9

Finding Groups of Data – Clustering with k-means

Have you ever spent time watching a large crowd? If so, you are likely to have seen some recurring personalities. Perhaps a certain type of person, identified by a freshly pressed suit and a briefcase, comes to typify the "fat cat" business executive. A twenty-something wearing skinny jeans, a flannel shirt, and sunglasses might be dubbed a "hipster," while a woman unloading children from a minivan may be labeled a "soccer mom."

Of course, these types of stereotypes are dangerous to apply to individuals, as no two people are exactly alike. Yet understood as a way to describe a collective, the labels capture some underlying aspect of similarity among the individuals within the group.

As you will soon learn, the act of clustering, or spotting patterns in data, is not much different from spotting patterns in groups of people. In this chapter, you will learn:

- The ways clustering tasks differ from the classification tasks we examined previously
- How clustering defines a group, and how such groups are identified by k-means, a classic and easy-to-understand clustering algorithm
- The steps needed to apply clustering to a real-world task of identifying marketing segments among teenage social media users

Before jumping into action, we'll begin by taking an in-depth look at exactly what clustering entails.

Understanding clustering

Clustering is an unsupervised machine learning task that automatically divides the data into **clusters**, or groups of similar items. It does this without having been told how the groups should look ahead of time. As we may not even know what we're looking for, clustering is used for knowledge discovery rather than prediction. It provides an insight into the natural groupings found within data.

Without advance knowledge of what comprises a cluster, how can a computer possibly know where one group ends and another begins? The answer is simple. Clustering is guided by the principle that items inside a cluster should be very similar to each other, but very different from those outside. The definition of similarity might vary across applications, but the basic idea is always the same—group the data so that the related elements are placed together.

The resulting clusters can then be used for action. For instance, you might find clustering methods employed in the following applications:

- Segmenting customers into groups with similar demographics or buying patterns for targeted marketing campaigns
- Detecting anomalous behavior, such as unauthorized network intrusions, by identifying patterns of use falling outside the known clusters
- Simplifying extremely large datasets by grouping features with similar values into a smaller number of homogeneous categories

Overall, clustering is useful whenever diverse and varied data can be exemplified by a much smaller number of groups. It results in meaningful and actionable data structures that reduce complexity and provide insight into patterns of relationships.

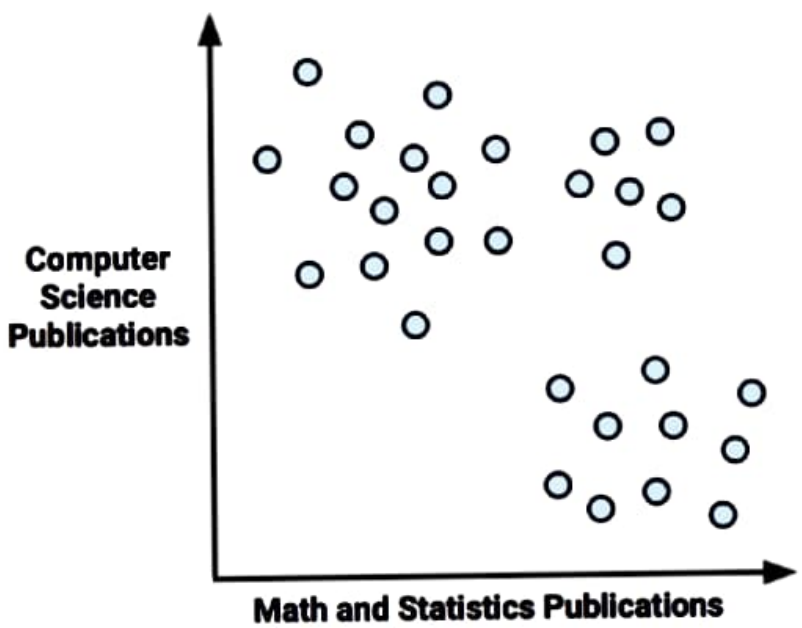
Clustering as a machine learning task

Clustering is somewhat different from the classification, numeric prediction, and pattern detection tasks we examined so far. In each of these cases, the result is a model that relates features to an outcome or features to other features; conceptually, the model describes the existing patterns within data. In contrast, clustering creates new data. Unlabeled examples are given a cluster label that has been inferred entirely from the relationships within the data. For this reason, you will, sometimes, see the clustering task referred to as **unsupervised classification** because, in a sense, it classifies unlabeled examples.

The catch is that the class labels obtained from an unsupervised classifier are without intrinsic meaning. Clustering will tell you which groups of examples are closely related—for instance, it might return the groups A, B, and C—but it's up to you to apply an actionable and meaningful label. To see how this impacts the clustering task, let's consider a hypothetical example.

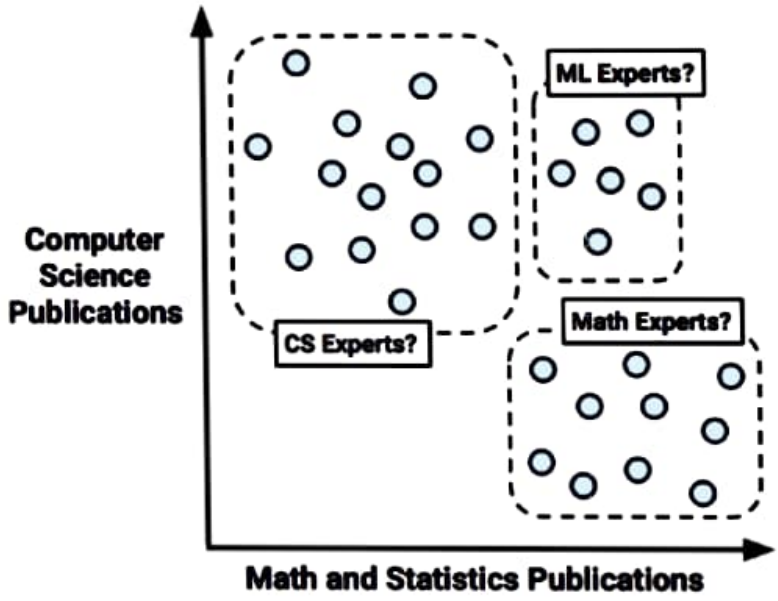
Suppose you were organizing a conference on the topic of data science. To facilitate professional networking and collaboration, you planned to seat people in groups according to one of three research specialties: computer and/or database science, math and statistics, and machine learning. Unfortunately, after sending out the conference invitations, you realize that you had forgotten to include a survey asking which discipline the attendee would prefer to be seated with.

In a stroke of brilliance, you realize that you might be able to infer each scholar's research specialty by examining his or her publication history. To this end, you begin collecting data on the number of articles each attendee published in computer science-related journals and the number of articles published in math or statistics-related journals. Using the data collected for several scholars, you create a scatterplot:



As expected, there seems to be a pattern. We might guess that the upper-left corner, which represents people with many computer science publications but few articles on math, could be a cluster of computer scientists. Following this logic, the lower-right corner might be a group of mathematicians. Similarly, the upper-right corner, those with both math and computer science experience, may be machine learning experts.

Our groupings were formed visually; we simply identified clusters as closely grouped data points. Yet in spite of the seemingly obvious groupings, we unfortunately have no way to know whether they are truly homogeneous without personally asking each scholar about his/her academic specialty. The labels we applied required us to make qualitative, presumptive judgments about the types of people that would fall into the group. For this reason, you might imagine the cluster labels in uncertain terms, as follows:



Rather than defining the group boundaries subjectively, it would be nice to use machine learning to define them objectively. Given the axis-parallel splits in the preceding diagram, our problem seems like an obvious application for the decision trees described in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*. This might provide us with a rule in the form "if a scholar has few math publications, then he/she is a computer science expert." Unfortunately, there's a problem with this plan. As we do not have data on the true class value for each point, a supervised learning algorithm would have no ability to learn such a pattern, as it would have no way of knowing what splits would result in homogenous groups.

On the other hand, clustering algorithms use a process very similar to what we did by visually inspecting the scatterplot. Using a measure of how closely the examples are related, homogeneous groups can be identified. In the next section, we'll start looking at how clustering algorithms are implemented.

This example highlights an interesting application of clustering. If you begin with unlabeled data, you can use clustering to create class labels. From there, you could apply a supervised learner such as decision trees to find the most important predictors of these classes. This is called **semi-supervised learning**.

The k-means clustering algorithm

The **k-means algorithm** is perhaps the most commonly used clustering method. Having been studied for several decades, it serves as the foundation for many more sophisticated clustering techniques. If you understand the simple principles it uses, you will have the knowledge needed to understand nearly any clustering algorithm in use today. Many such methods are listed on the following site, the **CRAN Task View** for clustering at `http://cran.r-project.org/web/views/Cluster.html`.

As k-means has evolved over time, there are many implementations of the algorithm. One popular approach is described in : Hartigan JA, Wong MA. A k-means clustering algorithm. Applied *Statistics*. 1979; 28:100-108.

Even though clustering methods have advanced since the inception of k-means, this is not to imply that k-means is obsolete. In fact, the method may be more popular now than ever. The following table lists some reasons why k-means is still used widely:

Strengths	Weaknesses
• Uses simple principles that can be explained in non-statistical terms • Highly flexible, and can be adapted with simple adjustments to address nearly all of its shortcomings • Performs well enough under many real-world use cases	• Not as sophisticated as more modern clustering algorithms • Because it uses an element of random chance, it is not guaranteed to find the optimal set of clusters • Requires a reasonable guess as to how many clusters naturally exist in the data • Not ideal for non-spherical clusters or clusters of widely varying density

If the name k-means sounds familiar to you, you may be recalling the k-NN algorithm discussed in *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*. As you will soon see, k-means shares more in common with the k-nearest neighbors than just the letter k.

The k-means algorithm assigns each of the n examples to one of the k clusters, where k is a number that has been determined ahead of time. The goal is to minimize the differences within each cluster and maximize the differences between the clusters.

Unless k and n are extremely small, it is not feasible to compute the optimal clusters across all the possible combinations of examples. Instead, the algorithm uses a heuristic process that finds **locally optimal** solutions. Put simply, this means that it starts with an initial guess for the cluster assignments, and then modifies the assignments slightly to see whether the changes improve the homogeneity within the clusters.

We will cover the process in depth shortly, but the algorithm essentially involves two phases. First, it assigns examples to an initial set of k clusters. Then, it updates the assignments by adjusting the cluster boundaries according to the examples that currently fall into the cluster. The process of updating and assigning occurs several times until changes no longer improve the cluster fit. At this point, the process stops and the clusters are finalized.

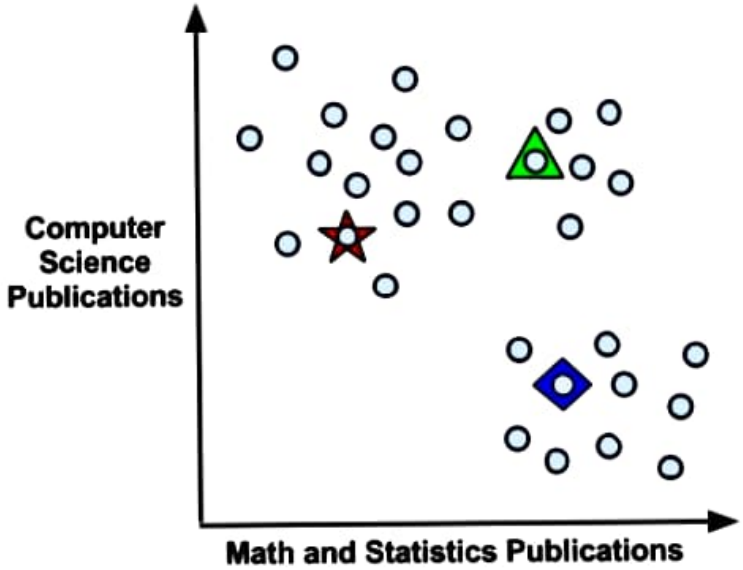
Due to the heuristic nature of k-means, you may end up with somewhat different final results by making only slight changes to the starting conditions. If the results vary dramatically, this could indicate a problem. For instance, the data may not have natural groupings or the value of k has been poorly chosen. With this in mind, it's a good idea to try a cluster analysis more than once to test the robustness of your findings.

To see how the process of assigning and updating works in practice, let's revisit the case of the hypothetical data science conference. Though this is a simple example, it will illustrate the basics of how k-means operates under the hood.

Using distance to assign and update clusters

As with k-NN, k-means treats feature values as coordinates in a multidimensional feature space. For the conference data, there are only two features, so we can represent the feature space as a two-dimensional scatterplot as depicted previously.

The k-means algorithm begins by choosing k points in the feature space to serve as the cluster centers. These centers are the catalyst that spurs the remaining examples to fall into place. Often, the points are chosen by selecting k random examples from the training dataset. As we hope to identify three clusters, according to this method, $k = 3$ points will be selected at random. These points are indicated by the star, triangle, and diamond in the following diagram:



It's worth noting that although the three cluster centers in the preceding diagram happen to be widely spaced apart, this is not always necessarily the case. Since they are selected at random, the three centers could have just as easily been three adjacent points. As the k-means algorithm is highly sensitive to the starting position of the cluster centers, this means that random chance may have a substantial impact on the final set of clusters.

To address this problem, k-means can be modified to use different methods for choosing the initial centers. For example, one variant chooses random values occurring anywhere in the feature space (rather than only selecting among the values observed in the data). Another option is to skip this step altogether; by randomly assigning each example to a cluster, the algorithm can jump ahead immediately to the update phase. Each of these approaches adds a particular bias to the final set of clusters, which you may be able to use to improve your results.

In 2007, an algorithm called **k-means++** was introduced, which proposes an alternative method for selecting the initial cluster centers. It purports to be an efficient way to get much closer to the optimal clustering solution while reducing the impact of random chance. For more information, refer to *Arthur D, Vassilvitskii S.* k-means++: The advantages of careful seeding. *Proceedings of the eighteenth annual ACM-SIAM symposium on discrete algorithms.* 2007:1027–1035.

After choosing the initial cluster centers, the other examples are assigned to the cluster center that is nearest according to the distance function. You will remember that we studied distance functions while learning about k-Nearest Neighbors. Traditionally, k-means uses Euclidean distance, but Manhattan distance or Minkowski distance are also sometimes used.

Recall that if n indicates the number of features, the formula for Euclidean distance between example x and example y is:

$$\text{dist}(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

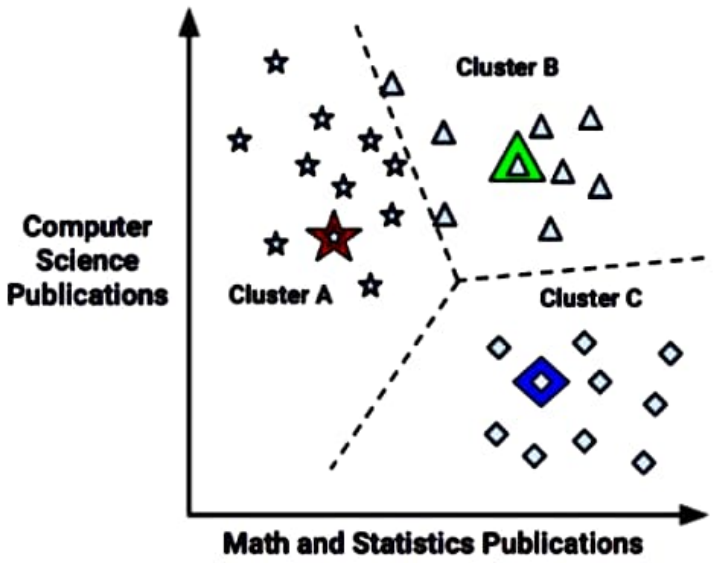
For instance, if we are comparing a guest with five computer science publications and one math publication to a guest with zero computer science papers and two math papers, we could compute this in R as follows:

```
> sqrt((5 - 0)^2 + (1 - 2)^2)
[1] 5.09902
```

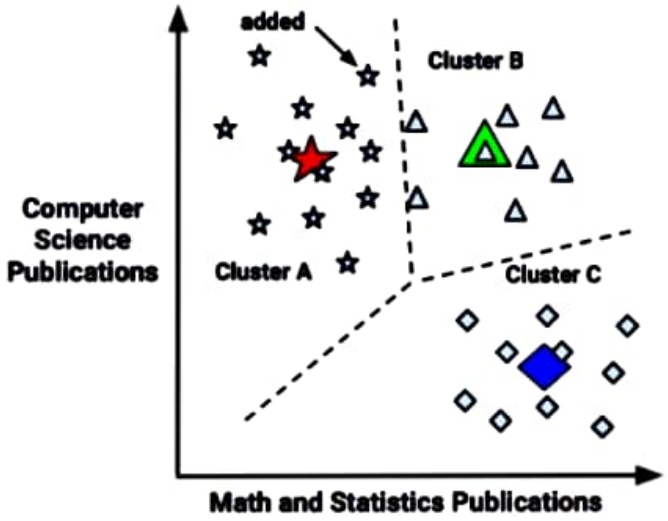
Using this distance function, we find the distance between each example and each cluster center. The example is then assigned to the nearest cluster center.

Keep in mind that as we are using distance calculations, all the features need to be numeric, and the values should be normalized to a standard range ahead of time. The methods discussed in *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*, will prove helpful for this task.

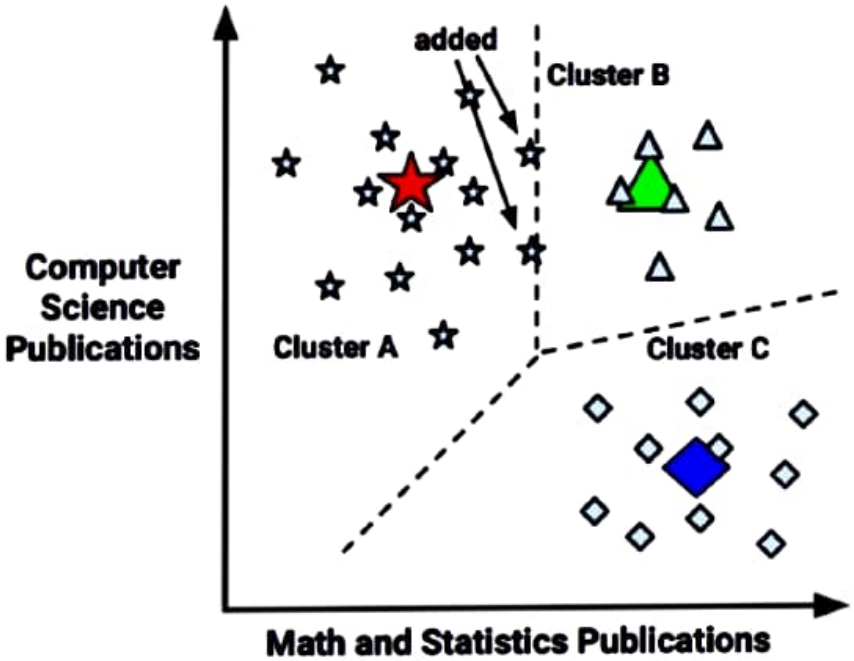
As shown in the following diagram, the three cluster centers partition the examples into three segments labeled **Cluster A**, **Cluster B**, and **Cluster C**. The dashed lines indicate the boundaries for the **Voronoi diagram** created by the cluster centers. The Voronoi diagram indicates the areas that are closer to one cluster center than any other; the vertex where all the three boundaries meet is the maximal distance from all three cluster centers. Using these boundaries, we can easily see the regions claimed by each of the initial k-means seeds:



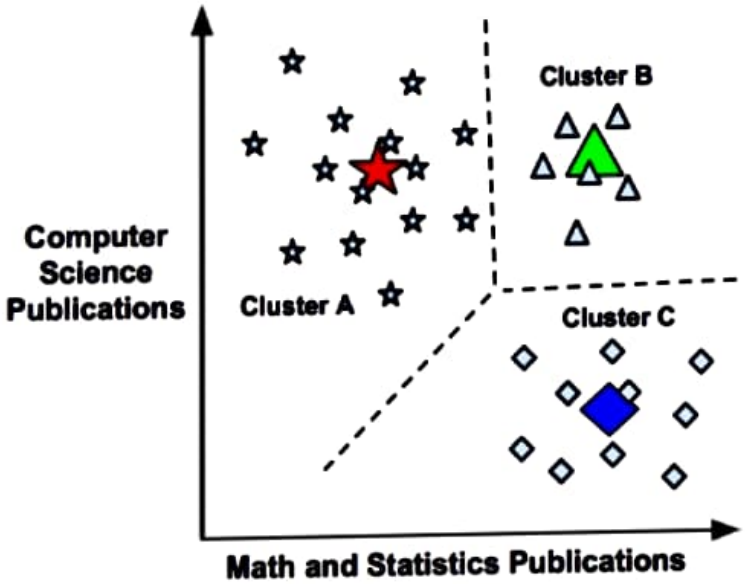
Now that the initial assignment phase has been completed, the k-means algorithm proceeds to the update phase. The first step of updating the clusters involves shifting the initial centers to a new location, known as the **centroid**, which is calculated as the average position of the points currently assigned to that cluster. The following diagram illustrates how as the cluster centers shift to the new centroids, the boundaries in the Voronoi diagram also shift and a point that was once in **Cluster B** (indicated by an arrow) is added to **Cluster A**:



As a result of this reassignment, the k-means algorithm will continue through another update phase. After shifting the cluster centroids, updating the cluster boundaries, and reassigning points into new clusters (as indicated by arrows), the figure looks like this:



Because two more points were reassigned, another update must occur, which moves the centroids and updates the cluster boundaries. However, because these changes result in no reassignments, the k-means algorithm stops. The cluster assignments are now final:



The final clusters can be reported in one of the two ways. First, you might simply report the cluster assignments such as A, B, or C for each example. Alternatively, you could report the coordinates of the cluster centroids after the final update. Given either reporting method, you are able to define the cluster boundaries by calculating the centroids or assigning each example to its nearest cluster.

Choosing the appropriate number of clusters

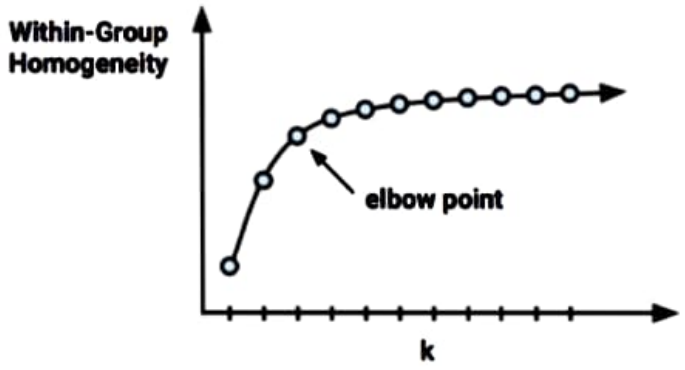
In the introduction to k-means, we learned that the algorithm is sensitive to the randomly-chosen cluster centers. Indeed, if we had selected a different combination of three starting points in the previous example, we may have found clusters that split the data differently from what we had expected. Similarly, k-means is sensitive to the number of clusters; the choice requires a delicate balance. Setting *k* to be very large will improve the homogeneity of the clusters, and at the same time, it risks overfitting the data.

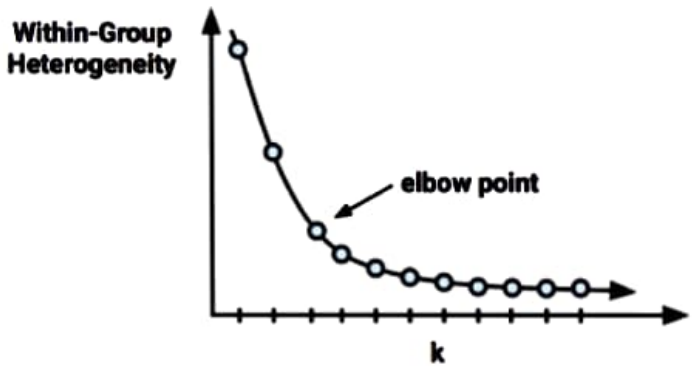
Ideally, you will have *a priori* knowledge (a prior belief) about the true groupings and you can apply this information to choosing the number of clusters. For instance, if you were clustering movies, you might begin by setting *k* equal to the number of genres considered for the Academy Awards. In the data science conference seating problem that we worked through previously, *k* might reflect the number of academic fields of study that were invited.

Sometimes the number of clusters is dictated by business requirements or the motivation for the analysis. For example, the number of tables in the meeting hall could dictate how many groups of people should be created from the data science attendee list. Extending this idea to another business case, if the marketing department only has resources to create three distinct advertising campaigns, it might make sense to set $k = 3$ to assign all the potential customers to one of the three appeals.

Without any prior knowledge, one rule of thumb suggests setting k equal to the square root of $(n / 2)$, where n is the number of examples in the dataset. However, this rule of thumb is likely to result in an unwieldy number of clusters for large datasets. Luckily, there are other statistical methods that can assist in finding a suitable k-means cluster set.

A technique known as the **elbow method** attempts to gauge how the homogeneity or heterogeneity within the clusters changes for various values of k. As illustrated in the following diagrams, the homogeneity within clusters is expected to increase as additional clusters are added; similarly, heterogeneity will also continue to decrease with more clusters. As you could continue to see improvements until each example is in its own cluster, the goal is not to maximize homogeneity or minimize heterogeneity, but rather to find k so that there are diminishing returns beyond that point. This value of k is known as the **elbow point** because it looks like an elbow.





There are numerous statistics to measure homogeneity and heterogeneity within the clusters that can be used with the elbow method (the following information box provides a citation for more detail). Still, in practice, it is not always feasible to iteratively test a large number of k values. This is in part because clustering large datasets can be fairly time consuming; clustering the data repeatedly is even worse. Regardless, applications requiring the exact optimal set of clusters are fairly rare. In most clustering applications, it suffices to choose a k value based on convenience rather than strict performance requirements.

For a very thorough review of the vast assortment of cluster performance measures, refer to: *Halkidi M, Batistakis Y, Vazirgiannis M.* On clustering validation techniques. *Journal of Intelligent Information Systems.* 2001; 17:107-145.

The process of setting *k* itself can sometimes lead to interesting insights. By observing how the characteristics of the clusters change as *k* is varied, one might infer where the data have naturally defined boundaries. Groups that are more tightly clustered will change a little, while less homogeneous groups will form and disband over time.

In general, it may be wise to spend little time worrying about getting *k* exactly right. The next example will demonstrate how even a tiny bit of subject-matter knowledge borrowed from a Hollywood film can be used to set *k* such that actionable and interesting clusters are found. As clustering is unsupervised, the task is really about what you make of it; the value is in the insights you take away from the algorithm's findings.

Example – finding teen market segments using k-means clustering

Interacting with friends on a **social networking service** (**SNS**), such as Facebook, Tumblr, and Instagram has become a rite of passage for teenagers around the world. Having a relatively large amount of disposable income, these adolescents are a coveted demographic for businesses hoping to sell snacks, beverages, electronics, and hygiene products.

The many millions of teenage consumers using such sites have attracted the attention of marketers struggling to find an edge in an increasingly competitive market. One way to gain this edge is to identify segments of teenagers who share similar tastes, so that clients can avoid targeting advertisements to teens with no interest in the product being sold. For instance, sporting apparel is likely to be a difficult sell to teens with no interest in sports.

Given the text of teenagers' SNS pages, we can identify groups that share common interests such as sports, religion, or music. Clustering can automate the process of discovering the natural segments in this population. However, it will be up to us to decide whether or not the clusters are interesting and how we can use them for advertising. Let's try this process from start to finish.

10
Evaluating Model Performance

When only the wealthy could afford education, tests and exams did not evaluate students' potential. Instead, teachers were judged for parents who wanted to know whether their children had learned enough to justify the instructors' wages. Obviously, this has changed over the years. Now, such evaluations are used to distinguish between high- and low-achieving students, filtering them into careers and other opportunities.

Given the significance of this process, a great deal of effort is invested in developing accurate student assessments. Fair assessments have a large number of questions that cover a wide breadth of topics and reward true knowledge over lucky guesses. They also require students to think about problems they have never faced before. Correct responses therefore indicate that students can generalize their knowledge more broadly.

The process of evaluating machine learning algorithms is very similar to the process of evaluating students. Since algorithms have varying strengths and weaknesses, tests should distinguish among the learners. It is also important to forecast how a learner will perform on future data.

This chapter provides the information needed to assess machine learners, such as:

- The reasons why predictive accuracy is not sufficient to measure performance, and the performance measures you might use instead
- Methods to ensure that the performance measures reasonably reflect a model's ability to predict or forecast unseen cases
- How to use R to apply these more useful measures and methods to the predictive models covered in the previous chapters

Just as the best way to learn a topic is to attempt to teach it to someone else, the process of teaching and evaluating machine learners will provide you with greater insight into the methods you've learned so far.

Measuring performance for classification

In the previous chapters, we measured classifier accuracy by dividing the proportion of correct predictions by the total number of predictions. This indicates the percentage of cases in which the learner is right or wrong. For example, suppose that for 99,990 out of 100,000 newborn babies a classifier correctly predicted whether they were a carrier of a treatable but potentially fatal genetic defect. This would imply an accuracy of 99.99 percent and an error rate of only 0.01 percent.

At first glance, this appears to be an extremely accurate classifier. However, it would be wise to collect additional information before trusting your child's life to the test. What if the genetic defect is found in only 10 out of every 100,000 babies? A test that predicts *no defect* regardless of the circumstances will be correct for 99.99 percent of all cases, but incorrect for 100 percent of the cases that matter most. In other words, even though the predictions are extremely accurate, the classifier is not very useful to prevent treatable birth defects.

This is one consequence of the **class imbalance problem**, which refers to the trouble associated with data having a large majority of records belonging to a single class.

Though there are many ways to measure a classifier's performance, the best measure is always the one that captures whether the classifier is successful at its intended purpose. It is crucial to define the performance measures for utility rather than raw accuracy. To this end, we will begin exploring a variety of alternative performance measures derived from the confusion matrix. Before we get started, however, we need to consider how to prepare a classifier for evaluation.

Working with classification prediction data in R

The goal of evaluating a classification model is to have a better understanding of how its performance will extrapolate to future cases. Since it is usually unfeasible to test a still-unproven model in a live environment, we typically simulate future conditions by asking the model to classify a dataset made of cases that resemble what it will be asked to do in the future. By observing the learner's responses to this examination, we can learn about its strengths and weaknesses.

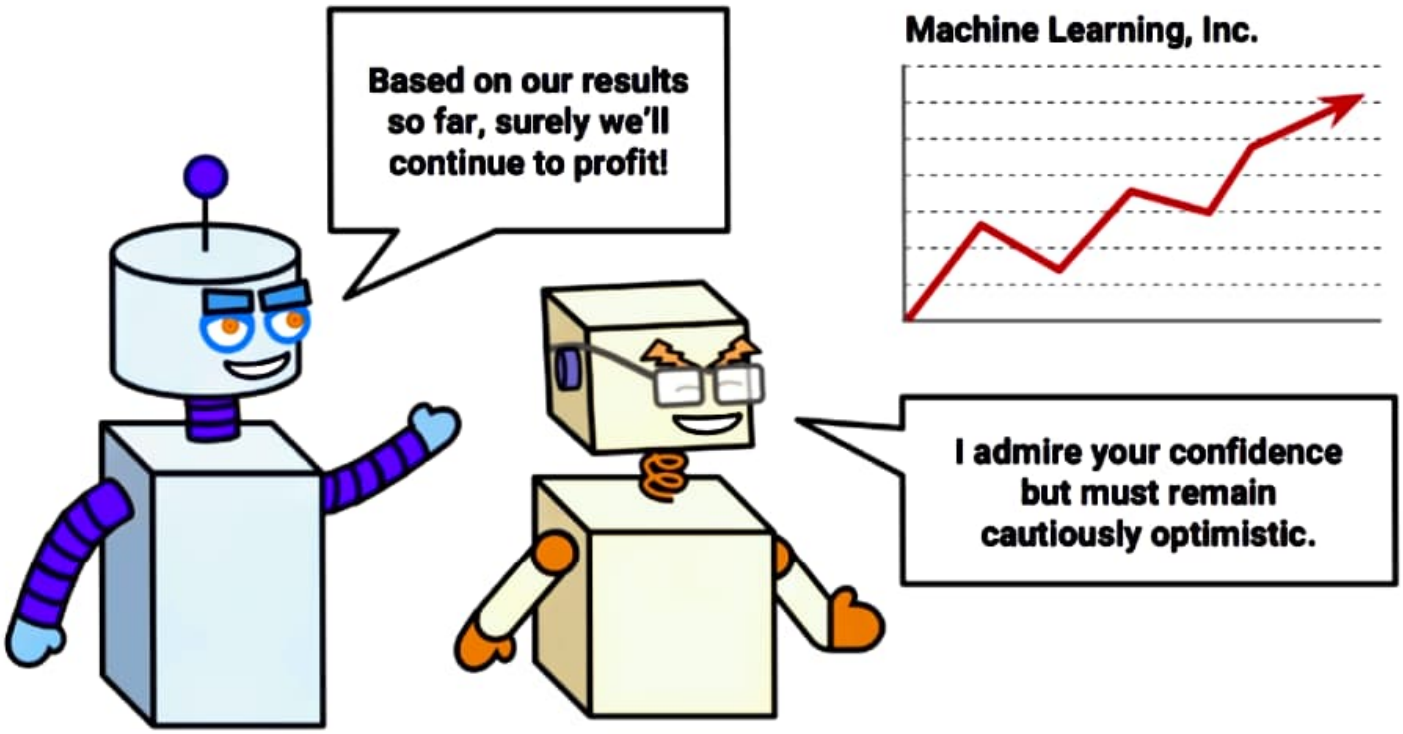
Though we've evaluated classifiers in the prior chapters, it's worth reflecting on the types of data at our disposal:

- Actual class values
- Predicted class values
- Estimated probability of the prediction

The actual and predicted class values may be self-evident, but they are the key to evaluation. Just like a teacher uses an answer key to assess the student's answers, we need to know the correct answer for a machine learner's predictions. The goal is to maintain two vectors of data: one holding the correct or actual class values, and the other holding the predicted class values. Both vectors must have the same number of values stored in the same order. The predicted and actual values may be stored as separate R vectors or columns in a single R data frame.

Obtaining this data is easy. The actual class values come directly from the target feature in the test dataset. Predicted class values are obtained from the classifier built upon the training data, and applied to the test data. For most machine learning packages, this involves applying the `predict()` function to a model object and a data frame of test data, such as: `predicted_outcome <- predict(model, test_data)`.

Until now, we have only examined classification predictions using these two vectors of data. Yet most models can supply another piece of useful information. Even though the classifier makes a single prediction about each example, it may be more confident about some decisions than others. For instance, a classifier may be 99 percent certain that an SMS with the words "free" and "ringtones" is spam, but is only 51 percent certain that an SMS with the word "tonight" is spam. In both cases, the classifier classifies the message as spam, but it is far more certain about one decision than the other.



Studying these internal prediction probabilities provides useful data to evaluate a model's performance. If two models make the same number of mistakes, but one is more capable of accurately assessing its uncertainty, then it is a smarter model. It's ideal to fid a learner that is extremely confident when making a correct prediction, but timid in the face of doubt. The balance between confidence and caution is a key part of model evaluation.

Unfortunately, obtaining internal prediction probabilities can be tricky because the method to do so varies across classifiers. In general, for most classifiers, the `predict()` function is used to specify the desired type of prediction. To obtain a single predicted class, such as spam or ham, you typically set the `type = "class"` parameter. To obtain the prediction probability, the `type` parameter should be set to one of `"prob"`, `"posterior"`, `"raw"`, or `"probability"` depending on the classifier used.

Nearly all of the classifiers presented in this book will provide prediction probabilities. The `type` parameter is included in the syntax box introducing each model.

For example, to output the predicted probabilities for the C5.0 classifier built in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, use the `predict()` function with `type = "prob"` as follows:

```
> predicted_prob <- predict(credit_model, credit_test, type = "prob")
```

To further illustrate the process of evaluating learning algorithms, let's look more closely at the performance of the SMS spam classification model developed in *Chapter 4, Probabilistic Learning – Classification Using Naive Bayes*. To output the naive Bayes predicted probabilities, use `predict()` with `type = "raw"` as follows:

```
> sms_test_prob <- predict(sms_classifier, sms_test, type = "raw")
```

In most cases, the `predict()` function returns a probability for each category of the outcome. For example, in the case of a two-outcome model like the SMS classifier, the predicted probabilities might be a matrix or data frame as shown here:

```
> head(sms_test_prob)
                ham          spam
[1,] 9.999995e-01 4.565938e-07
[2,] 9.999995e-01 4.540489e-07
[3,] 9.998418e-01 1.582360e-04
[4,] 9.999578e-01 4.223125e-05
[5,] 4.816137e-10 1.000000e+00
[6,] 9.997970e-01 2.030033e-04
```

Each line in this output shows the classifier's predicted probability of `spam` and `ham`, which always sum up to 1 because these are the only two outcomes. While constructing an evaluation dataset, it is important to ensure that you are using the correct probability for the class level of interest. To avoid confusion, in the case of a binary outcome, you might even consider dropping the vector for one of the two alternatives.

For convenience during the evaluation process, it can be helpful to construct a data frame containing the predicted class values, actual class values, as well as the estimated probabilities of interest.

The steps required to construct the evaluation dataset have been omitted for brevity, but are included in this chapter's code on the Packt Publishing website. To follow along with the examples here, download the `sms_results.csv` file, and load to a data frame using the `sms_results <- read.csv("sms_results.csv")` command.

The `sms_results` data frame is simple. It contains four vectors of 1,390 values. One vector contains values indicating the actual type of SMS message (`spam` or `ham`), one vector indicates the naive Bayes model's predicted type, and the third and fourth vectors indicate the probability that the message was `spam` or `ham`, respectively:

```
> head(sms_results)
  actual_type predict_type prob_spam prob_ham
1         ham          ham   0.00000  1.00000
2         ham          ham   0.00000  1.00000
3         ham          ham   0.00016  0.99984
4         ham          ham   0.00004  0.99996
5        spam         spam   1.00000  0.00000
6         ham          ham   0.00020  0.99980
```

For these six test cases, the predicted and actual SMS message types agree; the model predicted their status correctly. Furthermore, the prediction probabilities suggest that model was extremely confident about these predictions, because they all fall close to zero or one.

What happens when the predicted and actual values are further from zero and one? Using the `subset()` function, we can identify a few of these records. The following output shows test cases where the model estimated the probability of `spam` somewhere between 40 and 60 percent:

```
> head(subset(sms_results, prob_spam > 0.40 & prob_spam < 0.60))
     actual_type predict_type prob_spam prob_ham
377         spam          ham   0.47536  0.52464
717          ham         spam   0.56188  0.43812
1311         ham         spam   0.57917  0.42083
```

By the model's own admission, these were cases in which a correct prediction was virtually a coin flip. Yet all three predictions were wrong—an unlucky result. Let's look at a few more cases where the model was wrong:

```
> head(subset(sms_results, actual_type != predict_type))
    actual_type predict_type prob_spam prob_ham
```

53	spam	ham	0.00071	0.99929
59	spam	ham	0.00156	0.99844
73	spam	ham	0.01708	0.98292
76	spam	ham	0.00851	0.99149
184	spam	ham	0.01243	0.98757
332	spam	ham	0.00003	0.99997

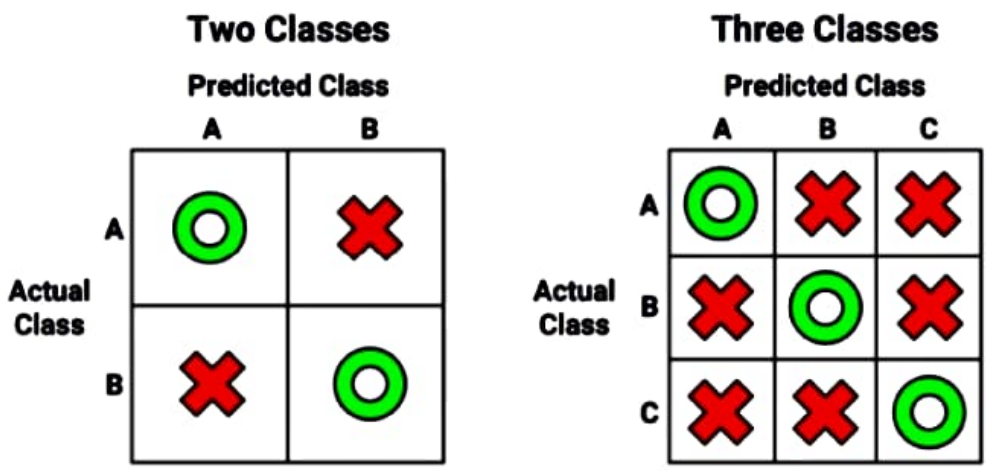
These cases illustrate the important fact that a model can be extremely confident and yet it can be extremely wrong. All six of these test cases were `spam` that the classifier believed to have no less than a 98 percent chance of being `ham`.

In spite of such mistakes, is the model still useful? We can answer this question by applying various error metrics to the evaluation data. In fact, many such metrics are based on a tool we've already used extensively in the previous chapters.

A closer look at confusion matrices

A **confusion matrix** is a table that categorizes predictions according to whether they match the actual value. One of the table's dimensions indicates the possible categories of predicted values, while the other dimension indicates the same for actual values. Although we have only seen 2 x 2 confusion matrices so far, a matrix can be created for models that predict any number of class values. The following figure depicts the familiar confusion matrix for a two-class binary model as well as the 3 x 3 confusion matrix for a three-class model.

When the predicted value is the same as the actual value, it is a correct classification. Correct predictions fall on the diagonal in the confusion matrix (denoted by **O**). The off-diagonal matrix cells (denoted by **X**) indicate the cases where the predicted value differs from the actual value. These are incorrect predictions. The performance measures for classification models are based on the counts of predictions falling on and off the diagonal in these tables:



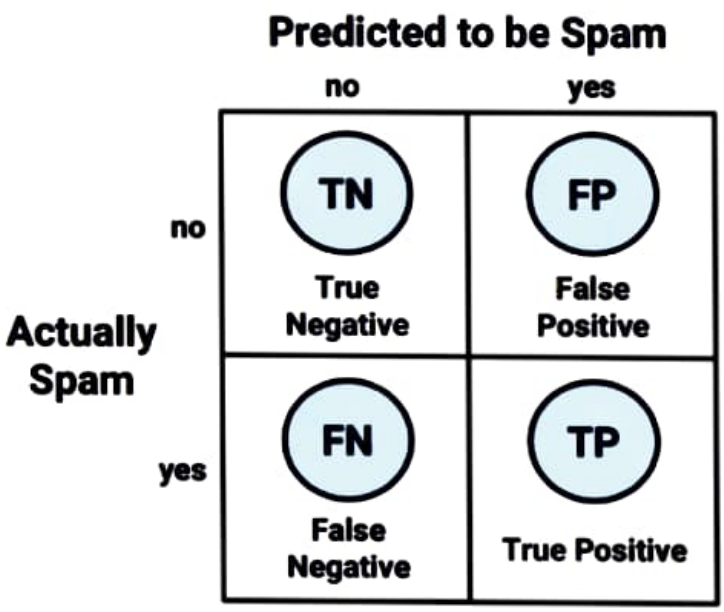
The most common performance measures consider the model's ability to discern one class versus all others. The class of interest is known as the **positive** class, while all others are known as **negative**.

The use of the terms positive and negative is not intended to imply any value judgment (that is, good versus bad), nor does it necessarily suggest that the outcome is present or absent (such as birth defect versus none). The choice of the positive outcome can even be arbitrary, as in cases where a model is predicting categories such as sunny versus rainy or dog versus cat.

The relationship between the positive class and negative class predictions can be depicted as a 2 x 2 confusion matrix that tabulates whether predictions fall into one of the four categories:

- **True Positive (TP)**: Correctly classified as the class of interest
- **True Negative (TN)**: Correctly classified as not the class of interest
- **False Positive (FP)**: Incorrectly classified as the class of interest
- **False Negative (FN)**: Incorrectly classified as not the class of interest

For the spam classifier, the positive class is `spam`, as this is the outcome we hope to detect. We can then imagine the confusion matrix as shown in the following diagram:



The confusion matrix, presented in this way, is the basis for many of the most important measures of model's performance. In the next section, we'll use this matrix to have a better understanding of what is meant by accuracy.