3

Lazy Learning – Classification Using Nearest Neighbors

An interesting new type of dining experience has been appearing in cities around the world. Patrons are served in a completely darkened restaurant by waiters who move carefully around memorized routes using only their sense of touch and sound. The allure of these establishments is the belief that depriving oneself of visual sensory input will enhance the sense of taste and smell, and foods will be experienced in new ways. Each bite provides a sense of wonder while discovering the flavors the chef has prepared.

Can you imagine how a diner experiences the unseen food? Upon first bite, the senses are overwhelmed. What are the dominant flavors? Does the food taste savory or sweet? Does it taste similar to something eaten previously? Personally, I imagine this process of discovery in terms of a slightly modified adage: if it smells like a duck and tastes like a duck, then you are probably eating duck.

This illustrates an idea that can be used for machine learning—as does another maxim involving poultry: "birds of a feather flock together." Stated differently, things that are alike are likely to have properties that are alike. Machine learning uses this principle to classify data by placing it in the same category as similar or "nearest" neighbors. This chapter is devoted to the classifiers that use this approach. You will learn:

- The key concepts that define **nearest neighbor** classifiers, and why they are considered "lazy" learners
- Methods to measure the similarity of two examples using distance
- To apply a popular nearest neighbor classifier called k-NN

If all these talks about food is making you hungry, feel free to grab a snack. Our first task will be to understand the k-NN approach by putting it to use by settling a long-running culinary debate.

Understanding nearest neighbor classification

In a single sentence, **nearest neighbor** classifiers are defined by their characteristic of classifying unlabeled examples by assigning them the class of similar labeled examples. Despite the simplicity of this idea, nearest neighbor methods are extremely powerful. They have been used successfully for:

- Computer vision applications, including optical character recognition and facial recognition in both still images and video
- Predicting whether a person will enjoy a movie or music recommendation
- Identifying patterns in genetic data, perhaps to use them in detecting specific proteins or diseases

In general, nearest neighbor classifiers are well-suited for classification tasks, where relationships among the features and the target classes are numerous, complicated, or extremely difficult to understand, yet the items of similar class type tend to be fairly homogeneous. Another way of putting it would be to say that if a concept is difficult to define, but you know it when you see it, then nearest neighbors might be appropriate. On the other hand, if the data is noisy and thus no clear distinction exists among the groups, the nearest neighbor algorithms may struggle to identify the class boundaries.

The k-NN algorithm

The nearest neighbors approach to classification is exemplified by the **k-nearest neighbors algorithm** (**k-NN**). Although this is perhaps one of the simplest machine learning algorithms, it is still used widely.

The strengths and weaknesses of this algorithm are as follows:

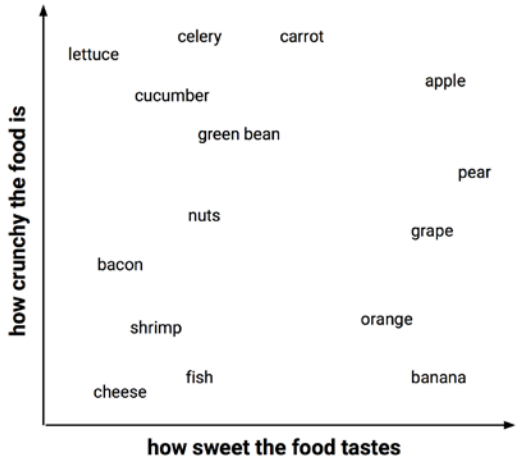
Strengths	Weaknesses
• Simple and effective • Makes no assumptions about the underlying data distribution • Fast training phase	• Does not produce a model, limiting the ability to understand how the features are related to the class • Requires selection of an appropriate *k* • Slow classification phase • Nominal features and missing data require additional processing

The k-NN algorithm gets its name from the fact that it uses information about an example's k-nearest neighbors to classify unlabeled examples. The letter *k* is a variable term implying that any number of nearest neighbors could be used. After choosing *k*, the algorithm requires a training dataset made up of examples that have been classified into several categories, as labeled by a nominal variable. Then, for each unlabeled record in the test dataset, k-NN identifies *k* records in the training data that are the "nearest" in similarity. The unlabeled test instance is assigned the class of the majority of the k nearest neighbors.

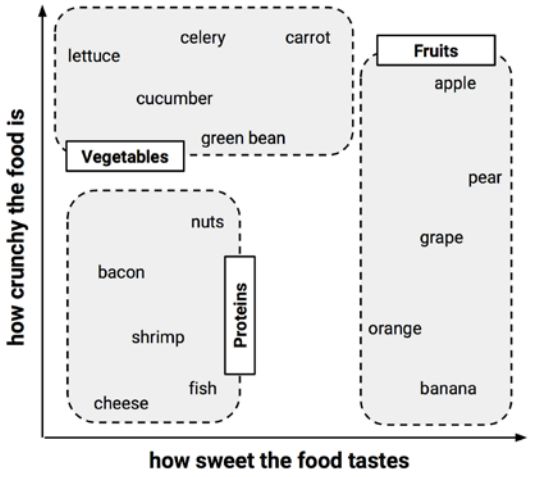
To illustrate this process, let's revisit the blind tasting experience described in the introduction. Suppose that prior to eating the mystery meal we had created a dataset in which we recorded our impressions of a number of ingredients we tasted previously. To keep things simple, we rated only two features of each ingredient. The first is a measure from 1 to 10 of how crunchy the ingredient is and the second is a 1 to 10 score of how sweet the ingredient tastes. We then labeled each ingredient as one of the three types of food: fruits, vegetables, or proteins. The first few rows of such a dataset might be structured as follows:

Ingredient	Sweetness	Crunchiness	Food type
apple	10	9	fruit
bacon	1	4	protein
banana	10	1	fruit
carrot	7	10	vegetable
celery	3	10	vegetable
cheese	1	1	protein

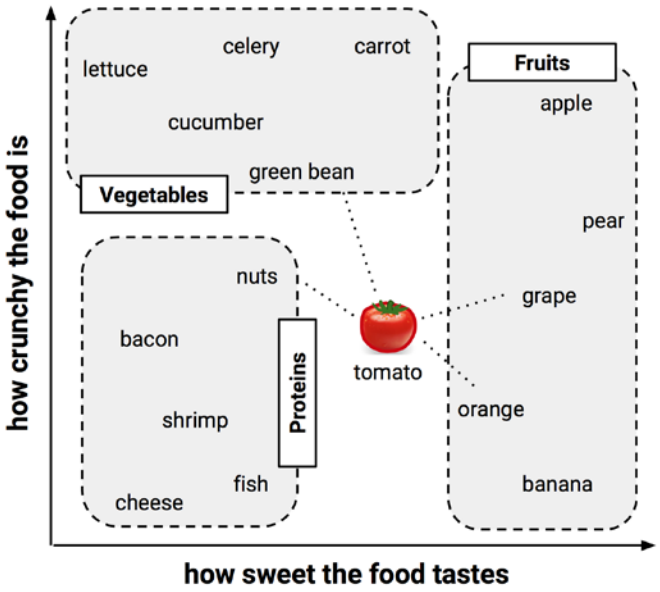
The k-NN algorithm treats the features as coordinates in a multidimensional feature space. As our dataset includes only two features, the feature space is two-dimensional. We can plot two-dimensional data on a scatter plot, with the *x* dimension indicating the ingredient's sweetness and the *y* dimension, the crunchiness. After adding a few more ingredients to the taste dataset, the scatter plot might look similar to this:



Did you notice the pattern? Similar types of food tend to be grouped closely together. As illustrated in the next diagram, vegetables tend to be crunchy but not sweet, fruits tend to be sweet and either crunchy or not crunchy, while proteins tend to be neither crunchy nor sweet:



Suppose that after constructing this dataset, we decide to use it to settle the age-old question: is tomato a fruit or vegetable? We can use the nearest neighbor approach to determine which class is a better fit, as shown in the following diagram:



Measuring similarity with distance

Locating the tomato's nearest neighbors requires a **distance function**, or a formula that measures the similarity between the two instances.

There are many different ways to calculate distance. Traditionally, the k-NN algorithm uses **Euclidean distance**, which is the distance one would measure if it were possible to use a ruler to connect two points, illustrated in the previous figure by the dotted lines connecting the tomato to its neighbors.

Euclidean distance is measured "as the crow flies," implying the shortest direct route. Another common distance measure is Manhattan distance, which is based on the paths a pedestrian would take by walking city blocks. If you are interested in learning more about other distance measures, you can read the documentation for R's distance function (a useful tool in its own right), using the `?dist` command.

Euclidean distance is specified by the following formula, where *p* and *q* are the examples to be compared, each having *n* features. The term p_1 refers to the value of the first feature of example *p*, while q_1 refers to the value of the first feature of example *q*:

$$\text{dist}(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2}$$

The distance formula involves comparing the values of each feature. For example, to calculate the distance between the tomato (*sweetness* = *6*, *crunchiness* = *4*), and the green bean (*sweetness* = *3*, *crunchiness* = *7*), we can use the formula as follows:

$$\text{dist}(\text{tomato, green bean}) = \sqrt{(6 - 3)^2 + (4 - 7)^2} = 4.2$$

In a similar vein, we can calculate the distance between the tomato and several of its closest neighbors as follows:

Ingredient	Sweetness	Crunchiness	Food type	Distance to the tomato
grape	8	5	fruit	*sqrt((6 - 8)^2 + (4 - 5)^2) = 2.2*
green bean	3	7	vegetable	*sqrt((6 - 3)^2 + (4 - 7)^2) = 4.2*
nuts	3	6	protein	*sqrt((6 - 3)^2 + (4 - 6)^2) = 3.6*
orange	7	3	fruit	*sqrt((6 - 7)^2 + (4 - 3)^2) = 1.4*

To classify the tomato as a vegetable, protein, or fruit, we'll begin by assigning the tomato, the food type of its single nearest neighbor. This is called 1-NN classification because *k = 1*. The orange is the nearest neighbor to the tomato, with a distance of 1.4. As orange is a fruit, the 1-NN algorithm would classify tomato as a fruit.

If we use the k-NN algorithm with *k = 3* instead, it performs a vote among the three nearest neighbors: orange, grape, and nuts. Since the majority class among these neighbors is fruit (two of the three votes), the tomato again is classified as a fruit.

Choosing an appropriate k

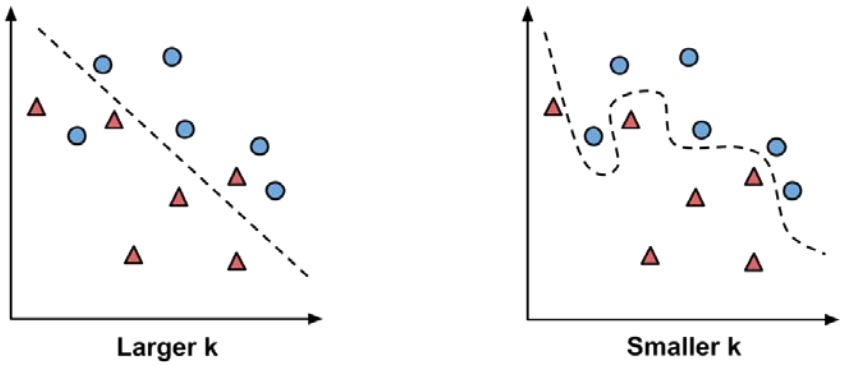
The decision of how many neighbors to use for k-NN determines how well the model will generalize to future data. The balance between overfitting and underfitting the training data is a problem known as **bias-variance tradeoff**. Choosing a large *k* reduces the impact or variance caused by noisy data, but can bias the learner so that it runs the risk of ignoring small, but important patterns.

Suppose we took the extreme stance of setting a very large *k*, as large as the total number of observations in the training data. With every training instance represented in the final vote, the most common class always has a majority of the voters. The model would consequently always predict the majority class, regardless of the nearest neighbors.

On the opposite extreme, using a single nearest neighbor allows the noisy data or outliers to unduly influence the classification of examples. For example, suppose some of the training examples were accidentally mislabeled. Any unlabeled example that happens to be nearest to the incorrectly labeled neighbor will be predicted to have the incorrect class, even if nine other nearest neighbors would have voted differently.

Obviously, the best *k* value is somewhere between these two extremes.

The following figure illustrates, more generally, how the decision boundary (depicted by a dashed line) is affected by larger or smaller *k* values. Smaller values allow more complex decision boundaries that more carefully fit the training data. The problem is that we do not know whether the straight boundary or the curved boundary better represents the true underlying concept to be learned.



In practice, choosing *k* depends on the difficulty of the concept to be learned, and the number of records in the training data. One common practice is to begin with *k* equal to the square root of the number of training examples. In the food classifier we developed previously, we might set *k* = 4 because there were 15 example ingredients in the training data and the square root of 15 is 3.87.

However, such rules may not always result in the single best *k*. An alternative approach is to test several *k* values on a variety of test datasets and choose the one that delivers the best classification performance. That said, unless the data is very noisy, a large training dataset can make the choice of *k* less important. This is because even subtle concepts will have a sufficiently large pool of examples to vote as nearest neighbors.

A less common, but interesting solution to this problem is to choose a larger *k*, but apply a **weighted voting** process in which the vote of the closer neighbors is considered more authoritative than the vote of the far away neighbors. Many k-NN implementations offer this option.

Preparing data for use with k-NN

Features are typically transformed to a standard range prior to applying the k-NN algorithm. The rationale for this step is that the distance formula is highly dependent on how features are measured. In particular, if certain features have a much larger range of values than the others, the distance measurements will be strongly dominated by the features with larger ranges. This wasn't a problem for food tasting example as both sweetness and crunchiness were measured on a scale from 1 to 10.

However, suppose we added an additional feature to the dataset for a food's spiciness, which was measured using the Scoville scale. If you are not familiar with this metric, it is a standardized measure of spice heat, ranging from zero (not at all spicy) to over a million (for the hottest chili peppers). Since the difference between spicy and non-spicy foods can be over a million, while the difference between sweet and non-sweet or crunchy and non-crunchy foods is at most 10, the difference in scale allows the spice level to impact the distance function much more than the other two factors. Without adjusting our data, we might find that our distance measures only differentiate foods by their spiciness; the impact of crunchiness and sweetness would be dwarfed by the contribution of spiciness.

The solution is to rescale the features by shrinking or expanding their range such that each one contributes relatively equally to the distance formula. For example, if sweetness and crunchiness are both measured on a scale from 1 to 10, we would also like spiciness to be measured on a scale from 1 to 10. There are several common ways to accomplish such scaling.

The traditional method of rescaling features for k-NN is **min-max normalization**. This process transforms a feature such that all of its values fall in a range between 0 and 1. The formula for normalizing a feature is as follows:

$$X_{new} = \frac{X - \min(X)}{\max(X) - \min(X)}$$

Essentially, the formula subtracts the minimum of feature *X* from each value and divides by the range of *X*.

Normalized feature values can be interpreted as indicating how far, from 0 percent to 100 percent, the original value fell along the range between the original minimum and maximum.

Another common transformation is called **z-score standardization**. The following formula subtracts the mean value of feature *X*, and divides the outcome by the standard deviation of *X*:

$$X_{new} = \frac{X - \mu}{\sigma} = \frac{X - \text{Mean}(X)}{\text{StdDev}(X)}$$

This formula, which is based on the properties of the normal distribution covered in *Chapter 2*, *Managing and Understanding Data*, rescales each of the feature's values in terms of how many standard deviations they fall above or below the mean value. The resulting value is called a **z-score**. The z-scores fall in an unbound range of negative and positive numbers. Unlike the normalized values, they have no predefined minimum and maximum.

The same rescaling method used on the k-NN training dataset must also be applied to the examples the algorithm will later classify. This can lead to a tricky situation for min-max normalization, as the minimum or maximum of future cases might be outside the range of values observed in the training data. If you know the plausible minimum or maximum value ahead of time, you can use these constants rather than the observed values. Alternatively, you can use z-score standardization under the assumption that the future examples will have similar mean and standard deviation as the training examples.

The Euclidean distance formula is not defined for nominal data. Therefore, to calculate the distance between nominal features, we need to convert them into a numeric format. A typical solution utilizes **dummy coding**, where a value of *1* indicates one category, and *0*, the other. For instance, dummy coding for a gender variable could be constructed as:

$$\text{male} = \begin{cases} 1 & \text{if x} = \text{male} \\ 0 & \text{otherwise} \end{cases}$$

Notice how the dummy coding of the two-category (binary) gender variable results in a single new feature named male. There is no need to construct a separate feature for female; since the two sexes are mutually exclusive, knowing one or the other is enough.

This is true more generally as well. An *n*-category nominal feature can be dummy coded by creating the binary indicator variables for (*n - 1*) levels of the feature. For example, the dummy coding for a three-category temperature variable (for example, hot, medium, or cold) could be set up as *(3 - 1) = 2* features, as shown here:

$$\text{hot} = \begin{cases} 1 & \text{if x} = \text{hot} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{medium} = \begin{cases} 1 & \text{if x} = \text{medium} \\ 0 & \text{otherwise} \end{cases}$$

Knowing that hot and medium are both *0* is enough to know that the temperature is cold. We, therefore, do not need a third feature for the cold category.

A convenient aspect of dummy coding is that the distance between dummy coded features is always one or zero, and thus, the values fall on the same scale as min-max normalized numeric data. No additional transformation is necessary.

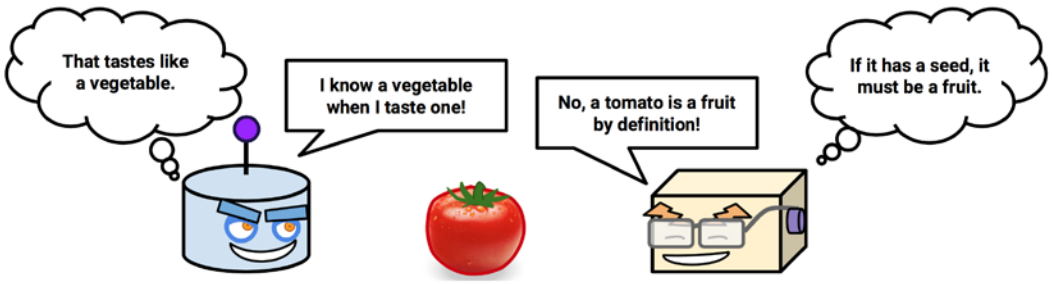
If a nominal feature is ordinal (one could make such an argument for temperature), an alternative to dummy coding is to number the categories and apply normalization. For instance, cold, warm, and hot could be numbered as 1, 2, and 3, which normalizes to 0, 0.5, and 1. A caveat to this approach is that it should only be used if the steps between the categories are equivalent. For instance, although income categories for poor, middle class, and wealthy are ordered, the difference between the poor and middle class may be different than the difference between the middle class and wealthy. Since the steps between groups are not equal, dummy coding is a safer approach.

Why is the k-NN algorithm lazy?

Classification algorithms based on the nearest neighbor methods are considered **lazy learning** algorithms because, technically speaking, no abstraction occurs. The abstraction and generalization processes are skipped altogether, and this undermines the definition of learning, proposed in *Chapter 1*, *Introducing Machine Learning*.

Under the strict definition of learning, a lazy learner is not really learning anything. Instead, it merely stores the training data verbatim. This allows the training phase, which is not actually training anything, to occur very rapidly. Of course, the downside is that the process of making predictions tends to be relatively slow in comparison to training. Due to the heavy reliance on the training instances rather than an abstracted model, lazy learning is also known as **instance-based learning** or **rote learning**.

As instance-based learners do not build a model, the method is said to be in a class of **non-parametric** learning methods—no parameters are learned about the data. Without generating theories about the underlying data, non-parametric methods limit our ability to understand how the classifier is using the data. On the other hand, this allows the learner to find natural patterns rather than trying to fit the data into a preconceived and potentially biased functional form.



Although k-NN classifiers may be considered lazy, they are still quite powerful. As you will soon see, the simple principles of nearest neighbor learning can be used to automate the process of screening for cancer.

Example – diagnosing breast cancer with the k-NN algorithm

Routine breast cancer screening allows the disease to be diagnosed and treated prior to it causing noticeable symptoms. The process of early detection involves examining the breast tissue for abnormal lumps or masses. If a lump is found, a fine-needle aspiration biopsy is performed, which uses a hollow needle to extract a small sample of cells from the mass. A clinician then examines the cells under a microscope to determine whether the mass is likely to be malignant or benign.

If machine learning could automate the identification of cancerous cells, it would provide considerable benefit to the health system. Automated processes are likely to improve the efficiency of the detection process, allowing physicians to spend less time diagnosing and more time treating the disease. An automated screening system might also provide greater detection accuracy by removing the inherently subjective human component from the process.

We will investigate the utility of machine learning for detecting cancer by applying the k-NN algorithm to measurements of biopsied cells from women with abnormal breast masses.

Step 1 – collecting data

We will utilize the Wisconsin Breast Cancer Diagnostic dataset from the UCI Machine Learning Repository at `http://archive.ics.uci.edu/ml`. This data was donated by researchers of the University of Wisconsin and includes the measurements from digitized images of fine-needle aspirate of a breast mass. The values represent the characteristics of the cell nuclei present in the digital image.

To read more about this dataset, refer to: Mangasarian OL, Street WN, Wolberg WH. Breast cancer diagnosis and prognosis via linear programming. *Operations Research*. 1995; 43:570-577.

The breast cancer data includes 569 examples of cancer biopsies, each with 32 features. One feature is an identification number, another is the cancer diagnosis, and 30 are numeric-valued laboratory measurements. The diagnosis is coded as `"M"` to indicate malignant or `"B"` to indicate benign.

The other 30 numeric measurements comprise the mean, standard error, and worst (that is, largest) value for 10 different characteristics of the digitized cell nuclei. These include:

- Radius
- Texture
- Perimeter
- Area
- Smoothness
- Compactness
- Concavity
- Concave points
- Symmetry
- Fractal dimension

Based on these names, all the features seem to relate to the shape and size of the cell nuclei. Unless you are an oncologist, you are unlikely to know how each relates to benign or malignant masses. These patterns will be revealed as we continue in the machine learning process.

Step 2 – exploring and preparing the data

Let's explore the data and see whether we can shine some light on the relationships. In doing so, we will prepare the data for use with the k-NN learning method.

If you plan on following along, download the `wisc_bc_data.csv` file from the Packt website and save it to your R working directory. The dataset was modified very slightly from its original form for this book. In particular, a header line was added and the rows of data were randomly ordered.

We'll begin by importing the CSV data file, as we have done in previous chapters, saving the Wisconsin breast cancer data to the `wbcd` data frame:

```
> wbcd <- read.csv("wisc_bc_data.csv", stringsAsFactors = FALSE)
```

Using the `str(wbcd)` command, we can confirm that the data is structured with 569 examples and 32 features as we expected. The first several lines of output are as follows:

```
'data.frame':  569 obs. of  32 variables:
 $ id               : int  87139402 8910251 905520 ...
 $ diagnosis        : chr  "B" "B" "B" "B" ...
 $ radius_mean      : num  12.3 10.6 11 11.3 15.2 ...
 $ texture_mean     : num  12.4 18.9 16.8 13.4 13.2 ...
 $ perimeter_mean   : num  78.8 69.3 70.9 73 97.7 ...
 $ area_mean        : num  464 346 373 385 712 ...
```

The first variable is an integer variable named `id`. As this is simply a unique identifier (ID) for each patient in the data, it does not provide useful information, and we will need to exclude it from the model.

Regardless of the machine learning method, ID variables should always be excluded. Neglecting to do so can lead to erroneous findings because the ID can be used to uniquely "predict" each example. Therefore, a model that includes an identifier will suffer from overfitting, and is unlikely to generalize well to other data.

Let's drop the `id` feature altogether. As it is located in the first column, we can exclude it by making a copy of the `wbcd` data frame without column `1`:

```
> wbcd <- wbcd[-1]
```

The next variable, `diagnosis`, is of particular interest as it is the outcome we hope to predict. This feature indicates whether the example is from a benign or malignant mass. The `table()` output indicates that 357 masses are benign while 212 are malignant:

```
> table(wbcd$diagnosis)

  B   M
357 212
```

Many R machine learning classifiers require that the target feature is coded as a factor, so we will need to recode the `diagnosis` variable. We will also take this opportunity to give the `"B"` and `"M"` values more informative labels using the `labels` parameter:

```
> wbcd$diagnosis<- factor(wbcd$diagnosis, levels = c("B", "M"),
        labels = c("Benign", "Malignant"))
```

Now, when we look at the `prop.table()` output, we notice that the values have been labeled `Benign` and `Malignant` with 62.7 percent and 37.3 percent of the masses, respectively:

```
> round(prop.table(table(wbcd$diagnosis)) * 100, digits = 1)

   Benign Malignant
     62.7      37.3
```

The remaining 30 features are all numeric, and as expected, they consist of three different measurements of ten characteristics. For illustrative purposes, we will only take a closer look at three of these features:

```
> summary(wbcd[c("radius_mean", "area_mean", "smoothness_mean")])

  radius_mean       area_mean      smoothness_mean
 Min.   : 6.981   Min.   : 143.5   Min.   :0.05263
 1st Qu.:11.700   1st Qu.: 420.3   1st Qu.:0.08637
 Median :13.370   Median : 551.1   Median :0.09587
 Mean   :14.127   Mean   : 654.9   Mean   :0.09636
 3rd Qu.:15.780   3rd Qu.: 782.7   3rd Qu.:0.10530
 Max.   :28.110   Max.   :2501.0   Max.   :0.16340
```

Looking at the features side-by-side, do you notice anything problematic about the values? Recall that the distance calculation for k-NN is heavily dependent upon the measurement scale of the input features. Since smoothness ranges from 0.05 to 0.16 and area ranges from `143.5` to `2501.0`, the impact of area is going to be much larger than the smoothness in the distance calculation. This could potentially cause problems for our classifier, so let's apply normalization to rescale the features to a standard range of values.

Transformation – normalizing numeric data

To normalize these features, we need to create a `normalize()` function in R. This function takes a vector `x` of numeric values, and for each value in `x`, subtracts the minimum value in `x` and divides by the range of values in `x`. Finally, the resulting vector is returned. The code for this function is as follows:

```
> normalize <- function(x) {
        return ((x - min(x)) / (max(x) - min(x)))
}
```

After executing the preceding code, the `normalize()` function is available for use in R. Let's test the function on a couple of vectors:

```
> normalize(c(1, 2, 3, 4, 5))
[1] 0.00 0.25 0.50 0.75 1.00
> normalize(c(10, 20, 30, 40, 50))
[1] 0.00 0.25 0.50 0.75 1.00
```

The function appears to be working correctly. Despite the fact that the values in the second vector are 10 times larger than the first vector, after normalization, they both appear exactly the same.

We can now apply the `normalize()` function to the numeric features in our data frame. Rather than normalizing each of the 30 numeric variables individually, we will use one of R's functions to automate the process.

The `lapply()` function takes a list and applies a specified function to each list element. As a data frame is a list of equal-length vectors, we can use `lapply()` to apply `normalize()` to each feature in the data frame. The final step is to convert the list returned by `lapply()` to a data frame, using the `as.data.frame()` function. The full process looks like this:

```
> wbcd_n <- as.data.frame(lapply(wbcd[2:31], normalize))
```

In plain English, this command applies the `normalize()` function to columns 2 through 31 in the `wbcd` data frame, converts the resulting list to a data frame, and assigns it the name `wbcd_n`. The `_n` suffix is used here as a reminder that the values in `wbcd` have been normalized.

To confirm that the transformation was applied correctly, let's look at one variable's summary statistics:

```
> summary(wbcd_n$area_mean)
Min. 1st Qu.Median    Mean 3rd Qu.    Max.
0.0000  0.1174  0.1729  0.2169  0.2711  1.0000
```

As expected, the `area_mean` variable, which originally ranged from 143.5 to 2501.0, now ranges from 0 to 1.

Data preparation – creating training and test datasets

Although all the 569 biopsies are labeled with a benign or malignant status, it is not very interesting to predict what we already know. Additionally, any performance measures we obtain during the training may be misleading as we do not know the extent to which cases have been overfitted or how well the learner will generalize to unseen cases. A more interesting question is how well our learner performs on a dataset of unlabeled data. If we had access to a laboratory, we could apply our learner to the measurements taken from the next 100 masses of unknown cancer status, and see how well the machine learner's predictions compare to the diagnoses obtained using conventional methods.

In the absence of such data, we can simulate this scenario by dividing our data into two portions: a training dataset that will be used to build the k-NN model and a test dataset that will be used to estimate the predictive accuracy of the model. We will use the first 469 records for the training dataset and the remaining 100 to simulate new patients.

Using the data extraction methods given in *Chapter 2*, *Managing and Understanding Data*, we will split the `wbcd_n` data frame into `wbcd_train` and `wbcd_test`:

```
> wbcd_train <- wbcd_n[1:469, ]
> wbcd_test <- wbcd_n[470:569, ]
```

If the preceding commands are confusing, remember that data is extracted from data frames using the `[row, column]` syntax. A blank value for the row or column value indicates that all the rows or columns should be included. Hence, the first line of code takes rows 1 to 469 and all columns, and the second line takes 100 rows from 470 to 569 and all columns.

When constructing training and test datasets, it is important that each dataset is a representative subset of the full set of data. The `wbcd` records were already randomly ordered, so we could simply extract 100 consecutive records to create a test dataset. This would not be appropriate if the data was ordered chronologically or in groups of similar values. In these cases, random sampling methods would be needed. Random sampling will be discussed in *Chapter 5*, *Divide and Conquer – Classification Using Decision Trees and Rules*.

When we constructed our normalized training and test datasets, we excluded the target variable, `diagnosis`. For training the k-NN model, we will need to store these class labels in factor vectors, split between the training and test datasets:

```
> wbcd_train_labels <- wbcd[1:469, 1]
> wbcd_test_labels <- wbcd[470:569, 1]
```

This code takes the `diagnosis` factor in the first column of the `wbcd` data frame, and creates the vectors `wbcd_train_labels` and `wbcd_test_labels`. We will use these in the next steps of training and evaluating our classifier.

Step 3 – training a model on the data

Equipped with our training data and labels vector, we are now ready to classify our unknown records. For the k-NN algorithm, the training phase actually involves no model building; the process of training a lazy learner like k-NN simply involves storing the input data in a structured format.

To classify our test instances, we will use a k-NN implementation from the `class` package, which provides a set of basic R functions for classification. If this package is not already installed on your system, you can install it by typing:

```
> install.packages("class")
```

To load the package during any session in which you wish to use the functions, simply enter the `library(class)` command.

The `knn()` function in the `class` package provides a standard, classic implementation of the k-NN algorithm. For each instance in the test data, the function will identify the k-Nearest Neighbors, using Euclidean distance, where *k* is a user-specified number. The test instance is classified by taking a "vote" among the k-Nearest Neighbors—specifically, this involves assigning the class of the majority of the *k* neighbors. A tie vote is broken at random.

There are several other k-NN functions in other R packages, which provide more sophisticated or more efficient implementations. If you run into limits with `knn()`, search for k-NN at the **Comprehensive R Archive Network** (**CRAN**).

Training and classification using the `knn()` function is performed in a single function call, using four parameters, as shown in the following table:

kNN classification syntax

using the `knn()` function in the `class` package

Building the classifier and making predictions:

```
p <- knn(train, test, class, k)
```

- `train` is a data frame containing numeric training data
- `test` is a data frame containing numeric test data
- `class` is a factor vector with the class for each row in the training data
- `k` is an integer indicating the number of nearest neighbors

The function returns a factor vector of predicted classes for each row in the test data frame.

Example:

```
wbcd_pred <- knn(train = wbcd_train, test = wbcd_test,
                 cl = wbcd_train_labels, k = 3)
```

We now have nearly everything that we need to apply the k-NN algorithm to this data. We've split our data into training and test datasets, each with exactly the same numeric features. The labels for the training data are stored in a separate factor vector. The only remaining parameter is `k`, which specifies the number of neighbors to include in the vote.

As our training data includes 469 instances, we might try `k = 21`, an odd number roughly equal to the square root of 469. With a two-category outcome, using an odd number eliminates the chance of ending with a tie vote.

Now we can use the `knn()` function to classify the `test` data:

```
> wbcd_test_pred <- knn(train = wbcd_train, test = wbcd_test,
                        cl = wbcd_train_labels, k = 21)
```

The `knn()` function returns a factor vector of predicted labels for each of the examples in the `test` dataset, which we have assigned to `wbcd_test_pred`.

Step 4 – evaluating model performance

The next step of the process is to evaluate how well the predicted classes in the `wbcd_test_pred` vector match up with the known values in the `wbcd_test_labels` vector. To do this, we can use the `CrossTable()` function in the `gmodels` package, which was introduced in *Chapter 2*, *Managing and Understanding Data*. If you haven't done so already, please install this package, using the `install.packages("gmodels")` command.

After loading the package with the `library(gmodels)` command, we can create a cross tabulation indicating the agreement between the two vectors. Specifying `prop.chisq = FALSE` will remove the unnecessary chi-square values from the output:

```
> CrossTable(x = wbcd_test_labels, y = wbcd_test_pred,
                       prop.chisq=FALSE)
```

The resulting table looks like this:

```
                   | wbcd_test_pred
wbcd_test_labels |    Benign | Malignant | Row Total |
-----------------|-----------|-----------|-----------|
          Benign |        61 |         0 |        61 |
                 |     1.000 |     0.000 |     0.610 |
                 |     0.968 |     0.000 |           |
                 |     0.610 |     0.000 |           |
-----------------|-----------|-----------|-----------|
       Malignant |         2 |        37 |        39 |
                 |     0.051 |     0.949 |     0.390 |
                 |     0.032 |     1.000 |           |
                 |     0.020 |     0.370 |           |
-----------------|-----------|-----------|-----------|
    Column Total |        63 |        37 |       100 |
                 |     0.630 |     0.370 |           |
-----------------|-----------|-----------|-----------|
```

The cell percentages in the table indicate the proportion of values that fall into four categories. The top-left cell indicates the **true negative** results. These 61 of 100 values are cases where the mass was benign and the k-NN algorithm correctly identified it as such. The bottom-right cell indicates the **true positive** results, where the classifier and the clinically determined label agree that the mass is malignant. A total of 37 of 100 predictions were true positives.

The cells falling on the other diagonal contain counts of examples where the k-NN approach disagreed with the true label. The two examples in the lower-left cell are **false negative** results; in this case, the predicted value was benign, but the tumor was actually malignant. Errors in this direction could be extremely costly as they might lead a patient to believe that she is cancer-free, but in reality, the disease may continue to spread. The top-right cell would contain the **false positive** results, if there were any. These values occur when the model classifies a mass as malignant, but in reality, it was benign. Although such errors are less dangerous than a false negative result, they should also be avoided as they could lead to additional financial burden on the health care system or additional stress for the patient as additional tests or treatment may have to be provided.

If we desired, we could totally eliminate false negatives by classifying every mass as malignant. Obviously, this is not a realistic strategy. Still, it illustrates the fact that prediction involves striking a balance between the false positive rate and the false negative rate. In *Chapter 10*, *Evaluating Model Performance*, you will learn more sophisticated methods for measuring predictive accuracy that can be used to identify places where the error rate can be optimized depending on the costs of each type of error.

A total of 2 out of 100, or 2 percent of masses were incorrectly classified by the k-NN approach. While 98 percent accuracy seems impressive for a few lines of R code, we might try another iteration of the model to see whether we can improve the performance and reduce the number of values that have been incorrectly classified, particularly because the errors were dangerous false negatives.

Step 5 – improving model performance

We will attempt two simple variations on our previous classifier. First, we will employ an alternative method for rescaling our numeric features. Second, we will try several different values for *k*.

Transformation – z-score standardization

Although normalization is traditionally used for k-NN classification, it may not always be the most appropriate way to rescale features. Since the z-score standardized values have no predefined minimum and maximum, extreme values are not compressed towards the center. One might suspect that with a malignant tumor, we might see some very extreme outliers as the tumors grow uncontrollably. It might, therefore, be reasonable to allow the outliers to be weighted more heavily in the distance calculation. Let's see whether z-score standardization can improve our predictive accuracy.

To standardize a vector, we can use the R's built-in `scale()` function, which, by default, rescales values using the z-score standardization. The `scale()` function offers the additional benefit that it can be applied directly to a data frame, so we can avoid the use of the `lapply()` function. To create a z-score standardized version of the `wbcd` data, we can use the following command:

```
> wbcd_z <- as.data.frame(scale(wbcd[-1]))
```

This command rescales all the features, with the exception of `diagnosis` and stores the result as the `wbcd_z` data frame. The `_z` suffix is a reminder that the values were z-score transformed.

To confirm that the transformation was applied correctly, we can look at the summary statistics:

```
> summary(wbcd_z$area_mean)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-1.4530 -0.6666 -0.2949  0.0000  0.3632  5.2460
```

The mean of a z-score standardized variable should always be zero, and the range should be fairly compact. A z-score greater than 3 or less than -3 indicates an extremely rare value. With this in mind, the transformation seems to have worked.

As we had done earlier, we need to divide the data into training and test sets, and then classify the test instances using the `knn()` function. We'll then compare the predicted labels to the actual labels using `CrossTable()`:

```
> wbcd_train <- wbcd_z[1:469, ]
> wbcd_test <- wbcd_z[470:569, ]
> wbcd_train_labels <- wbcd[1:469, 1]
> wbcd_test_labels <- wbcd[470:569, 1]
```

```
> wbcd_test_pred <- knn(train = wbcd_train, test = wbcd_test,
                        cl = wbcd_train_labels, k = 21)
> CrossTable(x = wbcd_test_labels, y = wbcd_test_pred,
                       prop.chisq = FALSE)
```

Unfortunately, in the following table, the results of our new transformation show a slight decline in accuracy. The instances where we had correctly classified 98 percent of examples previously, we classified only 95 percent correctly this time. Making matters worse, we did no better at classifying the dangerous false negatives:

```
                | wbcd_test_pred
wbcd_test_labels |    Benign | Malignant | Row Total |
-----------------|-----------|-----------|-----------|
          Benign |        61 |         0 |        61 |
                 |     1.000 |     0.000 |     0.610 |
                 |     0.924 |     0.000 |           |
                 |     0.610 |     0.000 |           |
-----------------|-----------|-----------|-----------|
       Malignant |         5 |        34 |        39 |
                 |     0.128 |     0.872 |     0.390 |
                 |     0.076 |     1.000 |           |
                 |     0.050 |     0.340 |           |
-----------------|-----------|-----------|-----------|
    Column Total |        66 |        34 |       100 |
                 |     0.660 |     0.340 |           |
-----------------|-----------|-----------|-----------|
```

Testing alternative values of k

We may be able do even better by examining performance across various *k* values. Using the normalized training and test datasets, the same 100 records were classified using several different *k* values. The number of false negatives and false positives are shown for each iteration:

k value	False negatives	False positives	Percent classified incorrectly
1	1	3	4 percent
5	2	0	2 percent
11	3	0	3 percent
15	3	0	3 percent
21	2	0	2 percent
27	4	0	4 percent

Although the classifier was never perfect, the 1-NN approach was able to avoid some of the false negatives at the expense of adding false positives. It is important to keep in mind, however, that it would be unwise to tailor our approach too closely to our test data; after all, a different set of 100 patient records is likely to be somewhat different from those used to measure our performance.

If you need to be certain that a learner will generalize to future data, you might create several sets of 100 patients at random and repeatedly retest the result. The methods to carefully evaluate the performance of machine learning models will be discussed further in *Chapter 10*, *Evaluating Model Performance*.

Summary

In this chapter, we learned about classification using k-NN. Unlike many classification algorithms, k-NN does not do any learning. It simply stores the training data verbatim. Unlabeled test examples are then matched to the most similar records in the training set using a distance function, and the unlabeled example is assigned the label of its neighbors.

In spite of the fact that k-NN is a very simple algorithm, it is capable of tackling extremely complex tasks, such as the identification of cancerous masses. In a few simple lines of R code, we were able to correctly identify whether a mass was malignant or benign 98 percent of the time.

In the next chapter, we will examine a classification method that uses probability to estimate the likelihood that an observation falls into certain categories. It will be interesting to compare how this approach differs from k-NN. Later on, in *Chapter 9*, *Finding Groups of Data – Clustering with k-means*, we will learn about a close relative to k-NN, which uses distance measures for a completely different learning task.