9 Supervised Learning

"Artificial Intelligence, deep learning, machine learning – whatever you're doing if you do not understand it – learn it. Because otherwise you're going to be a dinosaur within 3 years."

— Mark Cuban

What do you need?
- A good understanding of statistical concepts, probability theory (see Appendix B), and functions.
- Basics of differential calculus (see Appendix A for a few handy formulas).
- Introductory to intermediate-level experience with R, including installing packages or libraries (refer to Chapter 6).
- Everything covered in Chapter 8.

What will you learn?
- Solving data problems when truth values for training are available.
- Performing classification using various machine learning techniques.

9.1 Introduction

In the previous chapter we were introduced to the concept of learning – both for humans and for machines. In either case, a primary way one learns is first knowing what is a correct outcome or label of a given data point or a behavior. As it happens, there are many situations when we have training examples with correct labels. In other words, we have data for which we know the correct outcome value. This set of data problems collectively fall under supervised learning.

Supervised learning algorithms use a set of examples from previous records to make predictions about the future. For instance, existing car prices can be used to make guesses about the future models. Each example used to train such an algorithm is labeled with the value of interest – in this case, the car's price. A supervised learning algorithm looks for patterns in a training set. It may use any information that might be relevant – the season, the car's current sales records, similar offerings from competitors, the manufacturer's brand perception owned by the consumers – and each algorithm may look for a different set of information and find different types of

patterns. Once the algorithm has found the best pattern it can, it uses that pattern to make predictions for unlabeled testing data – tomorrow's values.

There are several types of supervised learning that exist within machine learning. Among them, the three most commonly used algorithm types are regression, classification, and anomaly detection. In this chapter, we will focus on regression and classification. Yes, we covered linear regression in the previous chapter, but that was for predicting a continuous variable such as age and income. When it comes to predicting discrete values, we need to use another form of regression – logistic regression or softmax regression. These are essentially forms of classification. And then we will see several of the most popular and useful techniques for classification. You will also find a quick introduction to anomaly detection in an FYI box later in the chapter.

9.2 Logistic Regression

One thing you should have noticed by now about linear regression is that the outcome variable is numerical. So, the question is: What happens when the outcome variable is not numerical? For example, if you have a weather dataset with the attributes humidity, temperature, and wind speed, each is describing one aspect of the weather for a day. And based on these attributes, you want to predict if the weather for the day is suitable for playing golf. In this case, the outcome variable that you want to predict is categorical ("yes" or "no"). Fortunately, to deal with this kind of classification problem, we have logistic regression.

Let us think of this in a formal way. Before, our outcome variable y was continuous. Now, it can have only two possible values (labels). For simplicity, let us call these labels "1" and "0" ("yes" and "no"). In other words,

$$y \in \{0, 1\}.$$

We are still going to have continuous value(s) for the input, but now we need to have only two possible values for the output. How do we do this? There is an amazing function called sigmoid, which is defined as

$$g(z) = \frac{1}{1 + e^{-z}} \tag{9.1}$$

and it looks like Figure 9.1.

As you can see in Figure 9.1, for any input, the output of this function is bound between 0 and 1. In other words, if used as the hypothesis function, we get the output in 0 to 1 range, with 0 and 1 included:

$$h_\theta(x) \in [0, 1]. \tag{9.2}$$

The nice thing about this is that it follows the constraints of a probability distribution that it should be contained between 0 and 1. And if we could compute a probability that ranges

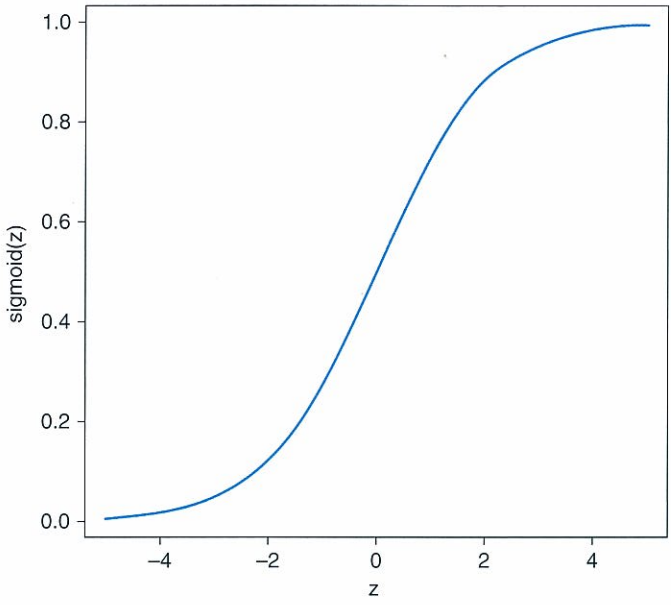

Figure 9.1 Sigmoid function.

from 0 to 1, it would be easy to draw a threshold at 0.5 and say that any time we get an outcome value from a hypothesis function h greater than that, we put it in class "1," otherwise it goes in class "0." Formally:

$$\begin{aligned} P(y = 1|x;\theta) &= h_\theta(x), \\ P(y = 0|x;\theta) &= 1 - h_\theta(x), \\ P(y|x;\theta) &= (h_\theta(x))^y(1 - h_\theta(x))^{1-y}. \end{aligned} \tag{9.3}$$

The last formulation is the result of combining the first two lines to form one expression. See if that makes sense. Try putting $y = 1$ and $y = 0$ in that expression and see if you get the previous two lines.

Now, how do we use this for classification? In essence, we want to input whatever features from the data we have into the hypothesis function (here, a sigmoid) and find out the value that comes out between 0 and 1. Based on which side of 0.5 it is, we can declare an appropriate label or class. But before we can do that (called testing), we need to train a model. For this we need some data. One way we could build a model from such data is to assume a model and ask if that model could explain or classify the training data and how well. In other words, we are asking how *good* our model is, given the data.

To understand the *goodness* of a model (as represented by the parameter vector θ), we can ask how likely it is that the data we have is generated by the given model. This is called the **likelihood** of the model and is represented as $L(\theta)$. Let us expand this likelihood function:

$$\begin{aligned} L(\theta) &= P(y = 1|X;\theta) \\ &= \prod_{i=1}^{m} P(y^i|x^i;\theta) \\ &= \prod_{i=1}^{m} (h_\theta(x^i))^{y^i}(1 - h_\theta(x^i))^{1-y^i}. \end{aligned} \tag{9.4}$$

To achieve a better model than the one we guessed, we need to increase the value of $L(\theta)$. But, look at that function above. It has all those multiplications and exponents. So, to make it easier for us to work with this function, we will take its log. This is using the property of log that it is an increasing function (as x goes up, $\log(x)$ also goes up). This will give us a log likelihood function as below:

$$\begin{aligned} l(\theta) &= \log L(\theta) \\ &= \sum_{i=1}^{m} [y^i \log(h_\theta(x^i)) + (1 - y^i)\log(1 - h_\theta(x^i))]. \end{aligned} \tag{9.5}$$

Once again, to achieve the best model, we need to maximize this log likelihood. For that, we do what we already know – take the partial derivative, one parameter at a time. In fact, for simplicity, we will even consider just one sample data at a time:

$$\begin{aligned} \frac{\partial}{\partial\theta_j} l(\theta) &= \left(y\frac{1}{h_\theta(x)} - (1 - y)\frac{1}{1 - h_\theta(X)} \right) \frac{\partial}{\partial\theta_j} h_\theta(x) \\ &= [y(1 - h_\theta(x)) - (1 - y)h_\theta(x)]x_j \\ &= (y - h_\theta(x))x_j. \end{aligned} \tag{9.6}$$

The second line above follows the fact that, for a sigmoid function $g(z)$, the derivative can be expressed as: $g'(z) = g(z)(1 - g(z))$.

Considering all training samples, we get:

$$\frac{\partial}{\partial\theta_j} l(\theta) = \sum_{i=1}^{m} (y^i - h_\theta(x^i))x_j^i. \tag{9.7}$$

This gives us our learning algorithm:

$$\theta_j = \theta_j + \alpha \sum_{i=1}^{m} (y^i - h_\theta(x^i))x_j^i. \tag{9.8}$$

Notice how we are updating the θ this time. We are moving up on the gradient instead of moving down. And that is why this is called gradient ascent. It does look similar to gradient descent, but the difference is the nature of the hypothesis function. Before, it was a linear function. Now it is sigmoid or logit function. And because of that, this regression is called **logistic regression**.

Hands-On Example 9.1: Logistic Regression

Let us practice logistic regression with an example. We are going to use the *Titanic dataset*, different versions of which are freely available online; however, I suggest using the one from OA 9.1, since it is almost ready to be used and requires minimum pre-processing. In this exercise, we are trying to predict the survival chances of the passengers on the *Titanic*.

After obtaining the datasets, first import the training dataset into a dataframe in R:

```
> titanic.data <- read.csv("train.csv", header = TRUE,
sep = ",")
> View(titanic.data)
```

Figure 9.2 shows a snapshot from RStudio with a sample of the data.

Before we go ahead and build the model, we need to check for missing values and find how many unique values there are for each variable using the sapply() function, which applies the function passed as argument to each column of the dataframe. Here is how to do it:

```
> sapply(titanic.data,function(x) sum(is.na(x)))
PassengerId Survived Pclass Name        Sex   Age  SibSp
          0        0      0     0          0 177      0
      Parch    Ticket   Fare Cabin  Embarked
          0        0      0     0          0

> sapply(titanic.data,function(x) length(unique(x)))
PassengerId Survived Pclass  Name        Sex   Age  SibSp
        891        2      3   891          2    89      7
      Parch    Ticket   Fare Cabin  Embarked
          7      681    248   148          4
```

Another way to estimate the missing values is to use a visualization package: the "Amelia" package has a plotting function missmap() that serves the purpose. It will plot your dataset and highlight missing values:

```
> library(Amelia)
Loading required package: Rcpp
##
## Amelia II: Multiple Imputation
## (Version 1.7.4, built: 2015-12-05)
## Copyright (C) 2005-2017 James Honaker, Gary King and
Matthew Blackwell
## Refer to http://gking.harvard.edu/amelia/for more
information
##
  >missmap(titanic.data, main = "Missing values vs observed")
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	1	0	3	Braund, Mr. Owen Harris	male	22.00	1	0	A/5 21171	7.2500		S
2	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38.00	1	0	PC 17599	71.2833	C85	C
3	3	1	3	Heikkinen, Miss. Laina	female	26.00	0	0	STON/o2. 3101282	7.9250		S
4	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.00	1	0	113803	53.1000	C123	S
5	5	0	3	Allen, Mr. William Henry	male	35.00	0	0	373450	8.0500		S
6	6	0	3	Moran, Mr. James	male	*NA*	0	0	330877	8.4583		Q
7	7	0	1	McCarthy, Mr. Timothy J	male	54.00	0	0	17463	51.8625	E46	S
8	8	0	3	Palsson, Master. Gosta Leonard	male	2.00	3	1	349909	21.0750		S
9	9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27.00	0	2	347742	11.1333		S
10	10	1	2	Nasser, Mrs. Nicholas (Adele Achem)	female	14.00	1	0	237736	30.0708		C
11	11	1	3	Sandstrom, Miss. Marguerite Rut	female	4.00	1	1	PP 9549	16.7000	G6	S
12	12	1	1	Bonnell, Miss. Elizabeth	female	58.00	0	0	113783	26.5500	C103	S
13	13	0	3	Saundercock, Mr. William Henry	male	20.00	0	0	A/5. 2151	8.0500		S
14	14	0	3	Andersson, Mr. Anders Johan	male	39.00	1	5	347082	31.2750		S
15	15	0	3	Vestrom, Miss. Hulda Amanda Adolfina	female	14.00	0	0	350406	7.8542		S

Figure 9.2 Sample of the *Titanic* data.

Figure 9.3 Visualizing missing values.

As Figure 9.3 suggests, the Age column has multiple missing values. So, we must clean up the missing values before proceeding further. In Chapter 2, we saw multiple methods for doing such data cleanup. In this case, we will go with replacing those missing values with the mean age value. This is how to do that:

```
> titanic.data$Age[is.na(titanic.data$Age)] <- mean
(titanic.data$Age,na.rm=T)
```

Here we have replaced the missing values with the mean age of the rest of the population. If any column has a significant number of missing values, you may want to consider removing the column altogether. For the purpose of this exercise, we will use only the Age, Embarked, Fare, Ticket, Parch, SibSp, Sex, Pclass, and Survived columns to simplify our model:

```
> titanic.data <- subset(titanic.data,select=c
(2,3,5,6,7,8,10,12))
```

For the categorical variables in the dataset, using the read.table() or read.csv() by default will encode the categorical variables as factors. A factor is how R deals with categorical variables.

We can check the encoding using the is.factor() function, which should return "true" for all the categorical variables:

```
> is.factor(titanic.data$Sex)
[1] TRUE
```

Now, before building the model you need to separate the dataset into training and test sets. I have used the first 800 instances for training and the remaining 91 as test instances. You can opt for different separation strategies:

```
> train <- titanic.data[1:800,]
> test <- titanic.data[801:891,]
```

Now our dataset is ready for building the model. We are going to use parameter `family=binomial` (two classes or labels) in the glm() function:

```
> model <- glm(Survived ~.,family=binomial(link='logit'),
data=train)
> summary(model)
```

The above lines of code should produce the following lines of output:

```
Call:
glm(formula = Survived ~ ., family = binomial(link = "logit"),
    data = train)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.6059  -0.5933  -0.4250   0.6215   2.4148
Coefficients:
             Estimate  Std. Error  z value     Pr(>|z|)
(Intercept) 15.947761  535.411378    0.030   0.9762
Pclass      -1.087576    0.151088   -7.198   6.10e-13***
Sexmale     -2.754348    0.212018  -12.991     <2e-16***
Age         -0.037244    0.008192   -4.547   5.45e-06***
SibSp       -0.293478    0.114660   -2.560   0.0105*
Parch       -0.116828    0.128113   -0.912   0.3618
Fare         0.001515    0.002352    0.644   0.5196
EmbarkedC  -10.810682  535.411254   -0.020   0.9839
EmbarkedQ  -10.812679  535.411320   -0.020   0.9839
EmbarkedS  -11.126837  535.411235   -0.021   0.9834
---
Signif. codes: ***, 0; **,0.001; *, 0.01.

Dispersion parameter for binomial family taken to be 1
  Null deviance: 1066.33 on 799 degrees of freedom.
Residual deviance: 708.93 on 790 degrees of freedom.
```

```
AIC: 728.93.

Number of Fisher scoring iterations: 12.
```

From the result, it is clear that Fare and Embarked are not statistically significant. That means we do not have enough confidence that these factors contribute all that much to the overall model. As for the statistically significant variables, Sex has the lowest *p*-value, suggesting a strong association of the sex of the passenger with the probability of having survived. The negative coefficient for this predictor suggests that, all other variables being equal, the male passenger is less likely to have survived. At this time, we should pause and think about this insight. As *Titanic* started sinking and the lifeboats were being filled with rescued passengers, priority was given to women and children. And thus, it makes sense that a male passenger, especially a male adult, would have had less chance of survival.

Now, we will see how good our model is in predicting values for test instances. By setting the parameter `type='response'`, R will output probabilities in the form of $P(y = 1 \mid X)$. Our decision boundary will be 0.5. If $P(y = 1 \mid X) > 0.5$ then $y = 1$, otherwise $y = 0$.

```
> fitted.results <-
predict(model,newdata=subset(test,select=c
(2,3,4,5,6,7,8)), type='response')
> fitted.results <- ifelse(fitted.results > 0.5,1,0)
>
> misClasificError <- mean(fitted.results != test$Survived)
> print(paste('Accuracy',1-misClasificError))
[1] "Accuracy 0.842696629213483"
```

As we can see from the above result, the accuracy of our model in predicting the labels of the test instances is at 0.84, which suggests that the model performed decently.

At the final step, we are going to plot the receiver operating curve (ROC) and calculate the area under curve (AUC), which are typical performance measurements for a binary classifier (see Figure 9.4). For the details of these measures, refer to Chapter 12.

```
> library(ROCR)
Loading required package: gplots
Attaching package: 'gplots'
The following object is masked from 'package:stats': lowess

> p <- predict(model, newdata=subset(test,select=c
(2,3,4,5,6,7,8)), type="response")
> pr <- prediction(p, test$Survived)
> prf <- performance(pr, measure = "tpr", x.measure = "fpr")
> plot(prf)

> auc <- performance(pr, measure = "auc")
```

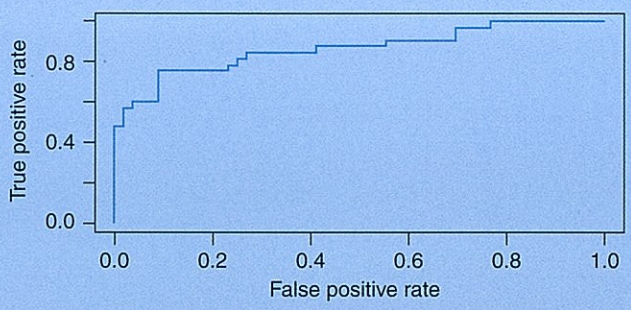

Figure 9.4 Receiver operating curve (ROC) for the classifier built on *Titanic* data.

```
> auc <- auc@y.values[[1]] > auc
[1] 0.866342
```

The ROC curve is generated by plotting the **true positive rate** (TPR) against the **false positive rate** (FPR) at various threshold settings, while the AUC is the area under the ROC curve. TPR indicates how much of what we detected as "1" was indeed "1," and FPR indicates how much of what we detected as "1" was actually "0." Typically, as one goes up, the other goes up too. Think about it – if you declare everything "1," you will have a high TPR, but you would have also wrongly labeled everything that was supposed to be "0," leading to high FPR.

As a rule of thumb, a model with good predictive ability should have an AUC closer to 1 than to 0.5. In Figure 9.4, we can see that the area under the curve is quite large – covering about 87% of the rectangle. That is quite a good number, indicating a good and balanced classifier.

Try It Yourself 9.1: Logistic Regression

First, obtain the social media ads data from OA 9.2. Using this data, build a logistic regression-based classifier to determine if the social media user's demographics can be used to predict the user buying the product being advertised. Report your model's classification accuracy as well as ROC value.

9.3 Softmax Regression

So far, we have seen regression for numerical outcome variable as well as regression for binomial ("yes" or "no", "1" or "0") categorical outcome. But what happens if we have more than two categories. For example, you want to rate a student's performance based on the numbers he got in individual subjects as "excellent," "good," "average," or "below average." We need to have multinomial logistic regression for this. In this sense

multinomial logistic regression or softmax regression is a generalization of regular logistic regression to handle multiple (more than two) classes.

In softmax regression, we replace the sigmoid function from the logistic regression by the so-called softmax function. This function takes a vector of n real numbers as input and normalizes the vector into a distribution of n probabilities. That is, the function transforms all the n components from any real values (positive or negative) to values in the interval (0, 1). How it does that is a discussion beyond this book; however, if you are interested, you can check the further reading and resources at the end of this chapter.

Hands-On Example 9.2: Softmax Regression

We will see softmax regression through an example in R. In this example, we will use the "hsbdemo" dataset available from OA 9.3. The dataset is about entering high-school students who make program choices among general programs, vocational programs, and academic programs. Their choices might be modeled using their writing scores and their social economic status. The dataset contains attribute values on 200 students. The outcome variable is "prog," which is the program type chosen by the student. The predictor variables are social economic status, "ses," a three-level categorical variable, and writing score, "write," a continuous variable.

Let us start with getting some descriptive statistics of the variables of interest.

```
> hsbdemo <- read.csv("hsbdemo.csv", header = TRUE, sep = ",")
> View(hsbdemo)
> with(hsbdemo, table(ses, prog))
        prog
ses    academic general vocation
 high       42       9        7
 low        19      16       12
 middle     44      20       31
> with(hsbdemo, do.call(rbind, tapply(write, prog, function
(x) c(M = mean(x), SD = sd(x)))))
                 M       SD
academic  56.257147.943343
general   51.333339.397775
vocation  46.760009.318754
```

There are multiple ways and packages available in R capable of performing multinomial logistic regression. For example, you can use the mlogit package to do multinomial logistic regression. However, for this example, if we want to use the mlogit package, we must reshape the dataset first. A possible work-around of this pre-processing step is to use the multinom function from the nnet package to estimate a multinomial logistic regression model, which does not require any reshaping of the data.

Before running the multinomial regression, we need to remember our outcome variable is not ordinal (e.g., "good," "better," and "best"). So, to create our model, we need to choose the level of the outcome that we wish to use as the baseline and specify this in the relevel function. Here is how to do that:

```
> hsbdemo$prog2 <- relevel(hsbdemo$prog, ref = "academic")
```

Here, instead of transforming the original variable "prog," we have declared another variable "prog2" using the relevel function, where the level "academic" is declared as baseline:

```
> library(nnet)
> model1 <- multinom(prog2 ~ ses + write, data = hsbdemo)
# weights: 15 (8 variable)
initial value 219.722458
iter 10 value 179.983731
final value 179.981726
Converged
```

Ignore all the warning messages. As we can see, we have built a model where the outcome variable is "prog2." For demonstration purposes, we used only "ses" and "write" as our predictors and ignored the remaining variables. As we can see, the model has generated some output itself even though we are assigning the model to a new R object. This model-running output includes some iteration history and includes the final negative log-likelihood value, 179.981726.

Next, to explore more details about the model we have built so far, we can issue a summary command on our model:

```
> summary(model1)
Call:
multinom(formula = prog2 ~ ses + write, data = hsbdemo)
Coefficients:
          (Intercept)    seslow  sesmiddle       write
general     1.689478 1.1628411 0.6295638 -0.05793086
vocation    4.235574 0.9827182 1.2740985 -0.11360389
Std. Errors:
          (Intercept)    seslow  sesmiddle       write
general     1.226939 0.5142211 0.4650289 0.02141101
vocation    1.204690 0.5955688 0.5111119 0.02222000

Residual Deviance: 359.9635
AIC: 375.9635
```

The output summary generated by the model has a block of coefficients and a block of standard errors. Each of these blocks has one row of values corresponding to a model equation. We are going to focus on the block of coefficients first. As we can see, the first row is comparing `prog=general` to our baseline

`prog=academic`. Similarly, the second row represents a comparison between `prog=vocation` and the baseline.

Let us declare the coefficients from the first row to be "b_1" (b_{10} for the intercept, b_{11} for "seslow," b_{12} for "sesmiddle," and b_{13} for "write") and our coefficients from the second row to be "b_2" (b_{20} for the intercept, b_{21} for "seslow," b_{22} for "sesmiddle," and b_{23} for "write"). Using these, we can compute something called log odds, which compares a given model to the baseline and informs us how a one-unit change in an independent variable would change a dependent variable in the model compared to how it would change the same variable for the baseline. These model equations as can be written as follows:

$$\ln\left(\frac{P(\text{prog2} = \text{general})}{P(\text{prog2} = \text{academic})}\right) = b_{10} + b_{11}(\text{ses} = 2) + b_{12}(\text{ses} = 3) + b_{13}(\text{write}) \tag{9.9}$$

and

$$\ln\left(\frac{P(\text{prog2} = \text{vocation})}{P(\text{prog2} = \text{academic})}\right) = b_{20} + b_{21}(\text{ses} = 2) + b_{22}(\text{ses} = 3) + b_{23}(\text{write}). \tag{9.10}$$

Using these equations, we can find out that a one-unit increase in the variable "write" is associated with the decrease in the log odds of being in "general" program vs. "academic" program in the amount of 0.058. You can derive similar insights using the results from the summary of our model.

You will often come across a term – **relative risk** – in this kind of analysis, which is the ratio of the probability of choosing any outcome category ("vocation", "general") other than baseline over that of the baseline category. To find relative risk, we can exponentiate the coefficients from our model:

```
> exp(coef(model1))
        (Intercept)    seslow sesmiddle     write
general    5.416653 3.199009  1.876792 0.9437152
vocation 69.101326 2.671709  3.575477 0.8926115
```

As we can see, the relative risk ratio for a one-unit increase in the variable write is 0.9437 for being in "general" program vs. "academic" program.

You have seen before the need to have some test instances to examine the accuracy of your model. We saw how we could divide the entire dataset into training and test instances in cases where we do not have any pre-supplied test instances. I am not going to repeat the same here; instead, I will leave it to you to process the data and analyze the accuracy of the model.

Try It Yourself 9.2: Softmax Regression

Download the Car evaluation dataset from OA 9.4. Then, build a softmax regression model to classify the car acceptability class from other attributes of the class.

9.4 Classification with kNN

Sections 9.1 and 9.2 covered two forms of regression that accomplished one task: classification. We will continue with this now and look at other techniques for performing classification. The task of classification is: given a set of data points and their corresponding labels, to learn how they are classified so, when a new data point comes, we can put it in the correct class.

Classification can be supervised or unsupervised. The former is the case when assigning a label to a picture as, for example, either "cat" or "dog." Here the number of possible choices is predetermined. When there are only two choices, it is called two-class or binomial classification. When there are more categories, such as when predicting the winner of the NCAA March Madness tournament, it is known as multiclass or multinomial classification. There are many methods and algorithms for building classifiers, with k nearest neighbor (kNN) being one of the most popular ones.

Let us look at how kNN works by listing the major steps of the algorithm.

1. As in the general problem of classification, we have a set of data points for which we know the correct class labels.
2. When we get a new data point, we compare it to each of our existing data points and find similarity.
3. Take the most similar k data points (k nearest neighbors).
4. From these k data points, take the majority vote of their labels. The winning label is the label/class of the new data point.

The number k is usually small between 2 and 20. As you can imagine, the more the number of nearest neighbors (value of k), the longer it takes us to do the processing.

Hands-On Example 9.3: kNN

Let us take an example and try to visualize how to do the classification with kNN in R. For this example, we will use the "Iris" dataset. A brief description as well as the dataset itself are available from OA 9.5.

The dataset includes a sample of 50 flowers from three species of iris (*Iris setosa*, *Iris virginica* and *Iris versicolor*). Each sample was measured for four features: the length and the width of the sepals and petals, in centimeters. Based on the combination of these four attributes, we need to distinguish the species of iris.

The Iris dataset is built into R, so we can take a look at this dataset by typing "iris" into your R (or RStudio) console.

Before we proceed into classification, let us look into the distribution of values in the dataset. For this visualization, we will use the package *ggvis* in R. Here is how:

```
# Load the library "ggvis"
library(ggvis)
```

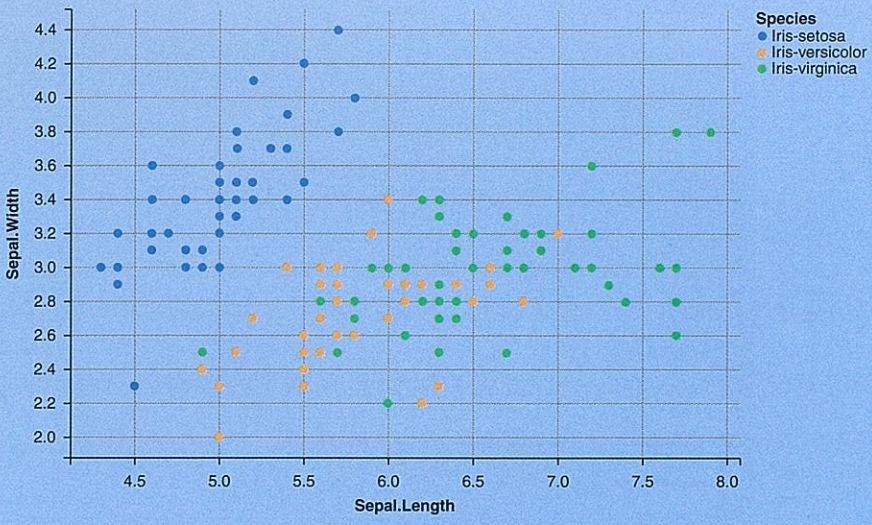

Figure 9.5 Plotting of IRIS data based on various flowers' sepal lengths and widths.

```
# Iris scatterplot
iris %>% ggvis(~Sepal.Length, ~Sepal.Width, fill = ~Species)
```

From Figure 9.5, we can see that there is a high correlation between the sepal length and the sepal width of the *Iris setosa* flowers, whereas the correlation is somewhat less for the *I. virginica* and *I. versicolor* flowers.

If we map the relation between petal length and the petal width, it tells a similar story, as shown in Figure 9.6.

```
iris %>% ggvis(~Petal.Length, ~Petal.Width, fill = ~Species)
```

The graph indicates a positive correlation between the *petal length* and the *petal width* for all different species that are included in the Iris dataset.

Once we have at least some idea of the nature of the dataset, we will see how to do kNN classification in R.

But before we proceed into the classification task, we need to have a test set to assess the model's performance later. Since we do not have that yet, we will need to divide the dataset into two parts: a training set and a test set. We will split the entire dataset into two-thirds and one-third. The first part, the larger chunk of the dataset, will be reserved for training, whereas the rest of the dataset is going to be used for testing. We can split them any way we want, but we need to remember that the training set must be sufficiently large to produce a good model. Also, we must make sure that all three classes of species are present in the training model. Even more important, the amounts of instances of all three species needs to be more or less equal, so that you do not favor one or the other class in your predictions.

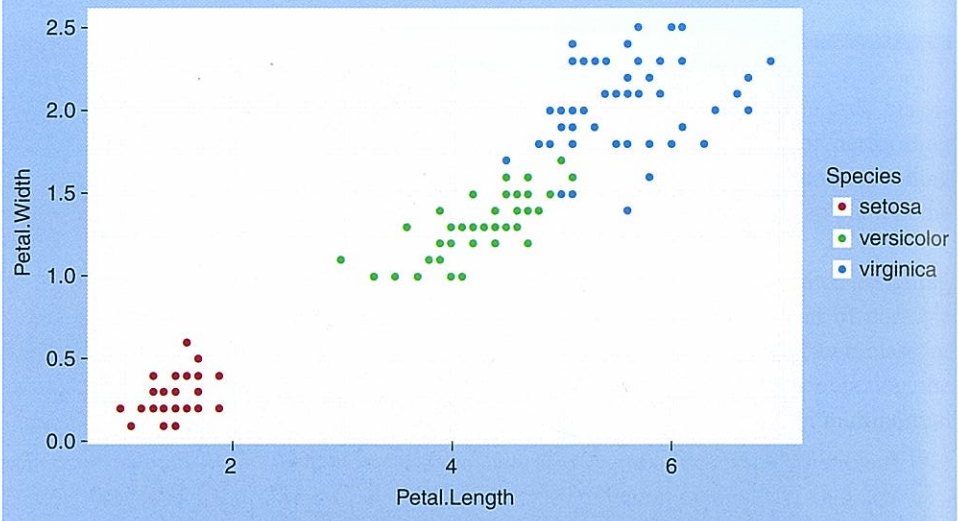

Figure 9.6 Plotting of Iris data based on various flowers' petal lengths and widths.

To divide the dataset into training and test sets, we should first set a seed. This is a number of R's random number generator. The major advantage of setting a seed is that we can get the same sequence of random numbers whenever you supply the same seed in the random number generator. Here is how to do that:

```
set.seed(1234)
```

You can pick any number other than 1234 in the above line. Next, we want to make sure that our Iris dataset is shuffled and that we have an equal amount of each species in our training and test sets. One way to ensure that is to use the *sample()* function to take a sample with a size that is set as the number of rows of the Iris dataset (here, 150). We sample with replacement: we choose from a vector of two elements and assign either "1" or "2" to the 150 rows of the Iris dataset. The assignment of the elements is subject to probability weights of 0.67 and 0.33. This results in getting about two-thirds of the data labeled as "1" (training) and the rest as "2" (testing).

```
ind <- sample(2, nrow(iris), replace=TRUE, prob=c
(0.67, 0.33))
```

We can then use the sample that is stored in the variable "ind" to define our training and test sets, only taking the first four columns or attributes from the data.

```
iris.training <- iris[ind==1, 1:4]
iris.test <- iris[ind==2, 1:4]
```

Also, we need to remember that "Species," which is the class label, is our target variable, and the remaining attributes are predictor attributes. Therefore, we need to store the class label in factor vectors and divide them over the training and test sets, which can be done by following steps:

```
iris.trainLabels <- iris[ind==1,5]
iris.testLabels <- iris[ind==2,5]
```

It is alright if you do not understand the above steps, as those have nothing to do with kNN, but are to do with how to prepare the dataset into training and test sets. What is more important is coming next.

Once all these preparatory steps are complete, we are ready to use the kNN in our training dataset. To do that, we need to use the knn() function as available in R. The knn() function uses the Euclidean distance to find the similarities between the *k*-training instances and your test instance. The value of *k* has to be supplied by the user, which is you in this case.

We can build the kNN-based model by executing the following steps:

```
library(class)
iris_pred <- knn(train = iris.training, test = iris.test, cl =
iris.trainLabels, k=3)
```

At this point, entering "iris_pred" prints the entire vector of our predictions. Or, we can ask for a summary by issuing "summary(iris_pred)" and get the following output:

```
setosa  versicolor  virginica
    12          13         15
```

Since we have built a model and predicted the class labels for our test attributes, let us evaluate how accurate those predictions are. For this, we will use cross-tabulation. A function for this can be found in the "gmodels" library. If you do not already have it installed, go ahead and install that package, and then run the following:

```
library(gmodels)
CrossTable(x=iris_pred, y=iris.testLabels, prop.chisq =
FALSE)
```

This gives us the cross-tabulation table as shown below. From this table, we can see how our predictions (iris_pred) matched up with the truth (iris.testLabels). There seems to be only one case when we were wrong – predicting "versicolor" for something that was "virginica." That is not bad at all.

```
Cell Contents
|-------------------|
|                   N|
|         N/Row Total|
|         N/Col Total|
|       N/Table Total|
|-------------------|

Total Observations in Table: 40
```

	iris.testLabels			
iris_pred	setosa	versicolor	virginica	Row Total
setosa	12	0	0	12
	1.000	0.000	0.000	0.300
	1.000	0.000	0.000	
	0.300	0.000	0.000	
versicolor	0	12	1	13
	0.000	0.923	0.077	0.325
	0.000	1.000	0.062	
	0.000	0.300	0.025	
virginica	0	0	15	15
	0.000	0.000	1.000	0.375
	0.000	0.000	0.938	
	0.000	0.000	0.375	
Column Total	12	12	16	40
	0.300	0.300	0.400	

Using this table, you can easily calculate our accuracy. Out of 40 predictions, we were wrong one time. So that gives us 39/40 = 97.5% accuracy.

Try It Yourself 9.3: kNN

Obtain the "hsbdemo" dataset from OA 9.3. Create a kNN-based classifier from the reading, writing, mathematics, and science scores of the high-school students. Evaluate the classifier's accuracy in predicting which academic program the student will be joining.

9.5 Decision Tree

In machine learning, a decision tree is used for classification problems. In such problems, the goal is to create a model that predicts the value of a target variable based on several input variables. A decision tree builds classification or regression models in the form of a tree structure. It breaks down a dataset into smaller and smaller subsets while at the same time an associated

Table 9.1 Balloons dataset.

Color	Size	Act	Age	Inflated
Yellow	Small	Stretch	Adult	T
Yellow	Small	Stretch	Adult	T
Yellow	Small	Stretch	Child	F
Yellow	Small	Dip	Adult	F
Yellow	Small	Dip	Child	F
Yellow	Large	Stretch	Adult	T
Yellow	Large	Stretch	Adult	T
Yellow	Large	Stretch	Child	F
Yellow	Large	Dip	Adult	F
Yellow	Large	Dip	Child	F
Purple	Small	Stretch	Adult	T
Purple	Small	Stretch	Adult	T
Purple	Small	Stretch	Child	F
Purple	Small	Dip	Adult	F
Purple	Small	Dip	Child	F
Purple	Large	Stretch	Adult	T
Purple	Large	Stretch	Adult	T
Purple	Large	Stretch	Child	F
Purple	Large	Dip	Adult	F
Purple	Large	Dip	Child	F

decision tree is incrementally developed. The final result is a tree with decision nodes and leaf nodes.

Consider the balloons dataset (download from OA 9.6) presented in Table 9.1. The dataset has four attributes: color, size, act, and age, and one class label, inflated (T = true or F = false). We will use this dataset to understand how a decision tree algorithm works.

Several algorithms exist that generate decision trees, such as ID3/4/5, CART, CLS, etc. Of these, the most popular one is ID3, developed by J. R. Quinlan, which uses a top-down, greedy search through the space of possible branches with no backtracking. ID3 employs *Entropy* and *Information Gain* to construct a decision tree. Before we go through the algorithm, let us understand these two terms.

Entropy: Entropy (E) is a measure of disorder, uncertainty, or randomness. If I toss a fair coin, there is an equal chance of getting a head as well as a tail. In other words, we would be most uncertain about the outcome, or we would have a high entropy. The formula of this measurement is:

$$\text{Entropy}(E) = -\sum_{i=1}^{k} p_i \log_2(p_i). \tag{9.11}$$

Here, k is the number of possible class values, and p_i is the number of occurrences of the class i=1 in the dataset. So, in the "balloons" dataset the number of possible class values is 2

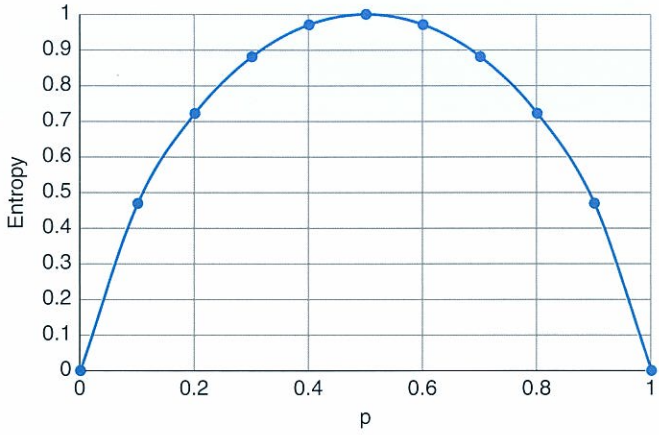

Figure 9.7 Depiction of entropy.

(T or F). The reason for the minus signs is that the logarithms of fractions $p_1, p_2, \ldots, p_n$ are negative, so the entropy is actually positive. Usually the logarithms are expressed in base 2, and then the entropy is in units called bits – just the usual kind of bits used with computers.

Figure 9.7 shows the entropy curve with respect to probability values for an event. As you can see, it is at its highest (1) when the probability of a two-outcome event is 0.5. If we are holding a fair coin, the probability of getting a head or a tail is 0.5. Entropy for this coin is the highest at this point, which reflects the highest amount of uncertainty we will have with this coin's outcome. If, on the other hand, our coin is completely unfair and flips to "heads" every time, the probability of a getting heads with this coin will be 1 and the corresponding entropy will be 0, indicating that there is no uncertainty about the outcome of this event.

Information Gain: If you thought it was not going to rain today and I tell you it will indeed rain, would you not say that you gained some information? On the other hand, if you already knew it was going to rain, then my prediction will not really impact your existing knowledge much. There is a mathematical way to measure such **information gain**:

$$\text{IG}(A, B) = \text{Entropy}(A) - \text{Entropy}(A, B). \tag{9.12}$$

Here, information gain achieved by knowing B along with A is the difference between the entropy (uncertainty) of A and both A and B. Keep this in the back of your mind and we will revisit it as we work through an example next.

But first, let us get back to that decision tree algorithm. A decision tree is a hierarchical, top-down tree, built from a root node to leaves, and involves partitioning the data into smaller subsets that contain instances with similar values (homogeneous). The ID3 algorithm uses entropy to calculate the homogeneity of a sample. If the sample is completely homogeneous, the entropy is zero, and if the sample is equally divided, it has entropy of one. In other words, entropy is a measurement of disorder in the data.

Now, to build the decision tree, we need to calculate two types of entropy using frequency tables, as follows:

a. Entropy using the frequency table of one attribute:

Inflated	
True	False
8	12

Therefore,

$$\begin{aligned} E(\text{Inflated}) &= E(12, 8) \\ &= E(0.6, 0.4) \\ &= -(0.6\log_2 0.6) - (0.4\log_2 0.4) \\ &= 0.4422 + 0.5288 \\ &= 0.9710. \end{aligned}$$

b. Similarly, entropy using the frequency table of two attributes:

$$E(A, B) = \sum_{k \in B} P(k)E(k). \tag{9.13}$$

Following is the explanation. Let us take a look at Table 9.2.
Therefore,

$$\begin{aligned} E(\text{Inflated, Act}) &= P(\text{Dip}) \times E(8, 0) + P(\text{Stretch}) \times E(4, 8) \\ &= \left(\frac{8}{20}\right) \times 0.0 + \left(\frac{12}{20}\right) \times \{-(0.3\log_2 0.3) - (0.7\log_2 0.7)\} \\ &= \left(\frac{12}{20}\right) \times (0.5278 + 0.3813) \\ &= \left(\frac{12}{20}\right) \times 0.9090 \\ &= 0.5454. \end{aligned}$$

As the above value suggests, there is a reduction of entropy from the first case. This decrease of entropy is called information gain. Here, information gain (IG) is:

$$\begin{aligned} \text{IG} &= \text{E(Inflated)} - \text{E(Inflated, Act)} \\ &= 0.9710 - 0.5454 \\ &= 0.4256. \end{aligned}$$

Table 9.2 Frequency table of act and inflated.

		Inflated		
		True	False	
Act	Dip	0	8	8
	Stretch	8	4	12
Total				20

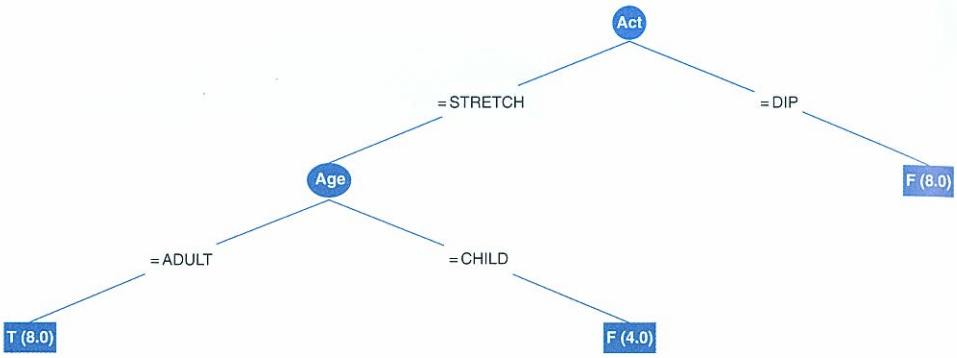

Figure 9.8 Final decision tree for the "balloons" dataset.

If entropy is disorder, then information gain is a measurement of reduction in that disorder achieved by partitioning the original dataset. Constructing a decision tree is all about finding an attribute that returns the highest information gain (i.e., the most homogeneous branches). Following are the steps to create a decision tree based on *entropy* and *information gain:*

Step 1: Calculate entropy of the target or class variable, which is 0.9710, in our case.

Step 2: The dataset is then split on the different attributes into smaller subtables, for example, Inflated and Act, Inflated and Age, Inflated and Size, and Inflated and Color. The entropy for each subtable is calculated. Then it is added proportionally, to get total entropy for the split. The resulting entropy is subtracted from the entropy before the split. The result is the information gain or decrease in entropy.

Step 3: Choose the attribute with the largest information gain as the decision node, divide the dataset by its branches, and repeat the same process on every branch.

If you follow the above guidelines step-by-step, you should end up with the decision tree shown in Figure 9.8.

9.5.1 Decision Rule

Rules are a popular alternative to decision trees. Rules typically take the form of an {IF: THEN} expression (e.g., {IF "condition" THEN "result"}). Typically for any dataset, an individual rule in itself is not a model, as this rule can be applied when the associated condition is satisfied. Therefore, rule-based machine learning methods typically identify a set of rules that collectively comprise the prediction model, or the knowledge base.

To fit any dataset, a set of rules can easily be derived from a decision tree by following the paths from the root node to the leaf nodes, one at a time. For the above decision tree in Figure 9.8, the corresponding decision rules are shown on the left-hand side of Figure 9.9.

Decision rules yield orthogonal hyperplanes in n-dimensional space. What that means is that, for each of the decision rules, we are looking at a line or a plane perpendicular to the axis for the corresponding dimension. This hyperplane (fancy word for a line or a plane in higher dimension) separates data points around that dimension. You can think about it as a

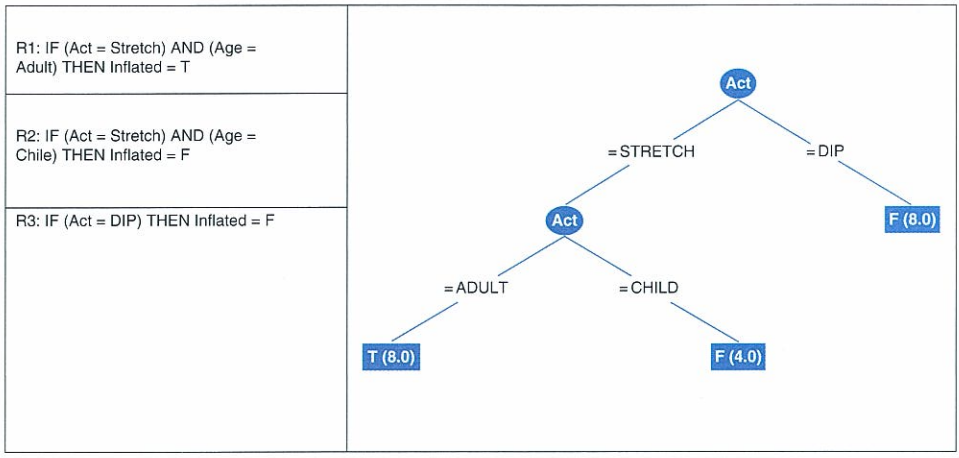

Figure 9.9 Deriving decision rules using a decision tree.

decision boundary. Anything on one side of it belongs to one class, and those data points on the other side belong to another class.

9.5.2 Classification Rule

It is easy to read a set of classification rules directly off a decision tree. One rule is generated for each leaf. The antecedent of the rule includes a condition for every node on the path from the root to that leaf, and the consequent of the rule is the class assigned by the leaf. This procedure produces rules that are unambiguous in that the order in which they are executed is irrelevant. However, in general, rules that are read directly off a decision tree are far more complex than necessary, and rules derived from trees are usually pruned to remove redundant tests. Because decision trees cannot easily express the disjunction implied among the different rules in a set, transforming a general set of rules into a tree is not quite so straightforward. A good illustration of this occurs when the rules have the same structure but different attributes, such as:

If a and b then x
If c and d then x

Then it is necessary to break the symmetry and choose a single test for the root node. If, for example, "if a" is chosen, the second rule must, in effect, be repeated twice in the tree. This is known as the replicated subtree problem.

9.5.3 Association Rule

Association rules are no different from classification rules except that they can predict any attribute, not just the class, and this gives them the freedom to predict combinations of attributes, too. Also, association rules are not intended to be used together as a set, as classification rules are. Different association rules express different regularities that underlie the dataset, and they generally predict different things.

Table 9.3 Weather data.

Outlook	Temperature	Humidity	Windy	Play
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Overcast	Hot	High	False	Yes
Rainy	Mild	High	False	Yes
Rainy	Cool	Normal	False	Yes
Rainy	Cool	Normal	True	No
Overcast	Cool	Normal	True	Yes
Sunny	Mild	High	False	No
Sunny	Cool	Normal	False	Yes
Rainy	Mild	Normal	False	Yes
Sunny	Mild	Normal	True	Yes
Overcast	Mild	High	True	Yes
Overcast	Hot	Normal	False	Yes
Rainy	Mild	High	True	No

Because so many different association rules can be derived from even a very small dataset, interest is restricted to those that apply to a reasonably large number of instances and have a reasonably high accuracy on the instances to which they apply. The coverage of an association rule is the number of instances for which it predicts correctly; this is often called its support. Its accuracy – often called confidence – is the number of instances that it predicts correctly, expressed as a proportion of all instances to which it applies.

For example, consider the weather data in Table 9.3, a training dataset of weather and corresponding target variable "Play" (suggesting possibilities of playing). The decision tree and the derived decision rules for this dataset are given in Figure 9.10.

Let us consider this rule:

If *temperature* = *cool* then *humidity* = *normal*.

The coverage is the number of days that are both cool and have normal humidity (four in the data of Table 9.3), and the accuracy is the proportion of cool days that have normal humidity (100% in this case). Some other good quality association rules for Figure 9.10 are:

If *humidity* = *normal* and *windy* = *false* then *play* = *yes*.
If *outlook* = *sunny* and *play* = *no* then *windy* = *true*.
If *windy* = *false* and *play* = *no* then *outlook* = *sunny* and *humidity* = *high*.

See if these rules make sense by going through the tree in Figure 9.10 as well as logically questioning them (e.g., if it is sunny but very windy outside, will we play?). Also try deriving some new rules.

R1: IF (Outlook = Sunny) AND (Windy = False) Then Play = Yes

R2: IF (Outlook = Sunny) AND (Windy = True) Then Play = No

R3: IF (Outlook = Overcast) THEN Play = Yes

R4: IF (Outlook = Rainy) AND (Humidity = High) THEN Play = Yes

R5: IF (Outlook = Rainy) AND (Humidity = Normal) THEN Play = Yes

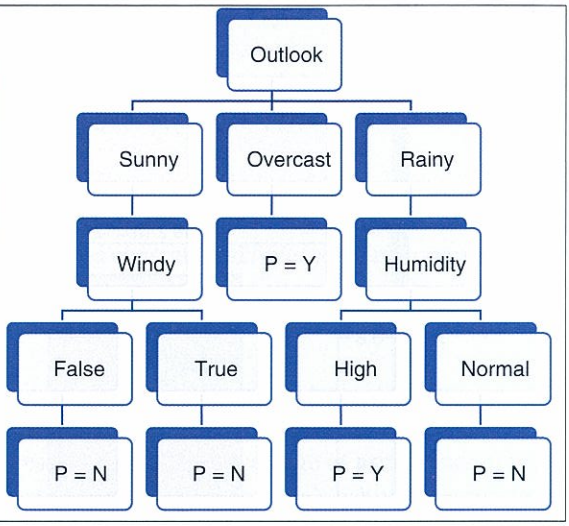

Figure 9.10 Decision rules (left) and decision tree (right) for the weather data.

Hands-On Example 9.4: Decision Tree

Let us see how a decision tree-based classifier can be implemented with R. For this demonstration we are going to use the other balloons dataset that you can download from OA 9.7. This is from the repository that we have used in the example earlier (see Table 9.1).

The first step is to import the data into RStudio. It is always recommended that, before performing any operation, the data types of all the attributes are checked first (one shown here).

```
> balloon <- read.csv("yellow-small.csv", sep = ',',
header = TRUE)
> View(balloon)
> is.factor(balloon$Size)
```

We are going to use the "party" package here. You have to install this package before you proceed to the next step.

```
> install.packages("party")
> library(party)
```

In the next step we are going to create the decision tree:

```
> View(balloon)
> inflated.Tree<- ctree(Inflated~., data = balloon)
```

To plot the tree, execute the following line:

```
> plot(inflated.Tree)
```

If you have correctly followed the steps, you should see a tree like the one in Figure 9.11. From the tree, it can be concluded that color is a good predictor if the balloon is inflated or not.

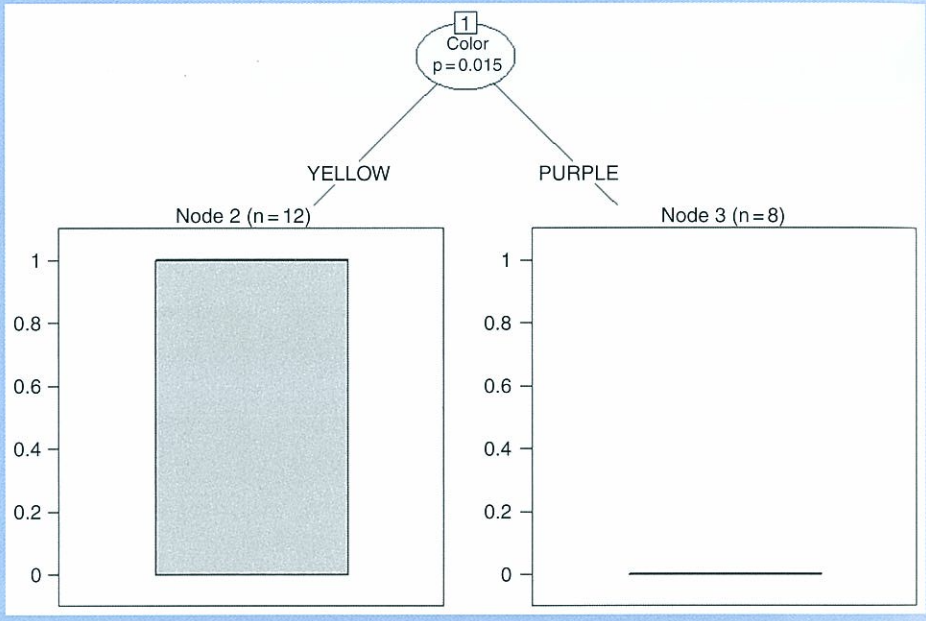

Figure 9.11 Plotting the decision tree.

Try It Yourself 9.4: Decision Tree

Let us test your understanding of a decision tree algorithm with all categorical variables. The dataset you are going to use is about contact lenses (download from OA 9.8), which has three class labels:

1. The patient should be prescribed hard contact lenses.
2. The patient should be prescribed soft contact lenses.
3. The patient should not be fitted with contact lenses.

Build a decision tree-based classifier that would recommend the class label based on the other attributes from the dataset.

9.6 Random Forest

A decision tree seems like a nice method for doing classification – it typically has a good accuracy, and, more importantly, it provides human-understandable insights. But one big problem the decision tree algorithm has is that it could overfit the data. What does that mean? It means it could try to model the given data so well that, while the classification accuracy on that dataset would be wonderful, the model may find itself crippled when looking at any new data; it learned *too much* from the data!

One way to address this problem is to use not just one, not just two, but many decision trees, each one created slightly differently. And then take some kind of average from what these trees decide and predict. Such an approach is so useful and desirable in many situations where there is a whole set of algorithms that apply them. They are called **ensemble methods**.

In machine learning, ensemble methods rely on multiple learning algorithms to obtain better prediction accuracy than what any of the constituent learning algorithms can achieve. In general, an ensemble algorithm consists of a concrete and finite set of alternative models but incorporates a much more flexible structure among those alternatives. One example of an ensemble method is random forest, which can be used for both regression and classification tasks.

Random forest operates by constructing a multitude of decision trees at training time and selecting the mode of the class as the final class label for classification or mean prediction of the individual trees when used for regression tasks. The advantage of using random forest over decision tree is that the former tries to correct the decision tree's habit of overfitting the data to their training set. Here is how it works.

For a training set of N, each decision tree is created in the following manner:

1. A sample of the N training cases is taken at random but with replacement from the original training set. This sample will be used as a training set to grow the tree.
2. If the dataset has M input variables, a number m (m being a lot smaller than M) is specified such that, at each node, m variables are selected at random out of M. Among this m, the best split is used to split the node. The value of m is held constant while we grow the forest.
3. Following the above steps, each tree is grown to its largest possible extent and there is no pruning.
4. Predict new data by aggregating the predictions of the n trees (i.e., majority votes for classification, average for regression).

Let us say a training dataset, N, has four observations on three predictor variables, namely A, B, and C. The training data is provided in Table 9.4.

We will now work through the random forest algorithm on this small dataset.

Step 1: Sample the N training cases at random. So these subsets of N, for example, n_1, n_2, n_3, . . ., n_n (as depicted in Figure 9.12) are used for growing (training) the n number of decision trees. These samples are drawn as randomly as possible, with or without

Table 9.4 Training dataset.

	Independent variables		
	A	B	C
Training instances	A_1	B_1	C_1
	A_2	B_2	C_2
	A_3	B_3	C_3
	A_4	B_4	C_4

Table 9.5 Selection of attributes for the tree.

Input variables	Training set
A	n_1
	n_2
B	n_3
	.
C	.
	n_5

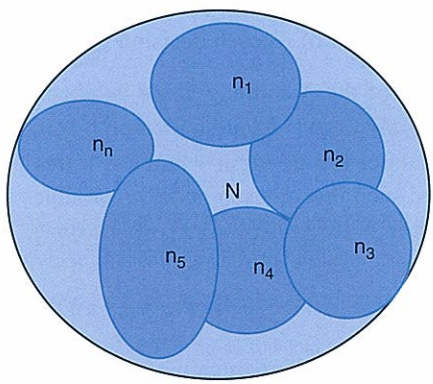

Figure 9.12 Sampling the training set.

overlap between them. For example, n_1 may consist of the training instances 1, 1, 1, and 4. Similarly, n_2 may consist of 2, 3, 3, and 4, and so on.

Step 2: Out of the three predictor variables, a number $m \ll 3$ is specified such that, at each node, m variables are selected at random out of the M. Let us say here m is 2. So, n_1 can be trained on A, B; n_2 can be trained on B, C; and so on (see Table 9.5).

So, the resultant decision trees may look something like what is shown in Figure 9.13.

Random forest uses a bootstrap sampling technique, which involves sampling of the input data with replacement. Before using the algorithm, a portion of the data (typically one-third) that is not used for training is set aside for testing. These are sometimes known as out-of-bag samples. An error estimation on this sample, known as out-of-bag error, provides evidence that the out-of-bag estimate can be as accurate as having a test set of equal size as the training set. Thus, use of an out-of-bag error estimate removes the need for a set-aside test set here.

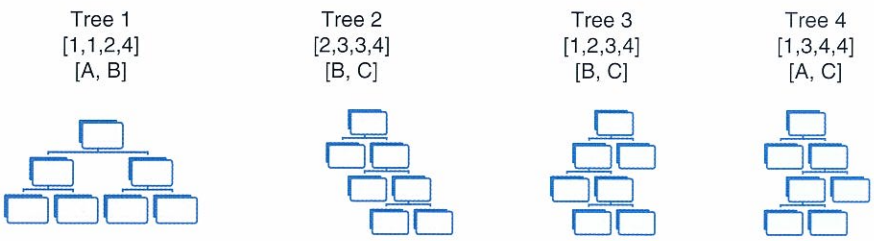

Figure 9.13 Various decision trees for the random forest data.

So, the big question is why does the random forest as a whole do a better job than the individual decision trees? Although there is no clear consensus among researchers, there are two major beliefs behind this:

1. As the saying goes, "Nobody knows everything, but everybody knows something." When it comes to a forest of trees, not all of them are perfect or most accurate. Most of the trees provide correct predictions of class labels for most of the data. So, even if some of the individual decision trees generate wrong predictions, the majority predict correctly. And since we are using the mode of output predictions to determine the class, it is unaffected by those wrong instances. Intuitively, validating this belief depends on the randomness in the sampling method. The more random the samples, the more decorrelated the trees will be, and the less likely are the chances of other trees being affected by wrong predictions from the other trees.
2. More importantly, different trees are making mistakes at different places and not all of them are making errors at the same location. Again, intuitively, this belief depends on how randomly the attributes are selected. The more random they are, the less likely the trees will make mistakes at the same location.

Hands-On Example 9.5: Random Forest

We will now take an example and see how to use random forest in R. For this, we are going to use the *Bank Marketing* dataset from the University of California, Irvine, machine learning dataset, which you can download from OA 9.9. Given the dataset, the goal is to predict if the client will subscribe (yes/no) a term deposit (variable *y*).

Let us import the dataset to RStudio first. Note that in the original dataset the columns are separated by semicolons. If you are not familiar with how to handle semicolon-delimited data, you may want to convert it into .csv or .tsv, or the format you are familiar with the most.

```
> bank <- read.csv(file.choose(), header = TRUE, sep = ";")
> View(bank)
> barplot(table(bank$y))
```

The above line of code generates the barplot shown in Figure 9.14.

As the barplot depicts, the majority of the data points in this dataset have class label "no." Now, before we build our model, let us separate the dataset into training and test instances:

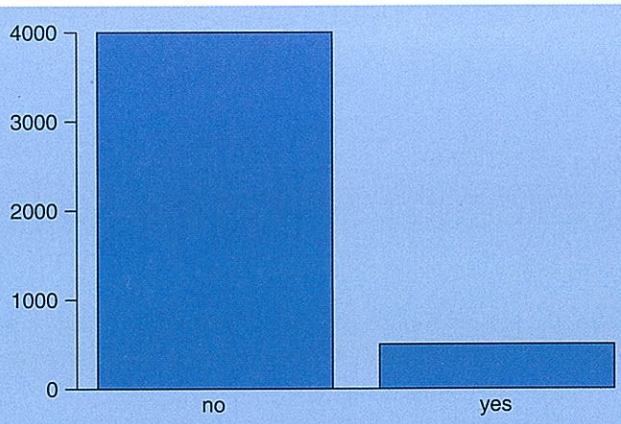

Figure 9.14 Bar plot depicting data points with "No" and "Yes" labels.

```
> set.seed(1234)
> population <- sample(nrow(bank), 0.75 * nrow(bank))
> train <- bank[population, ]
> test <- bank[-population, ]
```

As shown, the dataset is split into two parts: 75% for training purposes and the remaining to evaluate our model. To build the model, the *randomForest* library is used. In case your system does not have this package, make sure to install it first. Next, use the training instances to build the model:

```
> install.packages("randomForest")
> library(randomForest)
> model <- randomForest(y ~ ., data = train)
> model
```

We can use ntree and mtry to specify the total number of trees to build (default = 500), and the number of predictors to randomly sample at each split, respectively. The above lines of code should generate the following result:

```
Call:
 randomForest(formula = y ~ ., data = train)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 4
       OOB estimate of error rate: 9.94%
Confusion matrix:
      no yes class.error
no  2901  97   0.0323549
yes  240 152   0.6122449
```

We can see that 500 trees were built, and the model randomly sampled four predictors at each split. It also shows a confusion matrix containing prediction vs. actual, as well as classification error for each class. Let us test the model to see how it performs on the test dataset:

```
> prediction <- predict(model, newdata = test)
> table(prediction, test$y)
```

The following output is produced:

```
prediction  no  yes
       no  964   84
       yes  38   45
```

We can further evaluate the accuracy as follows:

```
> (964 + 45)/nrow(test)
[1] 0.8921309
```

There you have it. We achieved about 89% accuracy with a very simple model. We can try to improve the accuracy by feature selection, and also by trying different values of ntree and mtry.

Random forest is considered a *panacea* of all data science problems among most of its practitioners. There is a belief that when you cannot think of any algorithm irrespective of situation, use random forest. It is a bit irrational, since no algorithm strictly dominates in all applications (one size does not fit all). Nonetheless, people have their favorite algorithms. And there are reasons why, for many data scientists, random forest is the favorite:

1. It can solve both types of problems, that is, classification and regression, and does a decent estimation for both.
2. Random forest requires almost no input preparation. It can handle binary features, categorical features, and numerical features without any need for scaling.
3. Random forest is not very sensitive to the specific set of parameters used. As a result, it does not require a lot of tweaking and fiddling to get a decent model; just use a large number of trees and things will not go terribly awry.
4. It is an effective method for estimating missing data and maintains accuracy when a large proportion of the data are missing.

So, is random forest a *silver bullet*? Absolutely not. First, it does a good job at classification but not as good as for regression problems, since it does not give precise continuous nature predictions. Second, random forest can feel like a black-box approach for statistical modelers, as you have very little control on what the model does. At best, you can try different parameters and random seeds and hope that will change the output.

Try It Yourself 9.5: Random Forest

Get the balloons dataset presented in Table 9.1 and downloadable from OA 9.6. Use this dataset and create a random forest model to classify if the balloons are inflated or not from the available attributes. Compare the performance (e.g., accuracy) of this model against that of one created using a decision tree.

9.7 Naïve Bayes

We now move on to a very popular and robust approach for classification that uses Bayes' theorem. The Bayesian classification represents a supervised learning method as well as a statistical method for classification. In a nutshell, it is a classification technique based on Bayes' theorem with an assumption of independence among predictors. Here, all attributes contribute equally and independently to the decision.

In simple terms, a Naïve Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about three inches in diameter. Even if these features depend on each other or upon the existence of other features, all of these properties independently contribute to the probability that this fruit is an apple, and that is why it is known as *naïve*. It turns out that in most cases, while such a naïve assumption is found to be not true, the resulting classification models do amazingly well.

Let us first take a look at Bayes' theorem, which provides a way of calculating posterior probability $P(c|\,x)$ from $P(c)$, $P(x)$, and $P(x|\,c)$. Look at the equation below:

$$P(c|x) = \frac{P(x|c)P(c)}{P(x)}. \tag{9.14}$$

Here:

- $P(c|\,x)$ is the posterior probability of *class* (*c*, *target*) given *predictor* (*x*, *attributes*).
- $P(c)$ is the prior probability of *class*.
- $P(x|\,c)$ is the likelihood, which is the probability of *predictor* given *class*.
- $P(x)$ is the prior probability of *predictor*.

And here is that naïve assumption: we believe that evidence can be split into parts that are independent,

$$P(c|x) = \frac{P(x_1|c)P(x_2|c)P(x_3|c)P(x_4|c)\ldots P(x_n|c)P(c)}{P(x)}, \tag{9.15}$$

where $x_1, x_2, x_3, \ldots, x_n$ are independent priors.

To understand Naïve Bayes in action, let us revisit the golf dataset from earlier in this chapter to see how this algorithm works step-by-step. This dataset is repeated in reordered form in Table 9.6, and can be downloaded from OA 9.10.

Table 9.6 Weather dataset.

Outlook	Temperature	Humidity	Windy	Play
Overcast	Hot	High	False	Yes
Overcast	Cool	Normal	True	Yes
Overcast	Mild	High	True	Yes
Overcast	Hot	Normal	False	Yes
Rainy	Mild	High	False	Yes
Rainy	Cool	Normal	False	Yes
Rainy	Cool	Normal	True	No
Rainy	Mild	Normal	False	Yes
Rainy	Mild	High	True	No
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Sunny	Mild	High	False	No
Sunny	Cool	Normal	False	Yes
Sunny	Mild	Normal	True	Yes

(The source of this dataset is http://storm.cis.fordham.edu/~gweiss/data-mining/weka-data/weather.arff)

As shown in Table 9.6, the dataset has four attributes, namely *Outlook*, *Temperature*, *Humidity*, and *Windy*, which are all different aspects of weather conditions. Based on these four attributes, the goal is to predict the value of the outcome variable, *Play* (yes or no) – whether the weather is suitable to play golf. Following are the steps of the algorithm through which we could accomplish that goal.

- Step 1: First convert the dataset into a frequency table (see Figure 9.15).
- Step 2: Create a likelihood table by finding the probabilities, like probability of being hot is 0.29 and probability of playing is 0.64 as shown in Figure 9.15.
- Step 3: Now, use the Naïve Bayesian equation to calculate the posterior probability for each class. The class with the highest posterior probability is the outcome of the prediction.

To see this in action, let us say that we need to decide if one should go out to play when the weather is mild based on the dataset. We can solve it using the above discussed method of posterior probability. Using Bayes' theorem:

$$P(\text{Yes}|\text{Mild}) = \frac{P(\text{Mild}|\text{Yes}) \times P(\text{Yes})}{P(\text{Mild})}$$

Here we have:

$$P(\text{Mild}|\text{Yes}) = 4/9 = 0.44,$$
$$P(\text{Mild}) = 6/14 = 0.43,$$

Temperature	Play
Hot	no
Hot	no
Hot	yes
Mild	yes
Cool	yes
Cool	no
Cool	yes
Mild	no
Cool	yes
Mild	yes
Mild	yes
Mild	yes
Hot	yes
mild	no

Frequency Table

Temperature	No	Yes
Hot	2	2
Mild	2	4
Cool	1	3
Total	5	9

Likelihood Table

Temperature	No	Yes		
Hot	2	2	4/14	0.29
Mild	2	4	6/14	0.43
Cool	1	3	4/14	0.29
All	5	9		
	5/14	9/14		
	0.36	0.64		

Figure 9.15 Conversion of the dataset to a frequency table and to a likelihood table.

Now,

$$P(\text{Yes}|\text{Mild}) = (0.44 \times 0.64)/0.43 = 0.65.$$

In other words, we have derived that the probability of playing when the weather is mild is 65%, and if we wanted to turn that into a Yes–No decision, we can see that this probability is higher than the mid-point, that is, 50%. Thus, we can declare "Yes" for our answer.

Naïve Bayes uses a similar method to predict the probability of different classes based on various attributes. This algorithm is mostly used in text classification and with problems having two or more classes. One prominent example is spam detection. Spam filtering with Naïve Bayes is a two-classes problem, that is, to determine a message or an email as spam or not. Here is how it works.

Let us assume that there are certain words (e.g., "viagra," "rich," "friend") that indicate a given message as being spam. We can apply Bayes' theorem to calculate the probability that an email is spam given the email words as:

$$\begin{aligned} P(\text{spam}|\text{words}) &= \frac{P(\text{words}|\text{spam}) \times P(\text{spam})}{P(\text{words})} \\ &= \frac{P(\text{spam}) \times P(\text{viagra, rich, \ldots, friend}|\text{spam})}{P(\text{viagra, rich, \ldots, friend})} \\ &\propto P(\text{spam}) \times P(\text{viagra, rich, \ldots, friend}|\text{spam}). \end{aligned}$$

Here, $\propto$ is the proportion symbol.

According to Naïve Bayes, the word events are completely independent; therefore, simplifying the above formula using the Bayes formula would look like:

$$P(\text{spam}|\text{words}) \propto P(\text{viagra}|\text{spam}) \times P(\text{rich}|\text{spam}) \times \ldots \times P(\text{friend}|\text{spam})$$

Now, we can calculate P(viagra|spam), P(rich|spam), and P(friend|spam) each individually if we have a sizeable training dataset of previously categorized spam messages and the occurrences of these words ("viagra," "rich," "friend," etc.) in the training set. So, it is possible to determine the probability of an email from the test set as spam based on these values.

Naïve Bayes works surprisingly well even if the independence assumption is clearly violated because classification does not need accurate probability estimates so long as the greatest probability is assigned to the correct class. Naïve Bayes affords fast model building and scoring and can be used for both binary and multiclass classification problems.

Hands-On Example 9.6: Naïve Bayes

Let us use the golf data in Table 9.6 (available to download from OA 9.10) to explore how to perform Naïve Bayes classification in R.

First, you need to import the dataset into RStudio:

```
> golf <- read.csv(file = "golf.csv", header = TRUE, sep = ",")
> View(golf)
```

There are a bunch of packages in R that support Naïve Bayes classification. For this example, you are going to use the package "e1071." In case you do not have it in your system, make sure you install it first.

```
> install.packages("e1071")
> library(e1071)
```

Once you do that, building the Naïve Bayes model in R is simple and straightforward. However, when building a model, you will need test data to evaluate your model. Since you do not have that here, let us separate training and test data from your original dataset. Here is how to do that:

```
> set.seed(123)
> id <- sample(2, nrow(golf), prob = c(0.7, 0.3), replace = T)
> golfTrain <- golf[id ==1,]
> golfTest <- golf[id ==2,]
```

You check the training and test sets that you just built. From the original data, which have 14 data points, you reserved 70% for training; you should have nine data points in the training set and the remaining in the test set. Which individual data points will go into the training and which will go to the test depends on how you sample the data. You can explore more about both these datasets from the console.

```
> View(golfTest)
> View(golfTrain)
```

Next, let us build the model on the training data and evaluate it.

```
> golfModel <- naiveBayes(PlayGolf~., data = golfTrain)
> print(golfModel)
```

This should generate output similar to the following.

```
Naive Bayes Classifier for Discrete Predictors
Call:
naiveBayes.default(x = X, y = Y, laplace = laplace)

A-priori probabilities:
Y
       No        Yes
0.3333333  0.6666667

Conditional probabilities:
     Outlook
Y       Overcast      Rainy      Sunny
  No  0.0000000  0.3333333  0.6666667
  Yes 0.6666667  0.1666667  0.1666667

     Temp
Y          Cool        Hot       Mild
No   0.3333333  0.3333333  0.3333333
Yes  0.3333333  0.3333333  0.3333333

     Humidity
Y          High     Normal
No   0.6666667  0.3333333
Yes  0.3333333  0.6666667

     Windy
Y         FALSE       TRUE
No   0.3333333  0.6666667
Yes  0.6666667  0.3333333
```

The output contains a likelihood table as well as *a-priori* probabilities. The *a-priori* probabilities are equivalent to the prior probability in Bayes' theorem. That is, how frequently each level of class occurs in the training dataset. The rationale underlying the prior probability is that, if a level is rare in the training set, it is unlikely that such a level will occur in the test dataset. In other words, the prediction of an outcome is influenced not only by the predictors, but also by the prevalence of the outcome.

Let us move on to an evaluation of this model. You have to use the test set for this, the data which the algorithm did not see while training.

```
> prediction <- predict(golfModel, newdata = golfTest)
```

You can check what class labels your model has predicted for all the data points in the test data:

```
> print(prediction)
```

However, it will be easier if we can compare these predicted labels with actual labels side by side, or in a confusion matrix. Fortunately, in R there is a package for this functionality, named *caret*. Again, if you do not have it make sure you install it first. Keep in mind that installing *caret* requires some prerequisite

packages such as *lattice* and *ggplot2*. Make sure you install them first. Once you have all of them, use the following command:

```
> confusionMatrix(prediction, golfTest$PlayGolf)
```

And you will have a nice confusion matrix along with *p*-values and all other evaluation matrices.

```
Confusion Matrix and Statistics

          Reference
Prediction  No Yes
       No    2   2
       Yes   0   1

               Accuracy : 0.6
                 95% CI : (0.1466, 0.9473)
    No Information Rate : 0.6
    P-Value [Acc > NIR] : 0.6826

                  Kappa : 0.2857
 Mcnemar's Test P-Value : 0.4795

            Sensitivity : 1.0000
            Specificity : 0.3333
         Pos Pred Value : 0.5000
         Neg Pred Value : 1.0000
             Prevalence : 0.4000
         Detection Rate : 0.4000
   Detection Prevalence : 0.8000
      Balanced Accuracy : 0.6667

       'Positive' Class : No
```

As you can see, the accuracy is not great, 60%, which can be attributed to the fact that we had only nine examples in your training set. Still, by now, you should have some idea about how to build Naïve Bayes classifier in R.

Try It Yourself 9.6: Naïve Bayes

Use the contact lenses dataset (OA 9.8) from the decision tree problem under Try It Yourself 9.4 and build a Naïve Bayes classifier to predict the class label. Compare the accuracy between the Naïve Bayes algorithm and the decision tree.

9.8 Support Vector Machine (SVM)

Now we come to the last method for classification in this chapter. One thing that has been common in all the classifier models we have seen so far is that they assume linear separation of classes. In other words, they try to come up with a decision boundary that is a line (or a hyperplane in a higher dimension). But many problems do not have such linear characteristics. Support vector machine (SVM) is a method for the classification of both linear and nonlinear data. SVMs are considered by many to be the best stock classifier for doing machine learning tasks. By stock, here we mean in its basic form and not modified. This means you can take the basic form of the classifier and run it on the data, and the results will have low error rates. Support vector machines make good decisions for data points that are outside the training set. In a nutshell, an SVM is an algorithm that uses nonlinear mapping to transform the original training data into a higher dimension. Within this new dimension, it searches for the linear optimal separating hyperplane (i.e., a decision boundary separating the tuples of one class from another). With an appropriate nonlinear mapping to a sufficiently high dimension, data from two classes can always be separated by a hyperplane. The SVM finds this hyperplane using support vectors ("essential" training tuples) and margins (defined by the support vectors).

To understand what this means, let us look at an example. Let us start with a simple one, a two-class problem.

Let the dataset D be given as $(X_1, y_1), (X_2, y_2), \ldots, (X_{|D|}, y_{|D|})$, where X_i is the set of training tuples with associated class labels y_i. Each y_i can take one of two values, either $+1$ or -1 (i.e., $y_i \in \{+1, -1\}$), corresponding to the classes represented by the hollow red squares and blue circles (ignore the data points represented by the solid symbols for now), respectively, in Figure 9.16. From the graph, we see that the 2-D data are linearly separable (or "linear," for short), because a straight line can be drawn to separate all the tuples of class $+1$ from all the tuples of class -1.

Note that if our data had three attributes (two independent variables and one dependent), we would want to find the best separating plane (a demonstration is shown in Figure 9.17 for linearly non-separable data). Generalizing to n dimensions, if we had n number of attributes, we want to find the best $(n-1)$-dimensional plane, called a *hyperplane*. In general, we will use the term *hyperplane* to refer to the decision boundary that we are searching for, regardless of the number of input attributes. So, in other words, our problem is, how can we find the best hyperplane?

There are an infinite number of separating lines that could be drawn. We want to find the "best" one, that is, one that (we hope) will have the minimum classification error on previously unseen tuples. How can we find this best line?

An SVM approaches this problem by searching for the maximum marginal hyperplane. Consider Figure 9.18, which shows two possible separating hyperplanes and their associated margins. Before we get into the definition of margins, let us take an intuitive look at this Figure 9.18. Both hyperplanes can correctly classify all the given data tuples.

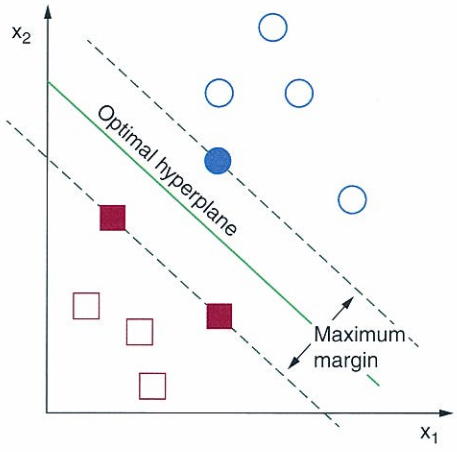

Figure 9.16 Linearly separable data.[1]

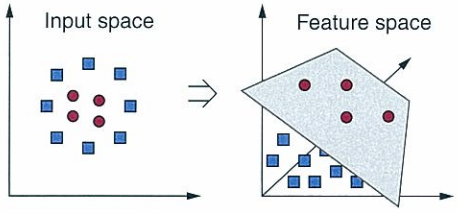

Figure 9.17 From line to hyperplane. (Source: Jiawei Han and Micheline Kamber. (2006). *Data Mining: Concepts and Techniques*. Morgan Kaufmann.)

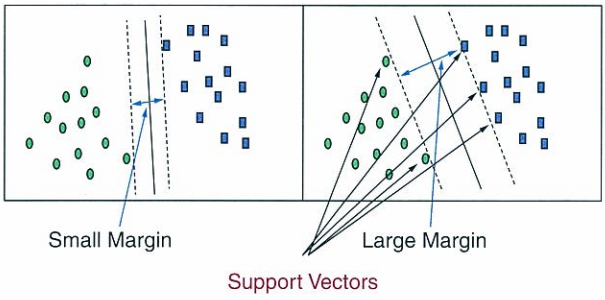

Figure 9.18 Possible hyperplanes and their margins. (Source: Jiawei Han and Micheline Kamber. (2006). *Data Mining: Concepts and Techniques*. Morgan Kaufmann.)

Intuitively, however, we expect the hyperplane with the larger margin to be more accurate at classifying future data tuples than the hyperplane with the smaller margin. This is why (during the learning or training phase) the SVM searches for the hyperplane with the largest margin, that is, the maximum marginal hyperplane (MMH). The associated margin gives the largest separation between classes.

Roughly speaking, we would like to find the point closest to the separating hyperplane and make sure this is as far away from the separating line as possible. This is known as "margin."

The points closest to the separating hyperplane are known as support vectors. We want to have the greatest possible margin, because if we made a mistake or trained our classifier on limited data, we would want it to be as robust as possible. Now that we know that we are trying to maximize the distance from the separating line to the support vectors, we need to find a way to optimize this problem; that is, how do we find an SVM with the MMH and the support vectors. Consider this: a separating hyperplane can be written as

$$f(x) = \beta_0 + \beta^T x, \tag{9.16}$$

where T is a weight vector, namely, $T = \{1,2,\ldots, n\}$, n is the number of attributes, and β_0 is a scalar, often referred to as a bias.[2] The optimal hyperplane can be represented in an infinite number of different ways by scaling of β and β_0. As a matter of convention, among all the possible representations of the hyperplane, the one chosen is

$$\beta_0 + \beta^T x = 1, \tag{9.17}$$

where x symbolizes the training examples closest to the hyperplane. In general, the training examples that are closest to the hyperplane are called support vectors. This representation is known as the canonical hyperplane.

Now, we know from geometry that the distance d between a point (m, n) and a straight line represented by $Ax + By + C = 0$ is given by

$$d = \frac{|Am + Bn + C|}{\sqrt{A^2 + B^2}}. \tag{9.18}$$

Therefore, extending the same equation to a hyperplane gives the distance between a point x and a hyperplane:

$$d = \frac{|\beta_0 + \beta^T x|}{\|\beta\|}. \tag{9.19}$$

In particular, for the canonical hyperplane, the numerator is equal to one and the distance to the support vectors is

$$d_{\text{support vectors}} = \frac{|\beta_0 + \beta^T x|}{\|\beta\|}. \tag{9.20}$$

Now, the margin M is twice the distance to the closest examples. So

$$M = \frac{2}{\|\beta\|}. \tag{9.21}$$

Finally, the problem of maximizing M is equivalent to the problem of minimizing a function $L()$ subject to some constraints. The constraints model the requirement for the hyperplane to classify correctly all the training examples x_i. Formally,

$$\min_{\beta,\beta_0} L(\beta) = \frac{1}{2}\|\beta\|^2 \text{ subject to } y_i(\beta^T x_i + \beta_0) \geq 1 \ \forall i, \tag{9.22}$$

where y_i represents each of the labels of the training examples. This is a problem of Lagrangian optimization that can be solved using Lagrange multipliers to obtain the weight vector and the bias β_0 of the optimal hyperplane.

This was the SVM theory in a nutshell, which is given here with a primary purpose of developing intuition behind how SVMs and in general such maximum marginal classifiers work. But this is a book that covers hands-on data science, so let us see how we could use SVM for a classification or a regression problem.

Hands-On Example 9.7: SVM

Consider a simple classification problem using regression.csv data (Figure 9.19, downloaded from OA 9.11) with just two attributes *X* and *Y*.

We can now use R to display the data and fit a line:

```
# Load the data from the csv file
data <- read.csv('regression.csv', header = TRUE, sep=",")
# Plot the data
plot(data, pch=16)
# Create a linear regression model
```

1	X, Y
2	1, 3
3	2, 4
4	3, 8
5	4, 4
6	5, 6
7	6, 9
8	7, 8
9	8, 12
10	9, 15
11	10, 26
12	11, 35
13	12, 40
14	13, 45
15	14, 54
16	15, 49
17	16, 59
18	17,60
19	18,62
20	19, 63
21	20, 68

Figure 9.19 The regression.csv file.

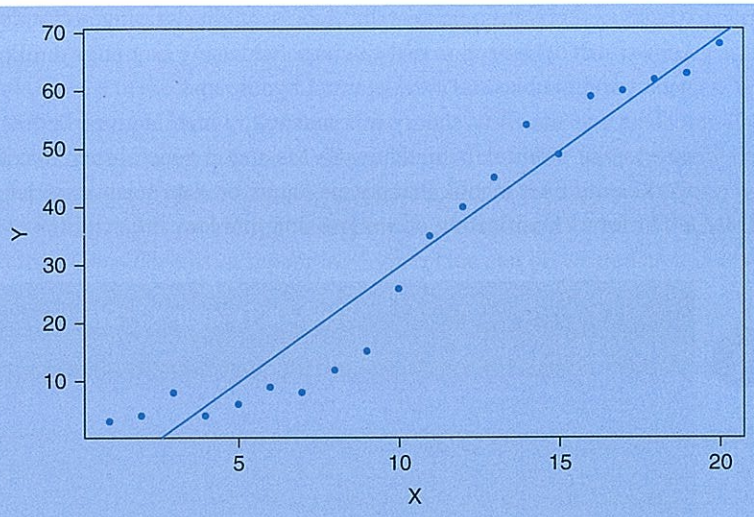

Figure 9.20 Regression line fitted onto the data.

```
model <- lm(y ~ x, data)
# Add the fitted line
abline(model)
```

Here, we have tried to use a linear regression model first to classify the data as shown in Figure 9.20. In order to be able to compare the linear regression with the support vector machine, first we need a way to measure how good it is.

To do that we will change our code just a little to visualize each prediction made by our model:

```
# re-plot the original data points
plot(data, pch=16)
# make a prediction for each X
predictedY <- predict(model, data)
# display the predictions
points(data$x, predictedY, col = "blue", pch=4)
```

This produces the graph shown in Figure 9.21.

For each data point x_i the model makes a prediction y_i displayed as a blue cross on the graph. The only difference with the previous graph is that the dots are not connected to each other.

In order to measure how good our model is, we will compute how many errors it makes, which we can accomplish by calculating the root mean square error (RMSE). The way to calculate RMSE in R is:

```
rmse <- function(error)
{
  sqrt(mean(error^2))
}
```

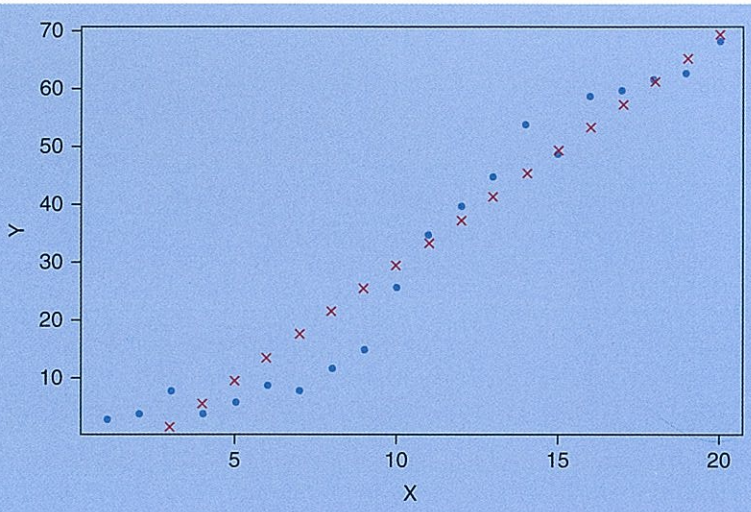

Figure 9.21 Predicted outcomes using linear regression plotted with the original data.

```
error <- model$residuals # same as data$Y - predictedY
lrPredictionRMSE <- rmse(error)
```

The RMSE value of our linear regression model that we have obtained from R is 5.70. Let us try to improve it with SVM.

Prerequisite: In order to create an SVM model with R, you will need the package "e1071." So before proceeding to the following steps, be sure to install it and add it to the library at the start of your file.[3]

Below is the code to create a model with SVM in R, once the "e1071" package is installed and loaded in the current session:

```
model2 <- svm(y ~ x, data)
predictedY <- predict(model2, data)
points(data$x, predictedY, col = "red", pch=4)
```

The code draws the graph shown in Figure 9.22.

This time the prediction is much closer to the real values. Let us compute the RMSE of our support vector regression model.

```
# /!\ this time svmModel$residuals is not the same as data$y -
predictedY in linear regression example, so we compute the
error like this:
error <- data$y - predictedY
svmPredictionRMSE <- rmse(error) # 3.157061
```

We can see that SVM gives better (lower) RMSE – 3.15 – compared to 5.70 with linear regression.

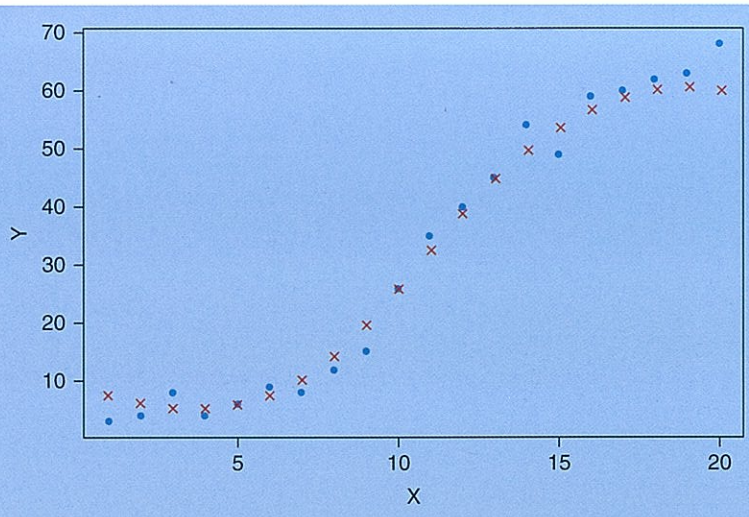

Figure 9.22 Predicted outcomes using SVM plotted with the original data.

Try It Yourself 9.7: SVM

The dataset you are going to use for this work comes from a Combined Cycle power plant and the measures were recorded for a period of six years (2006–2011). You can download it from OA 9.12. The features in this dataset consist of hourly average ambient variables and include

- Temperature (AT) in the range 1.81°C and 37.11°C
- Ambient pressure (AP) in the range 992.89–1033.30 millibar
- Relative humidity (RH) in the range 25.56% to 100.16%
- Exhaust vacuum (V) in the range 25.36–81.56 cm Hg
- Net hourly electrical energy output (PE) 420.26–495.76 MW

Create an SVM-based model that can be used to predict PE from the other four attributes. Note that none of the features are normalized in this dataset.

FYI: Anomaly Detection

In machine learning, anomaly detection (also outlier detection) is used to identify the items, events, or observations that do not conform to other similar items in a dataset. Typically, the anomalous items will not conform to some expected pattern from the dataset and therefore translate to some kind of problem. Such algorithms are used in a broad range of contexts, like identifying fraud in bank transactions, a structural defect in alloys, potential problems in medical records, or errors in a text. Anomalies in other subjects are also referred to as outliers, novelties, noise, deviations and exceptions.

Anomaly detection is applicable in a variety of domains, such as intrusion detection, fraud detection, fault detection, system health monitoring, event detection in sensor networks, and detecting ecosystem

disturbances. For example, any highly unusual credit card spending patterns are suspect. Because the possible variations are so numerous and the training examples so few, it is not feasible to learn what fraudulent activity looks like. The approach that anomaly detection takes is to simply learn what normal activity looks like (using a history of non-fraudulent transactions) and identify anything that is significantly different.

Supervised approaches for anomaly detection include kNN, Bayesian network, decision tree, and support vector machine.

To illustrate anomaly detection, let us run through the process with the kNN clustering algorithm. To detect the outliers using kNN, first you have to train the kNN algorithm by supplying it with data clusters you know to be correct. Therefore, the dataset to be used to train the kNN has to be a different one from the data that will be used to identify the outliers. For example, you want to cluster the house listings that are similar in price range in your chosen locality. Now, all the current listings that you have collected from the Web may not reflect the correct price of the listings. This may happen for several reasons: some of the listings may be outdated, some may have unintentional mistakes in listing price, and some may even contain intentional lower prices for clickbait. Now, you want to do a kNN clustering to discriminate such anomalies from the correct listings. One way to do this would be to collect all the previous correct records of list prices of the current listings and train the kNN clustering algorithm on that dataset first. Since I have covered kNN clustering with examples before, I will leave this part to you. Once the trained model is generated, the same can be used to identify the data points in the current listings dataset that do not properly fit into any cluster and thus ought to be identified as anomalies.

Summary

This was the longest chapter in this book and there is a reason for it. Supervised learning, and classification in particular, covers a big portion of today's data problems. Many of the problems that we encounter in the real world require us to analyze the data to provide decision-making insights. Which set of features will be acceptable to our customers for the next version of our app? Is an incoming message spam or not? Should we trust that news story or is it fake? Where should we place this new wine we are going to start selling – "premium" (high quality), "great value" (medium quality), or "great deal" (low quality)?

In this chapter we started with logistic regression as a popular technique for doing binary decision-making; such two-class classification happens to cover a large range of possibilities. But, of course, there are situations that require us to consider more than two classes. For them, we saw several techniques: softmax regression to kNN and decision trees. Often, a problem with some of these techniques is that they could overfit the data, leading to biased models. To overcome this, we could use random forest, which uses a large set of decision trees, with each tree intentionally and randomly created imperfectly, and then combine their outputs to produce a single decision. This makes random forest an example of an ensemble model.

Then we saw Naïve Bayes – a very popular technique for binary decision-making problems. It is based on a very naïve assumption that the presence of a particular feature in a class is

unrelated to the presence of any other feature. This is often not true. For example, think about the previous sentence. It has a structure. It has a logical flow, and a particular word is preceded by a certain word and followed by another. Naïve Bayes assumes that the order of these words is not important and the appearance of a word has nothing to do with the appearance of any other word in the sentence. Surprisingly, despite this simple and flawed assumption, the Naïve Bayes technique works really well. That independence assumption makes complex computations quite simple and feasible. And so, we find Naïve Bayes used in many commercial applications, especially information filtering (e.g., spam detection).

Finally, we dipped our toes into SVM. I say "dipped our toes" because SVM can be much more sophisticated and powerful than what we were able to afford in this chapter. The power and sophistication come from SVM's ability to use different kernels. Think about kernels as transformation functions that allow us to create nonlinear decision boundaries. There are situations where data seems hard to separate using a line or hyperplane, but we could perhaps draw a linear boundary in a higher dimension that looks like a curve in the current space. The full explanation of this process is beyond the scope of this book, but hopefully a few pointers provided at the end of this chapter will help you explore and learn more about this powerful technique.

Key Terms

- **Supervised learning:** Supervised learning algorithms use a set of examples from previous records that are labeled to make predictions about the future.
- **Gradient descent:** It is a machine learning algorithm that computes a slope down an error surface in order to find a model that provides the best fit for the given data.
- **Relative risk:** It is the ratio of the probability of choosing any outcome category other than baseline, over that of the baseline category.
- **Anomaly detection:** Anomaly detection refers to identification of data points, or observations that do not conform to the expected pattern of a given population.
- **Data overfitting:** When a model tries to create decision boundaries or curve fitting that connect or separate as many points as possible at the expense of simplicity. Such a process often leads to complex models that have very little error on the training data, but may not have an ability to generalize and do a good job on new data.
- **Training–validation–test data:** The training set consists of data points which are labeled and are to be used to learn the model. A validation set, often prepared separately from the training set, consists of observation that are used to tune the parameters of the model, for example, to test for overfitting, etc. The test set is used to evaluate the performance of the model.
- **Entropy:** It is a measure of disorder, uncertainty, or randomness. When the probability of each event in a given space happening is the same, entropy is the highest for that system. When there are imbalances in these probabilities, the absolute value of entropy goes down.
- **Information gain:** It is the decrease in entropy, and typically used to measure how much entropy (uncertainty) is reduced in a certain event, knowing the probability of another event.

- **Ensemble model:** It contains a combination of concrete and finite sets of alternative models, each perhaps with its own imperfections and biases, that are used together to produce a single decision.
- **True positive rate (TPR):** It indicates how much of what we detected as "1" was indeed "1."
- **False positive rate (FPR):** It indicates how much of what we detected as "1" was actually "0."

Conceptual Questions

1. What is supervised learning? Give two examples of data problems where you would use supervised learning.
2. Relate likelihood of a model given data, and probability of data given a model. Are these two the same? Different? How?
3. How does random forest address the issue of bias or overfitting?
4. Here are the past seven governors of the state of New Jersey based on their party affiliations (Democratic, Republican): R, D, D, D, D, R, D. Using Naïve Bayes formulation, calculate the probability of the next governor being a Republican. Show calculations.

Hands-On Problems

Problem 9.1 (Logistic regression)

The dataset crash.csv is an accident-survivors dataset portal for the USA (crash data for individual States can be searched) hosted by data.gov. The dataset, downloadable from OA 9.13, contains passengers' (not necessarily the driver's) age and the speed of the vehicle (mph) at the time of impact and the fate of the passengers (1 represents survived, 0 represents did not survive) after the crash. Now, use the logistic regression to decide if the age and speed can predict the survivability of the passengers.

Problem 9.2 (Logistic regression)

An automated answer-rating site marks each post in a community forum website as "good" or "bad" based on the quality of the post. The CSV file, which you can download from OA 9.14, contains the various types of quality as measured by the tool. Following are the type of qualities that the dataset contains:

i. num_words: number of words in the post
ii. num_characters: number of characters in the post
iii. num_misspelled: number of misspelled words
iv. bin_end_qmark: if the post ends with a question mark
v. num_interrogative: number of interrogative words in the post
vi. bin_start_small: if the answer starts with a lowercase letter ("1" means yes, otherwise no)
vii. num_sentences: number of sentences per post
viii. num_punctuations: number of punctuation symbols in the post
ix. label: the label of the post ("G" for good and "B" for bad) as determined by the tool.

Create a logistics regression model to predict the class label from the first eight attributes of the question set. Evaluate the accuracy of your model.

Problem 9.3 (Logistic regression)

In this exercise, you will use the Immunotherapy dataset, available from OA 9.15, which contains information about wart treatment results of 90 patients using immunotherapy. For each patient, the dataset has information about the patient's sex (either 1 or 0), age in years, number of warts, type, area, induration diameter, and result of treatment (a binary variable). Your objective in this exercise is to build a logistic regression model that will predict the result of treatment from the remaining features and to evaluate the accuracy of your model.

Problem 9.4 (Softmax regression)

The Iris flower dataset or Fisher's Iris dataset (built into R or downloadable from OA 9.16) is a multivariate dataset introduced by the British statistician and biologist Ronald Fisher in his 1936 paper.[4] The use of multiple measurements in taxonomic problems is an example of linear discriminant analysis. The dataset consists of 50 samples from each of three species of iris (*Iris sentosa*, *Iris virginica* and *Iris versicolor*). Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters. Based on the combination of these four features, create a prediction model using softmax regression for the species of iris flower.

Problem 9.5 (Softmax regression)

For the softmax regression challenge you will work with horseshoe crab data, available from OA 9.17. This dataset has 173 observations of female crabs, including the following characteristics:

i. Satellites: number of male partners in addition to the female's primary partner.
ii. Yes: a binary factor indicating if the female has satellites.
iii. Width: width of the female crab in centimeters.
iv. Weight: weight of the female in grams.
v. Color: a categorical value having range of 1 to 4, where 1 = light color, and 4 = dark.
vi. Spine: a categorical variable, valued between 1 and 3, indicating the goodness of spine of the female.

Use Softmax regression to predict the condition of the spine of female crabs based on the remaining features in the dataset and report the accuracy of your predictions.

Problem 9.6 (kNN)

Download weather.csv from OA 9.18. Entirely fictitious, it supposedly concerns the weather conditions that are suitable for playing some unspecified game. There are four predictor variables: outlook, temperature, humidity, and wind. The outcome is whether to play ("yes," "no," "maybe"). Use kNN (or, if you prefer, a different classification algorithm) to build a classifier that learns how various predictor variables could relate to the outcome. Report the accuracy of your model.

Problem 9.7 (kNN)

The following dataset came from Project 16P5 in Mosteller, F., & Tukey, J. W. (1977). *Data Analysis and Regression: A Second Course in Statistics*. Addison-Wesley, Reading, MA, pp. 549–551 (indicating their source as "Data used by permission of Franice van de Walle"). You can download it from OA 9.19.

The dataset represents standardized fertility measures and socio-economic indicators for each of 47 French-speaking provinces of Switzerland, circa 1888. Switzerland, in 1888, was entering a period known as the *demographic transition*; that is, its fertility was beginning to fall from the high level typical of underdeveloped countries. The dataset has observations on six variables, *each* of which is in percent, that is, in [0,100].

Use the kNN algorithm to find the provinces that have similar fertility measures.

	Fertility	Agriculture	Examination	Education	Catholic	Infant. Mortality
Courtelary	80.2	17	15	12	9.96	22.2
Delemont	83.1	45.1	6	9	84.84	22.2
Franches-Mnt	92.5	39.7	5	5	93.4	20.2
Moutier	85.8	36.5	12	7	33.77	20.3
Neuveville	76.9	43.5	17	15	5.16	20.6
Porrentruy	76.1	35.3	9	7	90.57	26.6
Broye	83.8	70.2	16	7	92.85	23.6
Glane	92.4	67.8	14	8	97.16	24.9
Gruyere	82.4	53.3	12	7	97.67	21
Sarine	82.9	45.2	16	13	91.38	24.4
Veveyse	87.1	64.5	14	6	98.61	24.5
Aigle	64.1	62	21	12	8.52	16.5
Aubonne	66.9	67.5	14	7	2.27	19.1
Avenches	68.9	60.7	19	12	4.43	22.7
Cossonay	61.7	69.3	22	5	2.82	18.7
Echallens	68.3	72.6	18	2	24.2	21.2
Grandson	71.7	34	17	8	3.3	20

Lausanne	55.7	19.4	26	28	12.11	20.2
La Vallee	54.3	15.2	31	20	2.15	10.8
Lavaux	65.1	73	19	9	2.84	20
Morges	65.5	59.8	22	10	5.23	18
Moudon	65	55.1	14	3	4.52	22.4
Nyone	56.6	50.9	22	12	15.14	16.7
Orbe	57.4	54.1	20	6	4.2	15.3
Oron	72.5	71.2	12	1	2.4	21
Payerne	74.2	58.1	14	8	5.23	23.8
Paysd'enhaut	72	63.5	6	3	2.56	18
Rolle	60.5	60.8	16	10	7.72	16.3
Vevey	58.3	26.8	25	19	18.46	20.9
Yverdon	65.4	49.5	15	8	6.1	22.5
Conthey	75.5	85.9	3	2	99.71	15.1
Entremont	69.3	84.9	7	6	99.68	19.8
Herens	77.3	89.7	5	2	100	18.3
Martigwy	70.5	78.2	12	6	98.96	19.4
Monthey	79.4	64.9	7	3	98.22	20.2
St Maurice	65	75.9	9	9	99.06	17.8
Sierre	92.2	84.6	3	3	99.46	16.3
Sion	79.3	63.1	13	13	96.83	18.1
Boudry	70.4	38.4	26	12	5.62	20.3
La Chauxdfnd	65.7	7.7	29	11	13.79	20.5
Le Locle	72.7	16.7	22	13	11.22	18.9
Neuchatel	64.4	17.6	35	32	16.92	23
Val de Ruz	77.6	37.6	15	7	4.97	20
ValdeTravers	67.6	18.7	25	7	8.65	19.5
V. De Geneve	35	1.2	37	53	42.34	18
Rive Droite	44.7	46.6	16	29	50.43	18.2
Rive Gauche	42.8	27.7	22	29	58.33	19.3

Problem 9.8 (kNN)

For this exercise you will work with the NFL 2014 Combine Performance Results data available from OA 9.20, which contains performance statistics of college football players at NFL, February 2014, in Indianapolis. The dataset includes the following attributes among others:

i. Overall Grade: lowest 4.5, highest 7.5
ii. Height: in inches
iii. Arm Length: in inches
iv. Weight: in lbs
v. 40 Yard Time: in seconds
vi. Bench Press: reps @ 225 lbs
vii. Vertical Jump: in inches
viii. Broad Jump: in inches
ix. 3 Cone Drill: in seconds
x. 20-Yard Shuttle: in seconds

Use kNN on this dataset to find the players who have similar performance statistics.

Problem 9.9 (Decision trees)

Obtain the dataset from OA 9.21, which was collected from Worthy, S. L., Jonkman, J. N., & Blinn-Pike, L. (2010). Sensation-seeking, risk-taking, and problematic financial behaviors of college students. *Journal of Family and Economic Issues*, 31(2), 161–170.

For this dataset, the researchers conducted a survey of 450 undergraduates in large introductory courses at either Mississippi State University or the University of Mississippi. There were close to 150 questions on the survey, but only four of these variables are included in this dataset. (You can consult the paper to learn how the variables beyond these four affect the analysis.) The primary interest for the researchers was factors relating to whether or not a student has ever overdrawn a checking account.

The dataset contains the following variables:

Age	Age of the student (in years)
Sex	0 = male or 1 = female
DaysDrink	Number of days drinking alcohol (in past 30days)
Overdrawn	Has student overdrawn a checking account? 0 = no or 1 = yes

Create a decision tree-based model to predict the student overdrawing from the checking account based on Age, Sex, and DaysDrink. Since DaysDrink is a numeric variable; you may have to convert it into a categorical one. One suggestion for that would be:

if (no. of days of drinking alcohol > 7) = 0
(7 >= no. of days of drinking alcohol > 14) = 1
(no. of days of drinking alcohol >= 14) = 2

Problem 9.10 (Decision trees)

Download the dataset from OA 9.22 for this exercise, which is sourced from the study by Nicholas Gueaguen (2002). The effects of a joke on tipping when it is delivered at the same time as the bill. *Journal of Applied Social Psychology*, 32(9), 1955–1963.

Can telling a joke affect whether or not a waiter in a coffee bar receives a tip from a customer?

This study investigated this question at a coffee bar at a famous resort on the west coast of France. The waiter randomly assigned coffee-ordering customers to one of three groups: when receiving the bill, one group also received a card telling a joke, another group received a card containing an advertisement for a local restaurant, and a third group received no card at all. He recorded whether or not each customer left a tip.

The dataset contains the following variables:

Card	Type of card used: Ad, Joke, or None
Tip	1 = customer left a tip, or 0 = no tip
Ad	Indicator for Ad card
Joke	Indicator for Joke card
None	Indicator for No card

Use a decision tree to determine whether the waiter will receive a tip from the customer from the predictor variables.

Problem 9.11 (Random forests)

The following exercise is based on the abalone dataset, which can be downloaded from OA 9.23.

The job is to predict the age of abalone from physical measurements. The age of abalone is determined by cutting the shell through the cone, staining it, and counting the number of rings through a microscope – a boring and time-consuming task. Other measurements, which are easier to obtain, are used to predict the age.

Following are the list of attributes that are available in the current dataset:

The original data examples had a couple of missing values, which were removed (the majority having the predicted value missing), and the ranges of the continuous values have been scaled for use with an artificial neural network (by dividing by 200).

Name	Data type	Measurement unit	Description
Sex	Nominal	—	M, F, and I (infant)
Length	Continuous	millimeters	Longest shell measurement
Diameter	Continuous	millimeters	Perpendicular to length
Height	Continuous	millimeters	With meat in shell
Whole weight	Continuous	grams	Whole abalone
Shucked weight	Continuous	grams	Weight of meat
Viscera weight	Continuous	grams	Gut weight (after bleeding)
Shell weight	Continuous	grams	After being dried
Rings	Integer	—	+1.5 gives the age in years

Use this dataset to predict the age of abalone from the given attributes.

Problem 9.12 (Random forests)

In this exercise you will work with the Blues Guitarists Hand Posture and Thumbing Style by Region and Birth Period data, which you can download from OA 9.24. This dataset has 93 entries of various blues guitarists born between 1874 and 1940. Apart from the name of the guitarists, that dataset contains the following four features:

i. Regions: 1 means East, 2 means Delta, 3 means Texas
ii. Years: 0 for those born before 1906, 1 for the rest
iii. Hand postures: 1 = Extended, 2 = Stacked, 3 = Lutiform
iv. Thumb styles: between 1 and 3, 1 = Alternating, 2 = Utility, 3 = Dead

Using random forest on this dataset, how accurately can you tell their birth year from their hand postures and thumb styles? How does it affect the evaluation when you include the region while training the model?

Problem 9.13 (Naïve Bayes)

There is a YouTube spam collection dataset available from OA 9.25. It is a public set of comments collected for spam research. It has five datasets composed by 1956 real messages extracted from five videos. These five videos are popular pop songs that were among the 10 most viewed of the collection period.

All the five datasets have the following attributes:

COMMENT_ID: Unique ID representing the comment
AUTHOR: Author ID
DATE: Date the comment is posted
CONTENT: The comment
TAG: For spam 1, otherwise 0

For this exercise use any four of these five datasets to build a spam filter and use that filter to check the accuracy on the remaining dataset.

Problem 9.14 (Naïve Bayes)

The dataset for the following exercise on statistics of members and non-members of the Nazi party for teachers by religion, cohort, residence, and gender is sourced from Jarausch, K. H., & Arminger, G. (1989). The German teaching profession and Nazi party membership: A demographic logit model. *Journal of Interdisciplinary History*, 20(2), 197–225. It can be downloaded from OA 9.26. Following are the attribute values and their meaning:

i. Religion: 1 = Protestant, 2 = Catholic, 3 = None
ii. Cohort: 1 = Empire, 2 = Late Empire, 3 = Early Weimar, 4 = Late Weimar, 5 = Third Reich
iii. Residence: 1 = Rural, 2 = Urban
iv. Gender: 1 = Male, 2 = Female
v. Membership: 1 = Yes, 0 = No

vi. Count: in category

Use the Naïve Bayes algorithm on this dataset to determine the likelihood of the teachers being a member of the Nazi party from their religion, cohort, residence, and gender.

Problem 9.15 (SVM)

For this exercise, we have collected a sample of 198 cases from the NIST's AnthroKids dataset that is available for download from OA 9.27. The dataset comes from a 1977 anthropometric study of body measurements for children: Foster, T. A., Voors, A. W., Webber, L. S., Frerichs, R. R., & Berenson, G. S. (1977). Anthropometric and maturation measurements of children, ages 5 to 14 years, in a biracial community – the Bogalusa Heart Study. *American Journal of Clinical Nutrition*, 30(4), 582–591. Subjects in this sample are between the ages of 8 and 18 years old, selected at random from the much larger dataset of the original study.

Use the SVM to see if we can use Height, Weight, Age, and Sex (0 = male or 1 = female) to determine the Race (0 = white or 1 = other) of the child.

Problem 9.16 (SVM)

In this exercise you will use the Portuguese sea battles data from OA 9.28 that contains the outcomes of naval battles between Portuguese and Dutch/British ships between 1583 and 1663. The dataset has the following features:

i. Battle: Name of the battle place
ii. Year: Year of the battle
iii. Portuguese ships: Number of Portuguese ships
iv. Dutch ships: Number of Dutch ships
v. English ships: Number of ships from English side
vi. Ratio of Portuguese to Dutch/British ships
vii. Spanish Involvement: 1 = Yes, 0 = No
viii. Portuguese outcome: −1 = Defeat, 0 = Draw, +1 = Victory

Use an SVM based model to predict the Portuguese outcome of the battle from the number of ships involved in all sides and Spanish involvement.

Further Reading and Resources

If you are interested in learning more about supervised learning or any of the topics discussed above, following are a few links that might be useful:

1. Advanced regression models http://r-statistics.co/adv-regression-models.html
2. Further topics on logistic regression https://onlinecourses.science.psu.edu/stat504/node/217/
3. Common pitfalls in statistical analysis: logistic regression https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5543767/

4. Decision trees in machine learning, simplified https://blogs.oracle.com/bigdata/decision-trees-machine-learning
5. A practical explanation of a Naïve Bayes classifier https://monkeylearn.com/blog/practical-explanation-naive-bayes-classifier/
6. Softmax regression http://deeplearning.stanford.edu/tutorial/supervised/SoftmaxRegression/
7. 6 Easy steps to learn the Naïve Bayes algorithm https://www.analyticsvidhya.com/blog/2015/09/naive-bayes-explained/

Notes

1. Linearly separable data source: https://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html
2. Introduction to support vector machines: http://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html
3. SVM tutorial: Support vector regression with R: https://www.svm-tutorial.com/2014/10/support-vector-regression-r/
4. Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2), 179–188. doi:10.1111/j.1469-1809.1936.tb02137.x