Using 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 to occur very rapidly, with a potential downside being that the process of making predictions tends to be relatively slow. Due to the heavy reliance on the training instances, 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 form.

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

Diagnosing breast cancer with the kNN 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 utilizes a hollow needle to extract a small portion 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 kNN algorithm to measurements of biopsied cells from women with abnormal breast masses.

Step 1 – collecting data

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

To read more about the Wisconsin breast cancer data, refer to the authors' publication: *Nuclear feature extraction for breast tumor diagnosis. IS&T/SPIE 1993 International Symposium on Electronic Imaging: Science and Technology, volume 1905*, pp 861-870 by *W.N. Street*, *W.H. Wolberg*, and *O.L. Mangasarian*, 1993.

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 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 their names, all of 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 if we can shine some light on the relationships. At the same time, we will prepare the data for use with the kNN 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 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 previously, saving the Wisconsin breast cancer data to the `wbcd` data frame:

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

Using the command `str(wbcd)`, 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 most likely suffer from overfitting, and is not likely 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, consist of three different measurements of ten characteristics. For illustrative purposes, we will only take a closer look at three of the 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 kNN is heavily dependent upon the measurement scale of the input features. As `smoothness_mean` ranges from 0.05 to 0.16, while `area_mean` ranges from 143.5 to 2501.0, the impact of area is going to be much larger than 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`, subtract the minimum value in `x` and divide by the range of values in `x`. Finally, the resulting vector is returned. The code for the function is as follows:

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

After executing the previous code, the `normalize()` function is available for use. 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 of R takes a list and applies a function to each element of the list. 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 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 training may be misleading, as we do not know the extent to which cases has been overfitted, or how well it 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 measurements taken from the next 100 masses of unknown cancer status and see how well the machine learner's predictions compare to 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 kNN 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 presented in *Chapter 2*, *Managing and Understanding Data*, we will split the `wcbd_n` data frame into the `wbcd_train` and `wbcd_test` data frames:

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

If the previous code is 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 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. In the case that we just saw, the records were already sorted in a random order, so we could simply extract 100 consecutive records to create a test dataset. This would not be an appropriate method if the data was ordered in a non-random pattern such as chronologically, or in groups of similar values. In these cases, random sampling methods would be needed.

When we constructed our training and test data, we excluded the target variable, `diagnosis`. For training the kNN model, we will need to store these class labels in factor vectors, divided to 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 column 1 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 kNN algorithm, the training phase actually involves no model building—the process of training a lazy learner like kNN simply involves storing the input data in a structured format.

To classify our test instances, we will use a kNN 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 command `library(class)`.

The `knn()` function in the `class` package provides a standard, classic implementation of the kNN 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 kNN functions in other R packages, providing more sophisticated or more efficient implementations. If you run into limits with `knn()`, take a look at the **Comprehensive R Archive Network** (**CRAN**) to see what else is out there. With that said, you may be surprised how well the basic `knn()` function works out of the box.

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 already have nearly everything that we need to apply the kNN algorithm to this data. We 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. Using an odd number will reduce 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 command `install.packages("gmodels")`.

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 **chi-square** values that are not needed, from the output:

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

The resulting table looks like this:

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

Total Observations in Table:  100

                 | 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. In the top-left cell (labeled **TN**), are the true negative results. These 61 of 100 values indicate cases where the mass was benign, and the kNN algorithm correctly identified it as such. The bottom-right cell (labeled **TP**), 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 kNN approach disagreed with the true label. The 2 examples in the lower-left **FN** 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, when in reality the disease may continue to spread. The cell labeled **FP** would contain the false positive results, if there were any. These values occur when the model classifies a mass as **malignant** when 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 percent, that is, 2 out of 100 masses were incorrectly classified by the kNN approach. While 98 percent accuracy seems impressive for a few lines of R code, we might try another iteration of the model to see if we can improve the performance and reduce the number of values that have been incorrectly classified, particularly, as 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 kNN classification, it may not always be the most appropriate way to rescale features. Because 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 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 use of the `lapply()` function. To create a z-score standardized version of the `wbcd` data, we can use the following command, which rescales all features with the exception of `diagnosis`, and stores the result as a data frame in the `wbcd_z` variable. The `_z` suffix is a reminder that the values were z-score transformed.

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

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. The previous summary seems reasonable.

As we had done before, we need to divide the data into training and test sets, 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_labels	wbcd_test_pred 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 values of k. 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 1NN 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. Methods to carefully evaluate the performance of machine learning models are discussed further in *Chapter 10, Evaluating Model Performance*.