighted sum of base learners

$$h(\boldsymbol{x}) = \sum_i w_i h_i(\boldsymbol{x}),$$

where the weight w_i is determined by the accuracy of learner $h_i(\boldsymbol{x})$.

Similar to random forests, AdaBoost solves the computational issue by generating many base learners that are built on randomly generated datasets. Different from random forests, AdaBoost actively solves the representational issue as it tries to reduce the residual errors coming from previous trees. Figure 7.29 shows that, AdaBoost could construct more hypotheses around the true function, and also could put more weight to the hypotheses that are closer to the true function by using the weighted sum of base learners in aggregation of the base learners.

But AdaBoost is not as good as random forests in terms of addressing the statistical issue. AdaBoost handles the overfitting issue through the concept of margin. Suppose each data point is labeled as -1 or 1, then the margin of a classifier of a data point (x_i, y_i) is defined as

$$m_i = y_i h(x_i).$$

It can be shown that AdaBoost tries to minimize

$$\sum_i \exp(-y_i \sum_j w_j h_j(x_j)),$$

which can be considered as an effort to minimize the margin on the training data. However, as AdaBoost aggressively solves the representational issue, and optimizes for the training data, it is more likely to overfit, and may be less stable than random forests that place more emphasis on addressing the statistical issue.

III.3 R Lab

A single decision tree (`rpart`), random forests (`randomForest`), and AdaBoost (`gbm`) are applied to the AD dataset `AD_bl.csv`. First, we change the percentage of training data, and 50 replicates of data are generated. The boxplots of the classification error rates for single decision tree, random forests, and AdaBoost are plotted at different percentages of training data by the following R code, which is shown in Figure 7.30.

```
library(dplyr)
library(tidyr)
library(ggplot2)
require(randomForest)
require(gbm)
require(rpart)
```

```
set.seed(1)

theme_set(theme_gray(base_size = 15))

path <- "../../data/AD_bl.csv"
data <- read.csv(path, header = TRUE)
rm_indx <- which(colnames(data) %in% c("ID", "TOTAL13", "MMSCORE
"))
data <- data[, -rm_indx]
data$DX_bl <- as.factor(data$DX_bl)
# target_indx <- which( colnames(data) == 'DX_bl' ) target <-
# data[,target_indx] rm_indx <- which( colnames(data) %in%
# c('DX_bl','ID','TOTAL13','MMSCORE') ) X <- data X <- X[,-rm_ind
x]

set.seed(1)

err.mat <- NULL
for (K in c(0.2, 0.3, 0.4, 0.5, 0.6, 0.7)) {

    testing.indices <- NULL
    for (i in 1:50) {
        testing.indices <- rbind(testing.indices, sample(nrow(dat
a), floor((1 -
            K) * nrow(data))))
    }

    for (i in 1:nrow(testing.indices)) {

        testing.ix <- testing.indices[i, ]
        target.testing <- data$DX_bl[testing.ix]

        tree <- rpart(DX_bl ~ ., data[-testing.ix, ])
        pred <- predict(tree, data[testing.ix, ], type = "class")
        error <- length(which(as.character(pred) != target.testin
g))/length(target.testing)
        err.mat <- rbind(err.mat, c("tree", K, error))

        rf <- randomForest(DX_bl ~ ., data[-testing.ix, ])
        pred <- predict(rf, data[testing.ix, ])
        error <- length(which(as.character(pred) != target.testin
g))/length(target.testing)
        err.mat <- rbind(err.mat, c("RF", K, error))

        data1 <- data
        data1$DX_bl <- as.numeric(as.character(data1$DX_bl))
        boost <- gbm(DX_bl ~ ., data = data1[-testing.ix, ], dist
```

```
 = "adaboost",
            interaction.depth = 6, n.tree = 2000)  #cv.folds = 5,

        # best.iter <- gbm.perf(boost,method='cv')
        pred <- predict(boost, data1[testing.ix, ], n.tree = 2000,
 type = "response")  # best.iter n.tree = 400,
        pred[pred > 0.5] <- 1
        pred[pred <= 0.5] <- 0
        error <- length(which(as.character(pred) != target.testin
g))/length(target.testing)
        err.mat <- rbind(err.mat, c("AdaBoost", K, error))
    }
}
err.mat <- as.data.frame(err.mat)
colnames(err.mat) <- c("method", "training_percent", "error")
err.mat <- err.mat %>% mutate(training_percent = as.numeric(as.ch
aracter(training_percent)),
    error = as.numeric(as.character(error)))

ggplot() + geom_boxplot(data = err.mat %>% mutate(training_percen
t = as.factor(training_percent)),
    aes(y = error, x = training_percent, color = method)) + geom_
point(size = 3)
```

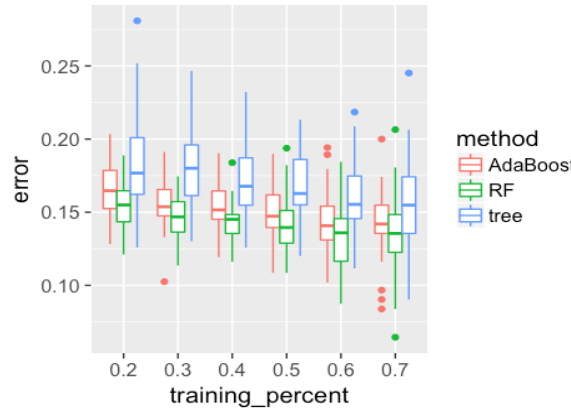


Figure 7.30: Boxplots of the classification error rates for single decision tree, random forests, and AdaBoost

It can be seen in Figure 7.30 that, all error rates are reduced as the percentage of the training data increases. The single decision tree is clearly less accurate than the other two ensemble methods. RF has lower error rates than AdaBoost in general. However, as the training data size increases, the gap between RF and AdaBoost seems to decrease slightly. This may indicate that when the training data size is small, RF is more stable due to its advantage of addressing the statistical issue.

Now we add model complexity to each model to see its impacts on the models' performance. First, we change the complexity parameter (`cp`) in decision tree. A smaller `cp` indicates a larger tree. It can be seen that the error rate decreases when the tree gets more complex, and only slightly increases after `cp` is greater than 0.02. This may indicate that, for this dataset, the main issue for decision tree may stem from the representational perspective, meaning that, a single decision is not able to capture enough information (as much as ensemble methods) using the training data. But the statistical issue is not severe, given that the error does not increase substantially given a complexity parameter value that could result in a large tree.

```
set.seed(1)
testing.indices <- NULL
for (i in 1:50) {
    testing.indices <- rbind(testing.indices, sample(nrow(data),
floor((0.3) *
        nrow(data))))
}


err.mat <- NULL
for (i in 1:nrow(testing.indices)) {
    testing.ix <- testing.indices[i, ]
    target.testing <- data$DX_bl[testing.ix]

    cp.v <- rev(c(0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.
08, 0.09, 0.1))
    for (j in cp.v) {
        tree <- rpart(DX_bl ~ ., data[-testing.ix, ], cp = j)
        pred <- predict(tree, data[testing.ix, ], type = "class")
        error <- length(which(as.character(pred) != target.testin
g))/length(target.testing)
```

```
        err.mat <- rbind(err.mat, c("Tree", j, error))
    }
}

err.mat <- as.data.frame(err.mat)
colnames(err.mat) <- c("method", "cp", "error")
err.mat <- err.mat %>% mutate(cp = as.numeric(as.character(cp)),
error = as.numeric(as.character(error)))
err.mat$cp <- factor(err.mat$cp, levels = sort(cp.v, decreasing =
 TRUE))


ggplot() + geom_boxplot(data = err.mat, aes(y = error, x = cp, co
lor = method)) +
    geom_point(size = 3)
```

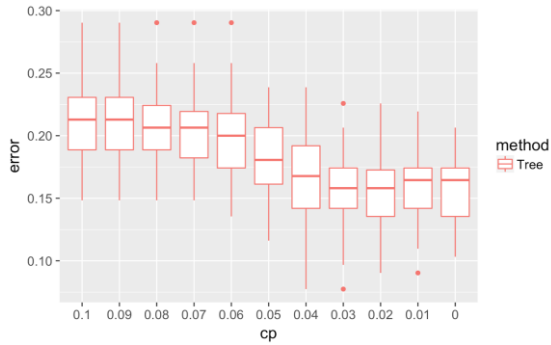


Figure 7.31: Boxplots of the classification error rates for single decision tree models with different model complexity (by controlling `cp`)

We also adjust the number of trees in AdaBoost and show the results in Figure 7.32. It can be seen that the error rates first go down as the number of trees increases to 400. However, the error rates increase after that, and decrease again. This may indicate that AdaBoost still have a statistical issue as the error rates are not stable.

```
err.mat <- NULL
set.seed(1)
for (i in 1:nrow(testing.indices)) {
```

```
    data1 <- data
    data1$DX_bl <- as.numeric(as.character(data1$DX_bl))
    ntree.v <- c(200, 300, 400, 500, 600, 800, 1000, 1200, 1400,
1600, 1800,
        2000)
    for (j in ntree.v) {
        boost <- gbm(DX_bl ~ ., data = data1[-testing.ix, ], dist
 = "adaboost",
            interaction.depth = 6, n.tree = j)
        # best.iter <- gbm.perf(boost,method='cv')
        pred <- predict(boost, data1[testing.ix, ], n.tree = j, t
ype = "response")
        pred[pred > 0.5] <- 1
        pred[pred <= 0.5] <- 0
        error <- length(which(as.character(pred) != target.testin
g))/length(target.testing)
        err.mat <- rbind(err.mat, c("AdaBoost", j, error))
    }
}
err.mat <- as.data.frame(err.mat)
colnames(err.mat) <- c("method", "num_trees", "error")
err.mat <- err.mat %>% mutate(num_trees = as.numeric(as.character
(num_trees)),
    error = as.numeric(as.character(error)))

ggplot() + geom_boxplot(data = err.mat %>% mutate(num_trees = as.
factor(num_trees)),
    aes(y = error, x = num_trees, color = method)) + geom_point(s
ize = 3)
```

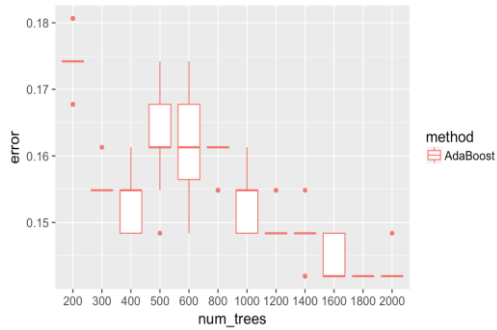


Figure 7.32: Boxplots of the classification error rates for AdaBoost with different number of trees

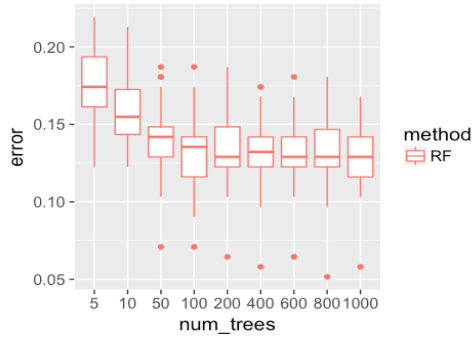


Figure 7.33: Boxplots of the classification error rates for random forest with different number of trees

Now let's look at random forests with different number of trees. Results are shown in Figure 7.33. Similar to AdaBoost, RF has high error rates initially at a small number of trees. Then, the error rates are reduced as more trees are added. However, the error rates become stable when more trees are added. This may indicate that RF handles the statistical issue well.

```
err.mat <- NULL
set.seed(1)
for (i in 1:nrow(testing.indices)) {
    testing.ix <- testing.indices[i, ]
    target.testing <- data$DX_bl[testing.ix]

    ntree.v <- c(5, 10, 50, 100, 200, 400, 600, 800, 1000)
    for (j in ntree.v) {
        rf <- randomForest(DX_bl ~ ., data[-testing.ix, ], ntree
= j)
        pred <- predict(rf, data[testing.ix, ])
        error <- length(which(as.character(pred) != target.testin
g))/length(target.testing)
        err.mat <- rbind(err.mat, c("RF", j, error))
    }
}
err.mat <- as.data.frame(err.mat)
colnames(err.mat) <- c("method", "num_trees", "error")
err.mat <- err.mat %>% mutate(num_trees = as.numeric(as.character
(num_trees)),
    error = as.numeric(as.character(error)))
```

```
ggplot() + geom_boxplot(data = err.mat %>% mutate(num_trees = as.
factor(num_trees)),
    aes(y = error, x = num_trees, color = method)) + geom_point(s
ize = 3)
```

Recall that, a requirement to solve the statistical issue in random forests is that a diverse set of learners need to be built. As we have mentioned, in random forests, there are two approaches to increase diversity, one is to boostrap samples for each tree while another one is to conduct random feature selection for splitting each node. First, we investigate the effectiveness of using randomly bootstrapped samples. We change sampling strategy from sampling with replacement to sampling without replacement, and change the sampling size from 10% to 100% (with the number of features tested at each node being the default value of $\sqrt[2]{p}$ where p is the number of features). The results as shown in Figure 7.34 do not show that the increased sample size has an impact on the error rates on this particular dataset.

```
err.mat <- NULL
set.seed(1)
for (i in 1:nrow(testing.indices)) {
    testing.ix <- testing.indices[i, ]
    target.testing <- data$DX_bl[testing.ix]

    sample.size.v <- seq(0.1, 1, by = 0.1)
    for (j in sample.size.v) {
        sample.size <- floor(nrow(data[-testing.ix, ]) * j)
        rf <- randomForest(DX_bl ~ ., data[-testing.ix, ], sampsi
ze = sample.size,
            replace = FALSE)
        pred <- predict(rf, data[testing.ix, ])
        error <- length(which(as.character(pred) != target.testin
g))/length(target.testing)
        err.mat <- rbind(err.mat, c("RF", j, error))
    }
}
err.mat <- as.data.frame(err.mat)
colnames(err.mat) <- c("method", "sample_size", "error")
err.mat <- err.mat %>% mutate(sample_size = as.numeric(as.charact
er(sample_size)),
    error = as.numeric(as.character(error)))
```

```
ggplot() + geom_boxplot(data = err.mat %>% mutate(sample_size = a
s.factor(sample_size)),
    aes(y = error, x = sample_size, color = method)) + geom_point
(size = 3)
```

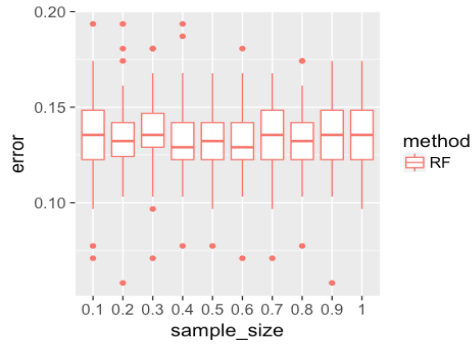


Figure 7.34: Boxplots of the classification error rates for random forest with different sample sizes

We then set the number of samples used at each tree the same as the size of the original training data set, and change the number of features in growing the random forest models. As shown in Figure 7.35, we can see that when the number of features is sufficiently large, the error rates start to increase, which could be due to a low diversity and indicate its compromised capability to address the statistical issue.

```
err.mat <- NULL
set.seed(1)
for (i in 1:nrow(testing.indices)) {
    testing.ix <- testing.indices[i, ]
    target.testing <- data$DX_bl[testing.ix]

    num.fea.v <- 1:(ncol(data) - 1)
    for (j in num.fea.v) {
        sample.size <- nrow(data[-testing.ix, ])
        rf <- randomForest(DX_bl ~ ., data[-testing.ix, ], mtry =
 j, sampsize = sample.size,
            replace = FALSE)
        pred <- predict(rf, data[testing.ix, ])
        error <- length(which(as.character(pred) != target.testin
```

```
g))/length(target.testing)
        err.mat <- rbind(err.mat, c("RF", j, error))
    }
}
err.mat <- as.data.frame(err.mat)
colnames(err.mat) <- c("method", "num_fea", "error")
err.mat <- err.mat %>% mutate(num_fea = as.numeric(as.character(n
um_fea)), error = as.numeric(as.character(error)))

ggplot() + geom_boxplot(data = err.mat %>% mutate(num_fea = as.fa
ctor(num_fea)),
    aes(y = error, x = num_fea, color = method)) + geom_point(siz
e = 3)
```

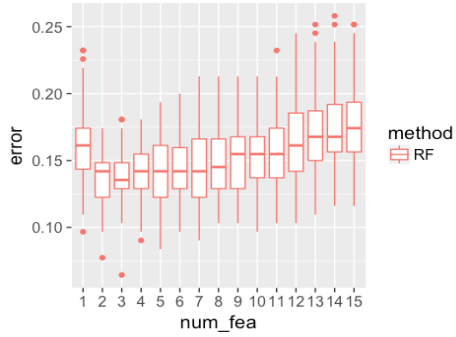


Figure 7.35: Boxplots of the classification error rates for random forest with different number of features

In the next experiment, we fix the number of tested features to be the total number of variables, and change the percentage of samples to be used at each tree. As shown in Figure 7.36, the error rate is relatively large when the percentage of samples is small (10%). But as more samples are used, the error rates increase, and reach the highest error rate when 100% of the samples are used (i.e., which leads to least diversity among the base learners).

```
err.mat <- NULL
set.seed(1)
for (i in 1:nrow(testing.indices)) {
```

```
    testing.ix <- testing.indices[i, ]
    target.testing <- data$DX_bl[testing.ix]

    sample.size.v <- seq(0.1, 1, by = 0.1)
    for (j in sample.size.v) {
        traing.data <- data[-testing.ix, ]
        sample.size <- floor(nrow(traing.data) * j)
        rf <- randomForest(DX_bl ~ ., traing.data, mtry = ncol(tr
aing.data) -
            1, sampsize = sample.size, replace = FALSE)
        pred <- predict(rf, data[testing.ix, ])
        error <- length(which(as.character(pred) != target.testin
g))/length(target.testing)
        err.mat <- rbind(err.mat, c("RF", j, error))
    }
}
err.mat <- as.data.frame(err.mat)
colnames(err.mat) <- c("method", "num_fea", "error")
err.mat <- err.mat %>% mutate(num_fea = as.numeric(as.character(n
um_fea)), error = as.numeric(as.character(error)))

ggplot() + geom_boxplot(data = err.mat %>% mutate(num_fea = as.fa
ctor(num_fea)),
    aes(y = error, x = num_fea, color = method)) + geom_point(siz
e = 3)
```

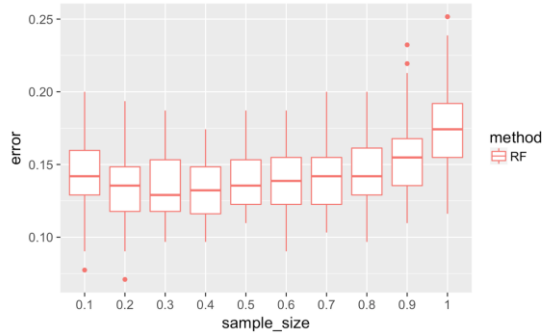


Figure 7.36: Boxplots of the classification error rates for random forest with different sample sizes

IV. Exercises

Data analysis

1. Find ten classification datasets from the UCI data repository or R datasets. Conduct a detailed analysis using the logistic regression model, SVM, decision tree, random forest, and AdaBoost. Conduct model selection and validation. Use cross-validation to select the best models. Conduct residual analysis of your final models, and comment on your results.

Programming

2. Write your own R script to implement the linear SVM model.
3. Extend your SVM code to nonlinear SVM models with Gaussian kernel and polynomial kernel. Compare your results with `ksvm()`.
4. Write your own R script to implement the AdaBoost model. Compare your results with `gbm()`.