Brian Steele · John Chandler
Swarna Reddy

Algorithms for Data Science

Springer

Algorithms for Data Science

Brian Steele • John Chandler • Swarna Reddy

Algorithms for Data Science

Springer

Brian Steele
University of Montana
Missoula, MT, USA

Swarna Reddy
SoftMath Consultants, LLC
Missoula, MT, USA

John Chandler
School of Business Administration
University of Montana
Missoula, MT, USA

ISBN 978-3-319-45795-6 ISBN 978-3-319-45797-0 (eBook)
DOI 10.1007/978-3-319-45797-0

Library of Congress Control Number: 2016952812

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Data science has been recognized as a science since 2001, roughly. Its origin lies in technological advances that are generating nearly inconceivable volumes of data. The rate at which new data are being produced is not likely to slow for some time. As a society, we have realized that these data provide opportunities to learn about the systems and processes generating the data. But data in its original form is of relatively little value. Paradoxically, the more of it that there is, the less the value. It has to be reduced to extract value from it. Extracting information from data is the subject of data science.

Becoming a successful practitioner of data science is a real challenge. The knowledge base incorporates demanding topics from statistics, computer science, and mathematics. On top of that, domain-specific knowledge, if not critical, is very helpful. Preparing students in these three or four areas is necessary. But at some point, the subject areas need to be brought together as a coherent package in what we consider to be a course in *data science*. A student that lacks a course that actually teaches data science is not well prepared to practice data science. This book serves as a backbone for a course that brings together the main subject areas.

We've paid attention to the needs of employers with respect to entry-level data scientists—and what they say is lacking from the skills of these new data scientists. What is most lacking are programming abilities. From the educators' point of view, we want to teach principles and theory—the stuff that's needed by students to learn on their own. We're not going to be able to teach them everything they need in their careers, or even in the short term. But teaching principles and foundations is the best preparation for independent learning. Fortuitously, there is a subject that encompasses both principles and programming—algorithms. Therefore, this book has been written about the algorithms of data science.

Algorithms for Data Science focuses on the principles of data reduction and core algorithms for analyzing the data of data science. Understanding the fundamentals is crucial to be able to adapt existing algorithms and create new algorithms. The text provides many opportunities for the reader to develop and improve their programming skills. Every algorithm discussed at length is accompanied by a tutorial that guides the reader through implementation of the algorithm in either `Python` or `R`. The algorithm is then applied to a real-world data set. Using real data allows us to talk about domain-specific problems. Regrettably, our self-imposed coding edict eliminates some important predictive analytic algorithms because of their complexity.

We have two audiences in mind. One audience is practitioners of data science and the allied areas of statistics, mathematics, and computer science. This audience would read the book if they have an interest in improving their analytical skills, perhaps with the objective of working as a data scientist. The second audience are upper-division undergraduate and graduate students in data science, business analytics, mathematics, statistics, and computer science. This audience would be engaged in a course on data analytics or self-study.

Depending on the sophistication of the audience, the book may be used for a one- or two-semester course on data analytics. If used for a one-semester course, the instructor has several options regarding the course content. All options begin with Chaps. 1 and 2 so that the concepts of data reduction and data dictionaries are firmly established.

1. If the instructional emphasis is on computation, then Chaps. 3 and 4 on methods for massively large data and distributed computing would be covered. Chapter 12 works with streaming data, and so this chapter is a nice choice to close the course. Chapter 7 on healthcare analytics is optional and might be covered as time allows. The tutorials of Chap. 7 involve relatively large and challenging data sets. These data sets provide the student and instructor with many opportunities for interesting projects.
2. A course oriented toward general analytical methods might pass over Chaps. 3 and 4 in favor of data visualization (Chap. 5) and linear regression (Chap. 6). The course could close with Chap. 9 on k-nearest neighbor prediction functions and Chap. 11 on forecasting.
3. A course oriented toward predictive analytics would focus on Chaps. 9 and 10 on k-nearest neighbor and naïve Bayes prediction functions. The course would close with Chaps. 11 and 12 on forecasting and streaming data.

Acknowledgments

We thank Brett Kassner, Jason Kolberg, and Greg St. George for reviewing chapters and Guy Shepard for help solving hardware problems and unraveling network mysteries. Many thanks to Alex Philp for anticipating the future and breaking trail. We thank Leonid Kalachev and Peter Golubstov for many interesting conversations and insights.

Missoula, MT, USA Brian Steele
Missoula, MT, USA John Chandler
Missoula, MT, USA Swarna Reddy

Contents

List of Figures

List of Tables

Biographical Sketches

Brian Steele

Brian Steele is a full professor of Mathematics at the University of Montana and a Senior Data Scientist for SoftMath Consultants, LLC. Dr. Steele has published on the EM algorithm, exact bagging, the bootstrap, and numerous statistical applications. He teaches data analytics and statistics and consults on a wide variety of subjects related to data science and statistics.

John Chandler

John has worked at the forefront of marketing and data analysis since 1999. He has worked with Fortune 100 advertisers and scores of agencies, measuring the effectiveness of advertising and improving performance. Dr. Chandler joined the faculty at the University of Montana School of Business Administration as a Clinical Professor of Marketing in 2015 and teaches classes in advanced marketing analytics and data science. He is one of the founders and Chief Data Scientist for Ars Quanta, a Seattle-based data science consultancy.

Swarna Reddy

Dr. Swarna Reddy is the founder, CEO, and a Senior Data Scientist for SoftMath Consultants, LLC and serves as a faculty affiliate with the Department of Mathematical Sciences at the University of Montana. Her area of expertise is computational mathematics and operations research. She is a published researcher and has developed computational solutions across a wide variety of areas—spanning bioinformatics, cybersecurity, and business analytics.

Part II
Extracting Information from Data

Chapter 5
Data Visualization

The drawing shows me at one glance what might be spread over ten pages in a book. — Ivan Turgenev, *Fathers and Sons*

Abstract A visual is successful when the information encoded in the data is efficiently transmitted to an audience. Data visualization is the discipline dedicated to the principles and methods of translating data to visual form. In this chapter we discuss the principles that produce successful visualizations. The second section illustrates the principles through examples of best and worst practices. In the final section, we navigate through the construction of our best-example graphics.

5.1 Introduction

Humans are visual animals. We absorb sensory information most efficiently through vision. It's no surprise that data visualization is very effective for extracting information from data. As we see throughout this volume, gathering large amounts of data has never been easier. Even 40 years ago, displaying that information was difficult to do well, requiring specialized tools or a steady hand. The democratization of creating figures from data allows us to create more visualizations than ever before. And with this profusion comes the ability to develop best practices. Figures *encode* information from a data set, displaying those data as ink on a page or, more commonly, as pixels on a screen. This encoding makes use of a vernacular that has developed over the last several centuries. The typical audience will understand the Cartesian plane and its axes. We understand how to determine values of points in a scatterplot and easily manage the color-coding of groups.

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0_5

A visualization is effective when it can be quickly and accurately *decoded* by the audience—the salient points should be almost immediately apparent. Edward Tufte, a vocal evangelist for better graphics, calls this the "Interocular Trauma Test": does the visualization hit one immediately between the eyes? A common expression about scripting languages, like `Python` and `Perl`, is that they should make easy things easy and difficult things possible. The same can be said of good visualizations: the key features of the narrative should emerge immediately and the more subtle relationships should be visible whenever possible.

Data visualization combines several different threads and we will cover each in a section. The first is an understanding of the guiding principles of graphics. In this section we lean heavily on the pioneering work of William Cleveland, the researcher who truly brought graphics into the modern era. Most of the excellent graphics one sees in data science journalism[1] are built using the ideas he introduced. The second is a basic understanding of the paradigms of graphics that are often employed for data of different types. Knowing these will allow us to create an abstract version of the graphic. This ability, to sketch a version of the graphic you wish to create, is critical. It is impossible to follow a map if you don't know where you're going. Finally, we must be able to tell a software package how to render the image that is in our minds. The best tool for creating data visualizations in the context of an analysis[2] is Hadley Wickham's `ggplot2` available in `R` [65, 66]. The "gg" in the package name stands for "The Grammar of Graphics". You can think of this grammar as being a semi-secret language that, for better or worse, you must be fluent in to realize the potential of graphics from data. Recognizing this reality, we will introduce the grammar and illustrate its implementation in `R` syntax.

The data used in this chapter and Chap. 10 originates from the largest cooperative (co-op) grocery store in the United States. We received approximately 20 gigabytes of transaction-level data covering 6 years of store activity. The data are essentially cash register or point-of-sales receipts. As is common with automatically recorded point-of-sales information, a considerable amount of associated meta-data is attached to the receipt. We work a great deal with two variables: the department classification of the item (e.g., produce) and whether the shopper is a member of the co-op. The co-op is member-owned and approximately three-quarters of the transaction records

[1] Three great examples:

- The Upshot from the New York Times: `http://www.nytimes.com/section/upshot`.
- Five Thirty Eight, Nate Silver's organization that has largely invented the field of data science journalism. `http://fivethirtyeight.com`.
- Flowing Data, a site created by Nathan Yau dedicated to creating beautiful and informative data visualizations. `http://flowingdata.com`.

[2] If you are building interactive graphics or large-scale graphics via the web, there are better tools. Check out `bootstrap`, `D3`, and `crossfilter`.

originated from members. The remainder originated from non-members. If the shopper is a member, then a unique, anonymous identifier is attached to receipt that allows us to analyze data at the shopper level.

We use the co-op data in this chapter to illustrate a variety of data visualizations. In Chap. 10, we develop a prediction function that classifies non-member shoppers to customer segment with the ultimate goal of better understanding the non-member shoppers.

5.2 Principles of Data Visualization

When describing what makes good data visualization, there are two paths to follow: that of simplicity and that of exhaustiveness. There are entire books of the latter variety and thus we opt for a treatment that will give the reader minimal guidelines and that points them in the direction of the more thorough treatments. Our goal, after all, is to be able to make good visualizations and improve on those found in the wild.

A data visualization is effective when the information that the analyst is trying to convey to the audience is transmitted. Complicated information sometimes requires complicated visualizations, but often we can organize our thoughts in terms of principles.

Let the Data Speak. If there is one overarching goal, then this is it: the data should be allowed to speak for itself. As Strunk and White say, "vigorous writing is concise." Similarly, good data visualization allows the data to tell its story. Tufte coined a term for elements of a visualization that do not add to understanding: *chartjunk*. A good, revised definition of chartjunk from Robert Kosara at Tableau is this: any element of a chart that does not contribute to clarifying the intended message. Historically, the most egregious violations of this principle came from Excel. The default settings in Excel now avoid the worst examples of chartjunk, but options to add them abound, particularly with the addition of mysterious third dimensions to one- or two-dimensional data sets, color coding for no reason, and patterns that impede understanding. For each element of a figure, we must ask if that element is serving our principal goal of letting the data speak.

Let the Data Speak Clearly. A subtle addition to our previous point is to let the data tell their story clearly and quickly. As we shall see, the relationship between two variables, often plotted against one another in a scatterplot, can be illuminated by the addition of a smoothed or fitted line. At the other end of the spectrum, Fig. 5.1 illustrates how certain visualizations, in this case the much- and rightly-maligned pie chart can obfuscate the story. In the pie chart version, it's extremely difficult to identify the predominant pattern—the presence of two sets of observations with different means. The lower panel in Fig. 5.1 shows a dotchart which

lets the data speak for itself and uses a sensible ordering of observations. This visualization conveys much more information and allows for it to be immediately decoded.

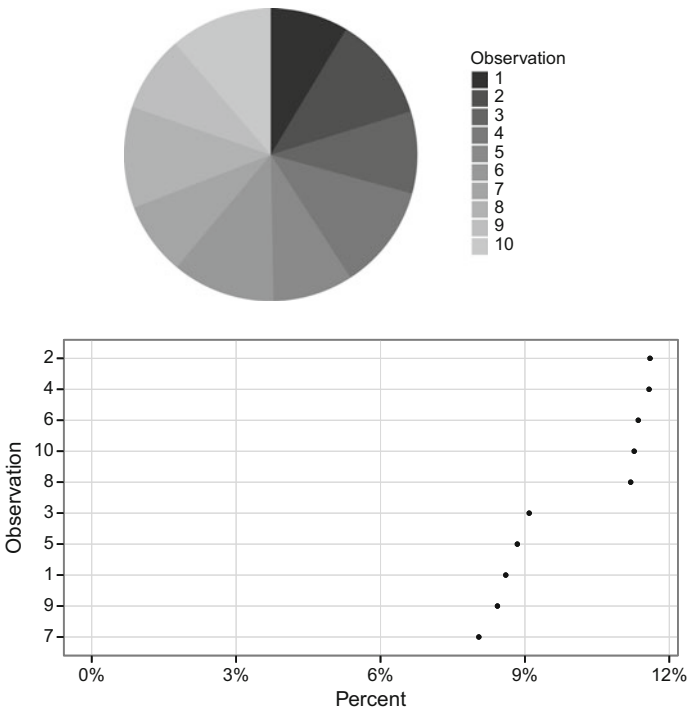

Fig. 5.1 A pie chart makes patterns in the data difficult to decode; the dotchart is an improvement

We introduce the dotchart in Sect. 5.3 and it is an excellent choice for displaying univariate data. There are other types of graphs besides the pie chart that inhibit understanding, notably stacked bar charts and area charts. The stacked bar chart makes it difficult to understand the behavior of the individual elements, as we see in Fig. 5.2. This figure has two views of average sales data by month for a grocery store for four large departments. In the upper panel, we can see that May is the month of maximum overall sales (by looking at the heights of bars and ignoring the colors). We can see that packaged grocery appears to have lower sales around July and that meat and bulk appear to be smaller departments, although it is difficult to gauge the magnitude of the difference. In the lower panel we have replaced the stacked bars with points connected by lines. The points allow precise estimation of individual observations and the lines (and the color) help us group the departments together across time. Now we see that packaged grocery and produce are larger than the other two departments, by a factor of just more than two. Note that the lower panel figure is less than perfect

because the vertical axis does not extend to 0. We also see that produce has an annual cycle out of phase with the other departments, peaking in the North American summer months. Area charts such as the pie chart typically arise when circles are scaled based on some variable that would not otherwise be plotted. There are cases where this is useful, for instance, adding sample size to a chart via area scaling. Research indicates that people are able to decode area to perceive order but magnitude is difficult accurately translate from area.

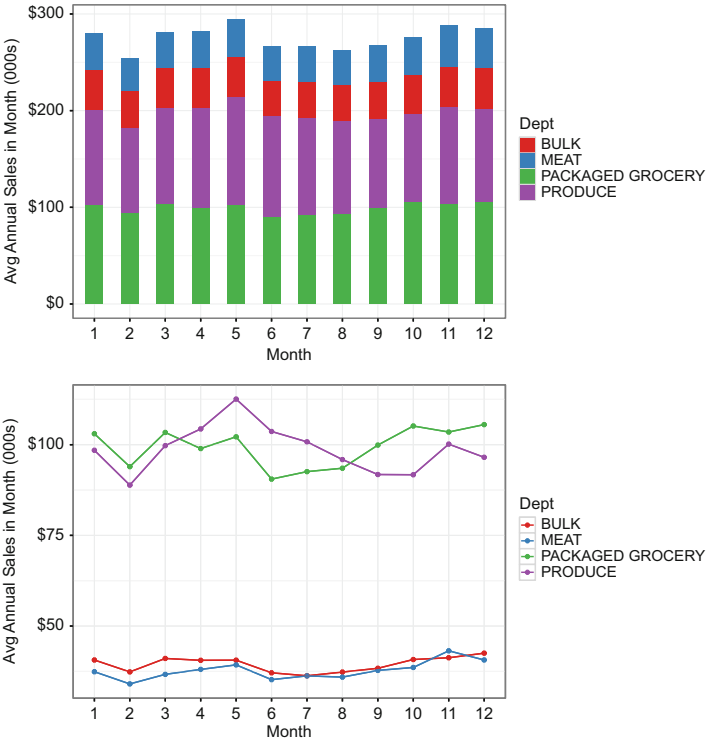

Fig. 5.2 Two views of monthly sales for four departments. The stacked bar chart obfuscates much information that the line chart makes clear

Choose Graphical Elements Judiciously. As one builds graphics, there are many choices: colors, shading, line types, line widths, plotting characters, axes, tick marks, legends, etc. Make these choices thoughtfully. Color is often used well when colors indicate membership according to a categorical variable. Color is often used poorly when practitioners get bored and add color haphazardly. Axes can be used intelligently to highlight certain observations or the range of the data. Smoothing lines can illustrate trends in bivariate data or mistakenly cover up observations. With a good graphics package, like the `R` package `ggplot2` which we introduce in Sect. 5.4, every element of a figure can be manipulated. Take advantage of the opportunities.

Help Your Audience. Wherever possible, make adjustments to your figure that helps your audience better understand the data. A choice had to be made in the lower panel of Fig. 5.1. The default behavior sorts the observations by name. This would be the desired order if our goal was to allow the reader to quickly find a given observation in the list and look up the value.[3] But if this is the goal, a table might be a better choice. By sorting the observations by value we can instantly see the minimum and maximum and the observations that define the break between the two groups. By manipulating the data using the `R` function `reorder`, we help the audience in several ways. Another example of this principle would be to highlight important observations by labeling them.

Limit Your Scope. Most interesting projects generate a profusion of data and, as our tools to visualize that data grow, so does the temptation to try to tell the entirety of a story with a single graphic. A well-written paragraph has a topic sentence and a unifying idea. Applying this concept to your graphics requires discipline and attention to detail. There will be times when you are tempted to add additional elements to a chart that already tells the story. Take care that the new elements do not detract from the central message. The classic example of overreaching is a line chart with two different axes for the lines. If you find yourself building such a chart, you are unlikely to be telling the story with clarity and power.

Armed with these general principles, we are now well positioned in the next section to delve into the types of data that are likely to be encountered. We'll describe the elements that make useful visualizations for univariate, bivariate, and multivariate data. In the subsequent section we learn how to produce these graphics in `R`.

5.3 Making Good Choices

Many of our visualization tasks will be defined by the number and type of variables that we will be plotting. The first and most important distinction is between quantitative and qualitative data. A qualitative, or categorical, variable is coded as a `factor` in `R`. In terms of the number of variables to plot, there are relatively clear approaches when we have one or two variables. Once we move beyond that, there are some general principles in effect but creativity plays a larger and larger role. One must be mindful of the edict to not try to do too much with one chart.

[3] When ordering is a problem, it is often referred to as the "Alabama First!" problem, given how often Alabama ends up at the top of lists that are thoughtlessly put, or left, in alphabetical order. Arrange your lists, like your factors, in an order that makes sense.

5.3.1 Univariate Data

Our goals with univariate quantitative data typically are to understand the distribution of the data. We typically begin describing the center and spread of the distribution and identifying unusual observations, often called *outliers*. More in-depth analyses describe the shape of the distribution, a task that provides more information and requires more effort. The classical starting point for investigating shape is either the histogram (described at length in Chap. 3, Sect. 3.4.2), the empirical density function, or one of several variations on the boxplot. In this first section we work with a grocery-store data set, specifically, sales and items by month and department for 6 years. We begin by looking at the distribution of sales, in units of thousands of dollars.

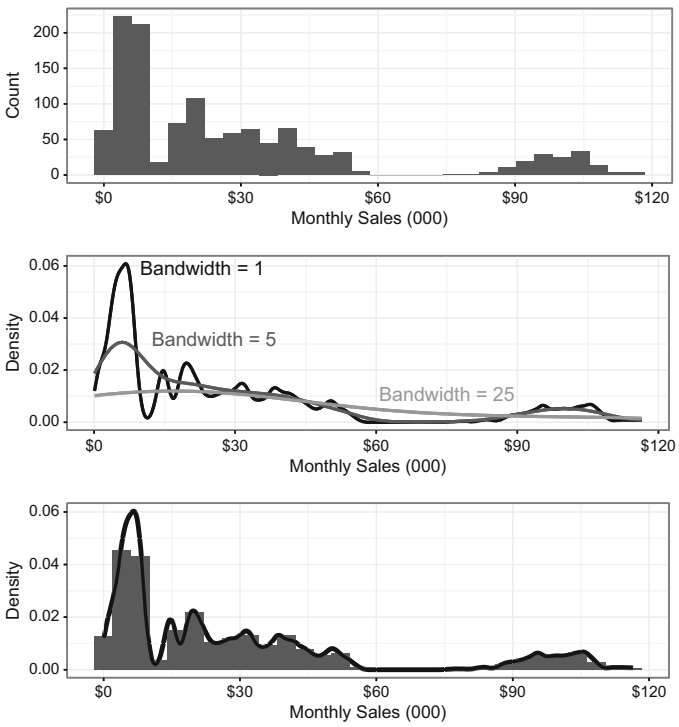

Fig. 5.3 Three different ways of looking at monthly sales by department in the grocery store data: a histogram, several empirical densities illustrating the variations possible from the bandwidth parameter, and a density superimposed on a histogram

The top panel of Fig. 5.3 shows a histogram depicting the numbers of observations falling in each interval, or *bin*, by the height of a vertical bars. We see sales by month by grocery department with bar height representing the count of observations that fall into that "bin", as the intervals on the x-axis are referred to.

The second panel illustrates a more powerful and complex way to visualize the distribution of a quantitative variable—empirical density functions. An empirical density function is a data-driven estimate of the probability distribution from which the data originated. More simply, they are smooth histograms. The formula that generates the empirical density function

$$\widehat{f}_b(x) = \frac{1}{n \cdot b} \sum_{i=1}^{n} K\left(\frac{x - x_i}{b}\right), \tag{5.1}$$

where n is the number of observations, b is the bandwidth, and K is the kernel.[4] The bandwidth is usually used to control the smoothness of the function by determining the influence of individual differences $x - x_i$ on the function. Setting the bandwidth to a small value, say 1, results in a very bumpy distribution. Setting it to a large value arguably makes the distribution overly smooth and removes the some of the secondary modes visible with $b = 1$. Much like the bin width choice for histograms, some trial and error may be necessary to capture the interesting features of the distribution. The default bandwidth in `R` which chosen by `bw.nrd0` and remains the default for historical reasons. The most recommended choice is `bw.SJ`, based on the method of Sheather and Jones [54]. We show how to set the bandwidth in Sect. 5.4.2.

The middle panel of Fig. 5.3 shows three different bandwidths. The bottom panel shows the histogram and density on the same plot, which necessitated changing the units on the histogram to relative frequency. Note that all depictions show some department-months with low sales, a larger group from \$20K to \$55K, and a group at \$100K.

Another useful way to display distributions uses boxplots or violin plots. These plots, illustrated in Fig. 5.4, are more compact displays of the distribution of monthly sales. Boxplots, first developed by John Tukey in the 1970s, summarize the distribution using five numbers. The box is defined by the first and third quartiles, Q_1 and Q_3, and is split by the median. The interquartile range is $\text{IQR} = Q_3 - Q_1$. The IQR is a useful nonparametric measure of spread. The whiskers, the length of which can be customized in `R`, are by default set to $Q_1 - 1.5 \times \text{IQR}$ and $Q_3 + 1.5 \times \text{IQR}$. Points beyond the whiskers are plotted individually and identified as outliers.

The definition of an outlier may seem arbitrary, but it captures data features in a predictable way if the data are normally distributed. Given a normal distribution, the 25th percentile of the data is $x_{25} = \mu - .67 \times \sigma$ and the IQR

[4] Kernels are an interesting side area of statistics and we will encounter them later in the chapter when we discuss `loess` smoothers. In order for a function to be a kernel, it must integrate to 1 and be symmetric about 0. The kernel is used to average the points in a neighborhood of a given value x. A simple average corresponds to a uniform kernel (all points get the same weight). Most high-performing kernels uses weights that diminish to 0 as you move further from a given x. The Epanechnikov kernel, which drops off with the square of distance and goes to zero outside a neighborhood, can be shown to be optimal with respect to mean square error. Most practitioners use Gaussian kernels, the default in `R`.

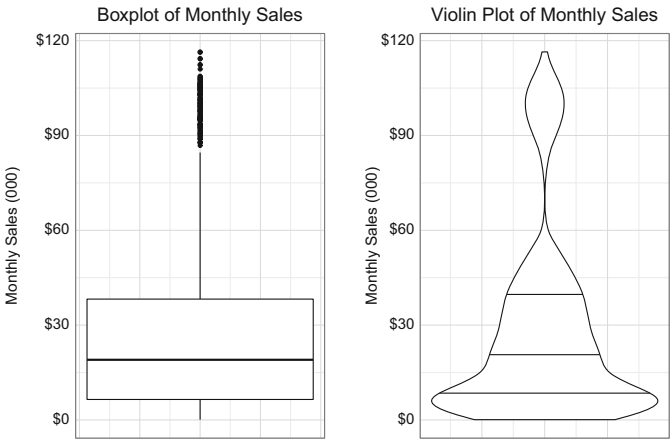

Fig. 5.4 Two different ways of visualizing the distribution of monthly sales numbers, the boxplot and the violin plot

has length $1.34 \times \sigma$. Thus, $1.5\times$ IQR $\approx 2\sigma$. The edge of the whiskers is placed at $\pm 2.67\sigma$ units from the mean so we would expect less than .8% of the data to fall outside the whiskers. Note that in our example, many points are plotted individually, indicating that the normal distribution is not an appropriate approximation of the monthly sales distribution.

The violin plot, by contrast, makes use of many more features of the data and can be seen as a boxplot replacement. The plot shows a smooth representation of the distribution turned on its side and gives us a more detailed visualization of the distribution. The width of the figure reflects the density of observations. With the addition of the horizontal lines at the first, second, and third quartiles, there is no loss of information compared to the boxplot. Moreover, the most interesting feature of this distribution, the lack of department-months with sales between $60K and $80K, is obscured by the boxplot but easy to see with the violin plot.

When we have univariate data that is labeled, we have already seen one of the best visualizations: dotcharts. These charts allow us to clearly depict single values and convey the uncertainty around those estimates (where appropriate). In Fig. 5.1 we saw how a dotchart was superior to a pie chart. In Fig. 5.5 we see another example of a dotchart, this time displaying summary statistics. This figure shows monthly spend by department with bars representing the range of values. Note that we have avoided the "Alabama First!" problem by sorting the departments from highest spend to lowest. In this depiction we can clearly see the two largest departments (produce and packaged grocery), the range of mid-sized departments (refrigerated grocery down to cheese), and the smaller departments. We have translated the x-axis variable to the $\log_{10}$ scale. This choice gives us a better view of the monthly sales for the small departments, but can make interpretation a bit trickier for the gray bars, which represent the range. At a first glance, it appears that bakehouse and flowers have by far the widest range in monthly sales. This is

true as a percentage; bakehouse ranges over almost an order of magnitude, from a minimum of $1000 to almost $10,000. By contrast, produce has a range of approximately $40,000 and a sample mean of nearly $100,000. An important note: the x-axis displays the values of monthly sales after transforming the values to the $\log_{10}$ scale. The labels, however, show the original, untransformed monthly sales in thousands of dollars. By retaining the labels in dollars, the reader is better able to interpret the variable. The best practice recommended by Cleveland is to place this axis at the bottom of the graphic and the corresponding log-based scale at the top. Unfortunately, `ggplot2` makes this difficult to do, so our labels show five values on the scale of thousands of dollars.

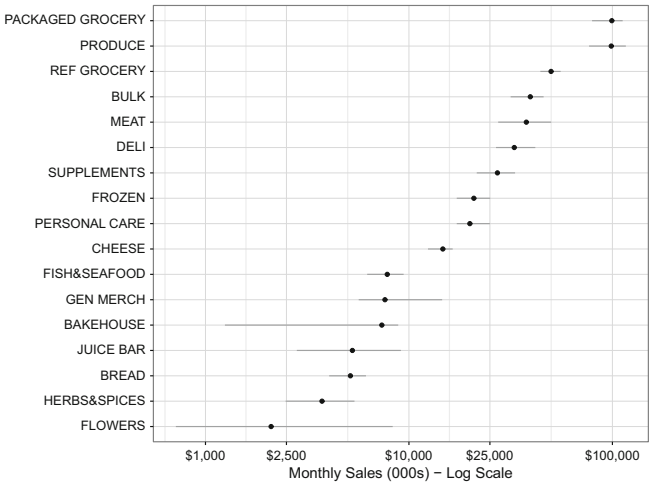

Fig. 5.5 A dotchart of spend by month by department, with bars indicating the range of the data. Monthly sales have been transformed to the $\log_{10}$ scale

The final type of data we might wish to visualize is univariate categorical data. If the number of categories is small, then a simple bar chart is an excellent choice to compare the proportions in each category. If the number of categories is very large, then summarizing with a dotchart as in Fig. 5.5 is often the best practice.

5.3.2 Bivariate and Multivariate Data

Bivariate data is usually straightforward to visualize. There are really three main possibilities: two categorical variables, a quantitative and categorical variable, and two quantitative variables.

Data consisting of all categorical variables typically are summarized by contingency tables and there is no reason to deviate from this practice. A contingency table is a cross-tabulation of observations according to the values of

the categorical variables. For instance, if there are a levels of variable A and b levels of variable B, then the table contains $a \times b$ cells and the content of a cell is the number of observations with a particular combination of levels. Table 5.1 is a contingency table cross-classifying receipts according to customer segment and department. We can see general trends by examining the table but specific and fine differences are not immediately discernible.

Table 5.1 Number of receipts cross-classified by department and the three largest customer segments, light, secondary, and primary

	Customer segment		
Department	Light	Secondary	Primary
Supplements	439	55,657	90,017
Cheese	859	96,987	147,679
Frozen	647	97,808	152,940
Meat	653	107,350	149,251
Deli	5830	138,722	155,086
Bulk	2713	144,862	178,520
Refrigerated grocery	3491	194,758	197,463
Produce	3971	211,226	199,978
Packaged grocery	6980	223,815	200,737

There exists, however, a useful visualization that rapidly conveys the relationships between categorical variables, the *mosaic plot*. Figure 5.6 shows a mosaic plot in which the area of the tiles represents the relative number of observations that fall into each cell of the contingency table. We can see, for instance, that there are many more primary shoppers than light shoppers. Packaged grocery is over-represented among light shoppers, whereas primary shoppers make up larger portions of the less popular departments.

This figure allows us to quickly absorb some of the features of the contingency table:

- Primary and secondary shoppers make up the majority of the observations.
- The four largest departments—produce, packaged grocery, refrigerated grocery, and bulk—represent about half the activity of the primary shoppers. Secondary shoppers used those departments to greater extent than primary shoppers and light shoppers used those departments to an even greater extent.
- Supplements represents a small fraction of the purchases of all segments.
- Primary shoppers tend to shop more of the store's departments, generally.

A virtue of the mosaic plot is that it allows estimation of the strength of the relationship between the categorical variables. A downside is that statistical features, such as confidence intervals for the difference in sizes, cannot be displayed. Moreover, the display becomes unwieldy beyond two dimensions. Figure 5.7 shows a mosaic plot built from data generated from two independent random variables with discrete uniform distributions. The plot shows no evidence of association. Conditioning on one variable shows approximately equal-sized rectangles as you travel across a row or down a column.

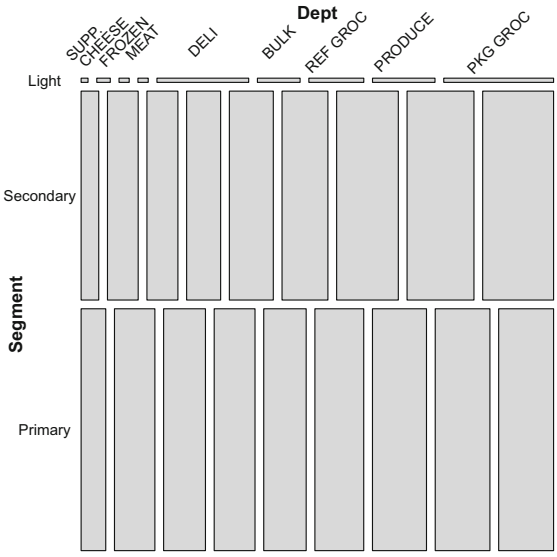

Fig. 5.6 A mosaic plot showing the relationship between customer segments and departments shopped

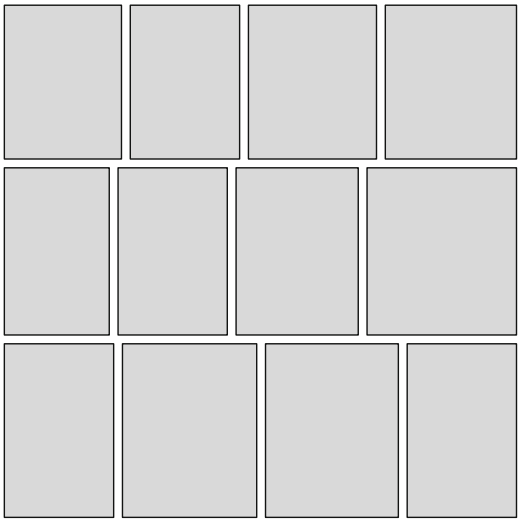

Fig. 5.7 A mosaic plot showing no relationship between two categorical variables. Reference plots like this are good to keep in mind when looking at mosaic plots

Most data that we need to visualize is not categorical, however. In the case of a quantitative variable and a categorical variable, we have already seen several good methods of showcasing relationships. Figure 5.5 shows several numeric results (the minimum, mean, and maximum spend by month) split by a categorical variable (the grocery store department). In Fig. 5.8 we see a great deal more information in a similar format. The previous chart showed spend at the department level. This chart shows spend at the individual shopper level for a sample of 10,000 shopper-months.

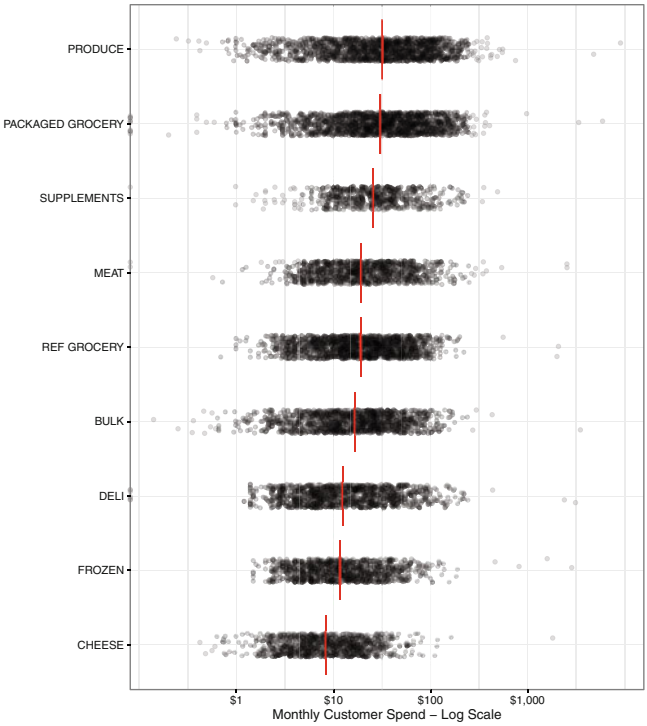

Fig. 5.8 A second example of a dotchart showing spending by department at the individual shopper level for 10,000 shoppers

Figure 5.8 displays spend on the $\log_{10}$ scale by individuals in a month across nine departments in the grocery store. The departments are ordered by the median monthly spend, shown as a vertical red line. The horizontal axis corresponds to a log scale to allow additional detail to be seen at the lower end of the scale. It also appears that the $\log_{10}$ scale reveals a much more symmetrical distribution—it is not uncommon for retail data to be approximately log-normal in distribution.

Several techniques we have not yet seen are illustrated in Fig. 5.8. The y-axis shows the levels of department. The values plotted at each level have been jittered so that more of them can be seen. Jittering adds a small random value, say between $-\varepsilon$ and ε, to the vertical coordinate of each plotted pair. Without the jittering all points would collapse onto their nearest horizontal grid line. We have also used transparency, set at the `ggplot2` value $\alpha = .1$. The interpretation of this value is that no fewer than $1/\alpha$ points plotted in the same location will appear completely opaque. Supplements are evidently shopped much less than, say, bulk, but the median spend is higher, presumably because each item is more expensive in this department.

With bivariate numerical data, the natural plotting technique is the scatterplot. Figure 5.9 illustrates this technique.

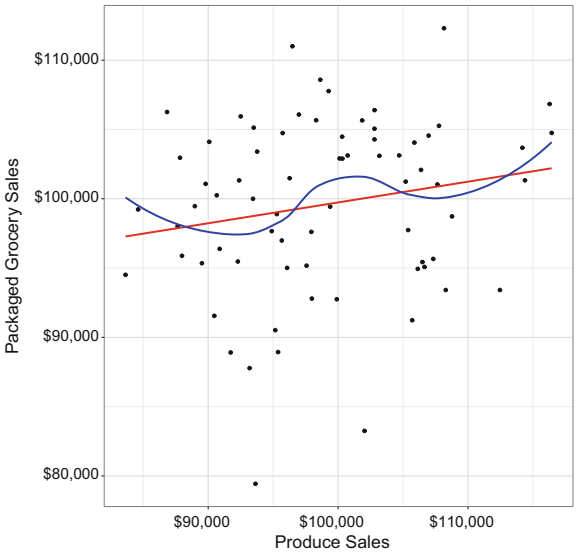

Fig. 5.9 A basic scatterplot, showing monthly sales for our two largest departments across 6 years. A linear regression line and loess smoother have been added to the plot to aid in interpretation of the relationship

Scatterplots are straightforward and adhering to the basic tenets laid out in Sect. 5.2 will keep a practitioner on the right track. The data speak for themselves, in this case illustrating the variability between produce and packaged grocery spend when viewed by month. Overall, the department sales move together, with a considerable amount of variation in the relationship.

When illustrating a relationship, one should strongly consider adding a fitted line to the graph to aid the viewer in decoding the relationship. In Fig. 5.9 we add two: a line built by linear regression and a curve created by locally weighted regression. The former needs little elaboration here, as linear regression is covered in Chap. 6. Locally-weighted regression, a technique introduced by Cleveland in 1979 and refined in 1988 [13], is a powerful technique for uncovering the relationship between two numerical variables. There are two competing terms, lowess and loess, and it seems the latter has become preeminent. They are closely related, both setting up a weighted neighborhood around an x value and fitting a regression line primarily influenced by points in vicinity of x. Much like our kernel smoothers discussed above, a bandwidth parameter, α, is specified. In each neighborhood a polynomial of degree d will be fit,[5] and we require a choice of $\alpha \in [\frac{d+1}{n}, 1]$. The default settings in `R` are $\alpha = .75$ and $d = 2$. For each subset of size $n\alpha$, a polynomial is fit. This fit is based on weighting the points and the typical weight function is the tricube weight. Let us assume that we have an observation x_i, a neighborhood around x_i denoted by $N(x_i)$, the width of which is r_i. Then the weight function is

[5] In practice d is almost always 1 or 2.

$$w_i(x) = \begin{cases} \left(1 - \left|\frac{x - x_i}{r_i}\right|^3\right)^3, & x \in N(x_i), \\ 0, & x \notin N(x_i). \end{cases}$$

The tricube kernel is nearly as effective as the Epanechnikov kernel. In practice, the choice of kernels is not nearly as important as the bandwidth choice.

We now turn our attention to multivariate data. Multivariate data is defined as data comprising at least three covariates per observation and, as mentioned earlier, most solutions require a thoughtful process. We can apply the general principles mentioned earlier and enjoy the profusion of options that are available. If there is one principal that stands above all others it is to avoid doing too much. This temptation is nearly irresistible. One may violate the rule profitably when a graphic will do the heavy lifting for several pages of text or when the graphic can be displayed on a slide for several minutes of explanation. This is not the norm and data scientists would be wise to split their story into multiple graphics if the audience is pressed for time.

Figure 5.10 is a reprisal of Fig. 5.8, made multivariate with the addition of the customer segmentation of the shopping data seen in Chap. 10. A number of interesting features emerge, notably the separation between secondary and primary shoppers for all departments except for supplements. The addition of median lines for each segment aids comprehension, particularly for the light shoppers, who make up small fraction of the shoppers. Interestingly, we can now see the compression of the deli spend—this department is one of the most popular with light shoppers.

Figure 5.10 is, essentially, a two-dimensional data visualization with a third dimension (segment), layered on top. Using two-dimensional plots with additional information to encode other variables is a common technique. Another variation, illustrated by Fig. 5.11 splits a two-dimensional plot into small multiples of pairs according to a third variable. The term "small multiples", coined by Tufte, captures the idea that readers, once they have been oriented to an individual plot, can quickly discern similarities and differences across a family of plots.

Each small panel, known as a "facet", is a scatterplot of spend versus items within a given department. A loess smoother is added to each panel. The advantages of faceting versus simply repeating scatterplots are several-fold: parsimonious code, an efficient use of space on the layout, the ability to order the facets in a sensible way, and common axes that facilitate comparison. With this treatment we can quickly identify interesting patterns within the data:

- The large volume of sales in produce, packaged grocery, and refrigerated grocery stand out relative to the other departments.
- Steeper curves indicate departments with cheaper items (produce, deli) while flatter curves show the expensive items (supplements, meat).
- Certain departments do not have large spends (cheese, frozen, and bulk).

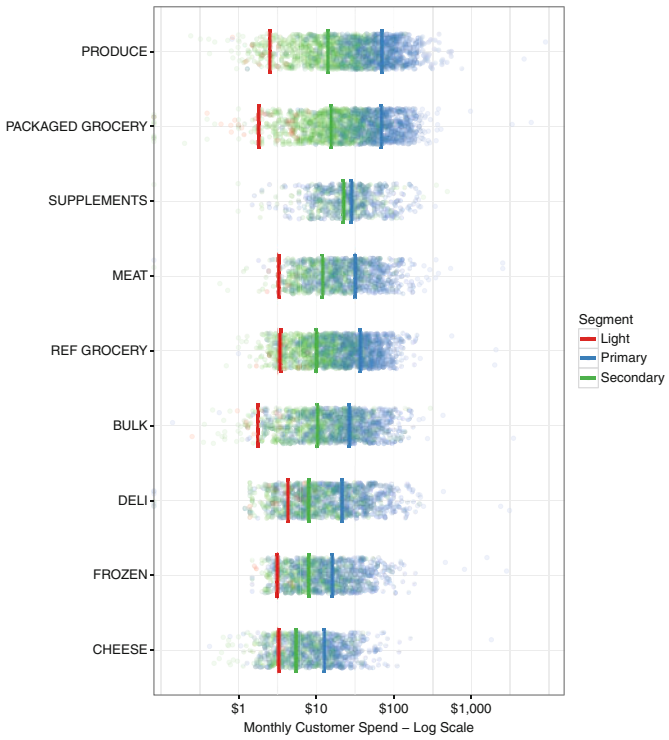

Fig. 5.10 An example of multivariate data. The spend-department dotchart is now colored based on the segments of the shoppers (either primary, secondary, or light). The medians are shown for each segment as a color-coded line

This section has served as a travelogue without a map. We have seen the destinations, but now we must learn how to get there. Building these plots requires a surprisingly small amount of code using the package that we will learn, `ggplot2`, partially because a great deal of complexity is hidden from us. Graphical software is a leaky abstraction, unfortunately, and so we will illustrate how to build up these charts from base elements.

5.4 Harnessing the Machine

The plots in this chapter were made with the `R` package `ggplot2`. This package is our strong recommendation for your personal data visualization tasks. There are three widely-used graphics packages associated with `R`: `base`, `lattice` [53], and `ggplot`. The `base` package is useful for quick plotting and for learning the basic techniques. It is possible to control many aspects of the plot with the `base` package but constructing publication-quality graphics is not easy. The second package, `lattice`, is based very closely on the ideas of Cleveland, but is not developed on the framework of a formal model. The `lattice` framework has limited its extensibility. The package `ggplot` is more flexible and easier to use.

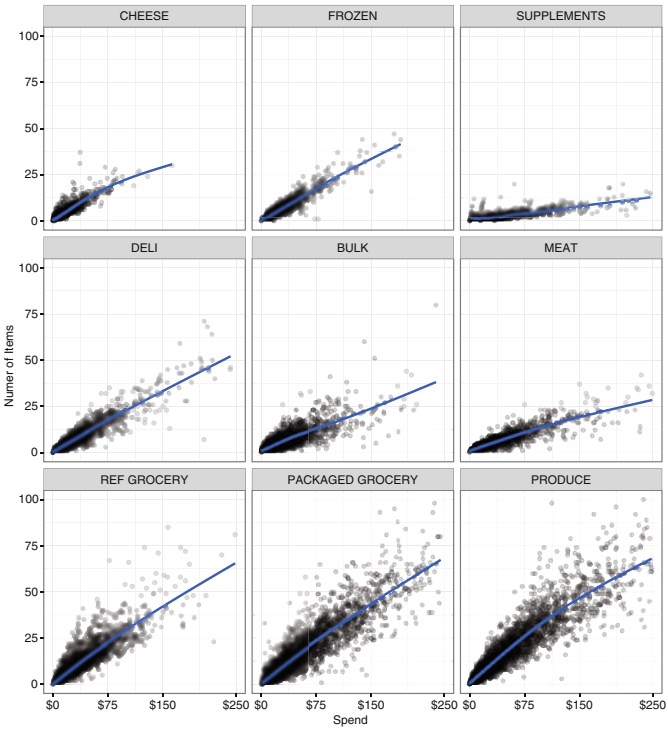

Fig. 5.11 A scatterplot, faceted by department, of spend versus items, with a loess smoother added to each panel

The package name, `ggplot2`, reflects both the version number, two, and the heritage, Wilkinson's Grammar of Graphics [69]. To quote Hadley Wickham, the principal author of `ggplot2`, "Wilkinson created the grammar of graphics to describe the deep features that underlie all statistical graphics." [66] Wickhams' books covers the grammar in some detail. Our treatment draws extensively from Wickham's texts.

The `ggplot` grammar requires work on the part of the reader. Once it is mastered, however, `ggplot2` provides the user with lots of flexibility and power. We encourage the reader to learn the grammar.

The grammar of graphics consists of a number of components that merge to form the grammar. They are

1. **data** The data to be rendered as a visual. Using `ggplot`, the data must be an `R` data frame. A `R` data frame is rectangular arrangement of the data with somewhat more specificity and overhead than a simple matrix. For example, a data frame may contain variables of several types, say, quantitative (numeric in the `R` lexicon) and categorical (a factor in the `R` lexicon).
2. **aes** *aes* is shorthand for *aesthetic mapping*. Aesthetic mappings tell `ggplot2` how to translate the data into graphic elements. The aesthetic mapping identifies the variables to be plotted on the x- and y-axes.

More may be added to the aesthetic mapping. For example, a categorical variable that determines the color of the points or lines is identified in the aesthetic mapping.

3. **geoms** *geoms* is shorthand for *geometric objects*. These are the elements that visually portray the data. Examples are points, lines, line segments, error bars, and polygons. All geometric object specifications take a form such as `geom_points()`.
4. **stats** These are statistical transformations that are used to reduce and summarize the data. The smoothers we saw above are examples of statistical transformations since the data was reduced in some manner to produce the smooth lines.
5. **scales** Scales provide the mapping between the raw data and the figure. Scales are also used to create legends and specialized axes.
6. **coord** The coordinate system takes us from the data to the plane of the graphic. This component is used, notably, to change axis scales from the original units of a variable to different units, say, logarithms.
7. **facet** As we saw in Fig. 5.11, facetting divides the graphic into subgraphics according to a categorical variable. This graphical component defines how the facets are arranged.

The process of building a visualization as a series of *layers* in `ggplot` is a remarkable improvement on the traditional process. Figures in `ggplot` are built by first creating a base consisting of a rudimentary plot and then adding more information and data to the base in the form of layers. Each layer may contain specific information from the attribute list above. Usually, layers inherit most of the attribute values from the initially created plot. The specification of the layer may be very simple since we only need to change a few items in each layer. Consequently, layers make the task of building a complex graphic much easier. We tend to build graphics one layer at a time. We can see the effect of each layer on the graphic and more easily correct coding errors. A highly readable tutorial on the subject is available at https://rpubs.com/hadley/ggplot2-layers.

Our intent in this chapter is not to systematically review `ggplot2` techniques but to provide a general understanding of how visualizations are constructed in `ggplot2`. In the remainder of the chapter, we explain how the graphics discussed above were built. To go further, utilize internet resources and, in particular, Stack Overflow (`https://www.stackoverflow.com`). If you can't solve a problem or remember an instruction, then search the internet for information. Adding *ggplot* to the search string makes one much more likely to find results related to `R` and to the `ggplot2` plotting package in particular. Using the vocabulary of the package in the search string is important to efficient searching.

We now turn to the code that created this chapter's figures. We first start by reading in the data and loading the necessary libraries.

```
library(ggplot2)
library(scales)
library(RColorBrewer)
```

The first two libraries, `ggplot2` and `scales` are directly related to plotting. The final one, `RColorBrewer`, is based on the pioneering work on color and perception of Brewer et al. [7], is indispensable and we recommend it highly.

5.4.1 Building Fig. 5.2

Figure 5.2 is built from a summary table named `month.summary`. This data frame holds the sum of monthly sales for four departments at the co-op. The first five rows are displayed in Table 5.2.

Table 5.2 The first few rows of the data frame `month.summary`

Row	Month	Department	Sales
1	6	Packaged grocery	90,516.72
2	6	Produce	103,650.8
3	6	Bulk	37,105.20
4	6	Meat	35,221.97
5	5	Packaged grocery	102,178.30

The code follows.

```
month.summary <- read.delim("../data/month_summary.txt")

ggplot(data=month.summary,
       aes(x=factor(month), y=sales/1000, group=Dept, col=Dept) )
  + scale_color_brewer(palette="Set1")
  + geom_point()
  + geom_line()
  + theme_bw()
  + scale_y_continuous(label=dollar)
  + ylab("Avg Annual Sales in Month (000s)")
  + xlab("Month")
  + theme(legend.key.size=unit(0.5, "cm") )
```

After reading in the data, we build the plot in `ggplot2`, layer by layer.

1. **data** We use the data shown in Table 5.2.
2. **aes** We assign month to the x-axis and sales-divided-by-one-thousand to the y-axis. Department is used as a grouping variable. The consequence of

setting `group=Dept` is that points belonging to the same department will be joined by lines. Departments will identified by color. When building graphics that use grouping and color, it is common to forget to include the grouping variable in the aesthetic. As an exercise, we encourage the reader to build the plot *without* declaring the grouping variable in `aes`. The appearance of the resulting plot, with the telltale diagonal lines, are the mark of a missing grouping variable.

3. **geoms** There are two geometric objects used in the figure: points and lines. They are added as separate layers and inherit the grouping and color from the aesthetic.
4. **scales** We use a continuous scale and, for the axis labeling, take advantage of the library `scales` to reduce the effort of the reader to understand what has been plotted. The units are dollars so we pass the keyword `dollars` into the `label` argument. This parameter will ensure that the axis labels are formatted with dollar signs, commas, and cents. Other options include `percent` and `comma`, all of which promote readability. We also use the colors provided by `RColorBrewer` for the same reason.
5. **theme** Although not part of our original list, themes help you adjust the appearance of your plots. We invoke the simple and clean `theme_bw`, label the axes, and shrink boxes in the legend a bit. These features are almost always included in our graphics.

5.4.2 Building Fig. 5.3

Our next code segment builds our histogram and empirical density function.

```
# Read in grocery store summary data.
working.dir <- "../data/"
gd <- read.delim(paste0(working.dir, "grocery_data.txt"))
ggplot(gd, aes(x = ownerSales/1000))
    + geom_histogram(aes(y=..density..))
    + geom_line(stat="density",col="gray50",size=1.5,
              bw="SJ") + # Changing the bandwidth algorithm to 'SJ'
    + theme_bw()
    + scale_x_continuous(label=dollar)
    + ylab("Density")
    +  xlab("Monthly Sales (000)")
```

The key features used to construct the figure are as follows.

1. **data** Histograms summarize distributions as they are built. We don't have to carry out data reduction before building the figure. It's done by `ggplot` in the construction of the histogram.

2. **aes** Histograms are defined by a single variable which we assign to the x-axis. `ggplot2` determines the y-values using our guidance.
3. **geoms** The first geometric object is a histogram. The `aes` argument sets the scale for the y-axis. Our choice of units for the histogram are proportions. The y-axis will show the proportion of observations in each interval and it's specified by setting `y=..density...` The syntax of `..density..` looks odd. The pattern of periods (two at the beginning and two at the end) indicates a statistical computation is necessary. The other commonly chosen option shows the counts of observations in each interval. Entering `?stat_bin` from the console will provide more information.

   A second `geom`, `line` is also specified. The `stat` argument specifies that the line to be drawn is the graph of a empirical density function (Sect. 5.3.1). Several other attributes are specified: the color, the line width (by adjusting `size`), and that the method of computing the bandwidth is to be Sheather and Jones' method [54].

5.4.3 Building Fig. 5.4

The code for the violin plot from Fig. 5.4 is next.

```
ggplot(gd, aes(x=1, y=ownerSales/1000))
    + geom_violin(draw_quantiles = c(0.25,0.5,0.75))
    + theme_bw()
    + scale_y_continuous(label=dollar)
    + xlab("")
    + ylab("Monthly Sales (000)")
    + theme(axis.text.x=element_blank(),axis.ticks=element_blank())
    + labs(title="Violin Plot of Monthly Sales")
```

A couple of new techniques emerge.

1. **aes** The x-axis is set to be a constant. The result is that the violin plot will be drawn in the center of the graphic.
2. **geoms** The `geom_violinplot` accepts an argument specifying that quantiles are to be drawn. We specify that these are to be the first, second, and third *quartiles*.
3. **theme** Since the x-axis does not have any meaning, we would like to avoid showing ticks marks and labels. The arguments to `theme` eliminate those features. A internet search for something similar to "ggplot blank x axis" will provide details.

5.4.4 Building Fig. 5.5

The dotchart in Fig. 5.5 requires a prepared data frame `dept.summary` containing the sample minimum, mean, and maximum sales the 17 departments of the grocery store co-op. The first five rows are shown in Table 5.3.

Table 5.3 The first five rows of the `dept.summary` data.frame

Row	Department	Sample Minimum	Sample Mean	Sample Maximum
1	Packaged grocery	79,434.24	99,348.55	112,303.60
2	Produce	76,799.20	98,711.50	116,442.92
3	Bulk	31,610.05	39,482.60	45,908.04
4	Refrigerated grocery	44,239.77	49,940.93	55,656.07
5	Cheese	12,407.99	14,674.32	16,404.14

After reading the summary data into a data frame, it is re-ordered by the sample mean.

```
dept.summary <- read.delim("../data/dept_summary.txt")
dept.summary$dept <- reorder(dept.summary$dept, dept.summary$mean.val)

ggplot(dept.summary, aes(x=mean.val, y=dept))
    + theme_bw()
    + geom_errorbarh(aes(x=mean.val, xmax=max.val, xmin=min.val, y=dept),
                     height=0, color="gray60")
    + geom_point(col="black")
    + ylab("")
    + xlab("Monthly Sales (000s) - Log Scale")
    + scale_x_continuous(label=dollar,trans="log10",
                      breaks=c(1000,2500,10000,25000,100000))
```

1. **aes** When a factor is used as an x- or y-variable in an aesthetic, the factor levels are mapped to sequential integers, say $1, 2, \ldots, g$, where g is the number of levels. Knowing that the levels occur at integer positions on the x-axis will be important when we add features like jittering.
2. **geoms** Another new `geom`, `geom_errorbarh`, is used. The "h" stands for horizontal—the vertical variety needs no suffix. The aesthetic mapping for `geom_errorbarh` requires us to supply x- or y-variables and the starting and ending positions for the error bars. The parameter `height` specifies the width of the whiskers on the bars. We've suppressed the whiskers as they add no information. Note that points are desired at the median and receive their own `geom`.
3. **scale_x_continuous** We repeat our label trick and instruct `ggplot` to transform the variable plotted on x-axis according to the transformation $x \to \log_{10}(x)$. Also, we set the label positions using the `breaks` argument.

5.4.5 Building Fig. 5.8

We build the plots for Figs. 5.8 and 5.10 in the next series of code segments. We begin using `sample.int` to draw a random sample of manageable size from the shopper data set. Setting the seed of the random number generator with the instruction `set.seed` ensures that our results are the same on repeated runs. We also order our departments based on the median total sales by department.

```
set.seed(3939394)
# Read in grocery store detail data.
gdd <- read.delim(paste0(working.dir,"shopper_dept_segment.txt"))
this.data <- gdd[sample.int(nrow(gdd),size=10000,replace=F),]
this.data$DepartmentName <- reorder(this.data$DepartmentName,
                                    this.data$TotalSales,
                                    FUN=median)
```

Drawing the vertical lines that depict the medians requires those values to be calculated and it is clearest if we assign them to a data frame of their own. We call that data frame `medians`.

```
medians <- aggregate(this.data$TotalSales,
                     list(this.data$DepartmentName),
                     FUN=median)
names(medians) <- c("DepartmentName","TotalSales")
medians$y.val <- as.numeric(medians$DepartmentName)
```

We are now ready to build Fig. 5.8.

```
ggplot(this.data, aes(x = TotalSales, y = DepartmentName))
  + theme_bw()
  + geom_point(position=position_jitter(h = .4),
               alpha=0.1)
  + ylab("")
  + xlab("Monthly Customer Spend - Log Scale")
  + scale_x_continuous(label=dollar, trans="log10",
                       breaks=c(1,10,100,1000))
  + geom_segment(data = medians,
               aes(x = TotalSales, xend = TotalSales,
                   y = y.val -.4, yend = y.val+ .4),
               col="red")
```

1. **data** Two data sets are used to construct the figure. The principal data set is the sample of individual shopper grocery data. A second data set called `medians` is used for the vertical lines and is passed directly into the necessary `geom_segment`.

2. **aes** This plot uses an aesthetic that is similar to what we have seen before, with the categorical variable `departmentName` assigned to the y-axis. Individual shopper sales are assigned to the x-axis. There is a separate set of aesthetic mappings used with `geom_segment`.
3. **geoms** There are two `geom`s used in this plot, one for the points and one for the line segments. The `geom_point` has been used before, but now we show how to add jittering to a plot. Recall that jittering perturbs points so that there is less over-plotting and individual points are easier to see. The command, `position=position_jitter(h=.4)` instructs `ggplot2` to place the points at the original x-axis coordinate and to perturb the y-axis coordinate. The argument $h = .4$ controls the magnitude of the random perturbations.

   The other `geom` layer, `geom_segment` uses the second data set. We specify the starting and ending line segment coordinates on the x- and y-axes.
4. **scale** Once again we use the *dollar* labeling from the library `scales`. As we have been doing in other plots, we instruct `ggplot` to transform the variable plotted on x-axis according to the mapping $x \to \log_{10}(x)$ and specify where we would like the labels to appear.

5.4.6 Building Fig. 5.10

The code used to construct Fig. 5.10 is a relatively simple modification of the code used to construct Fig. 5.8. The key difference is that we wish to color the points according to segment, as well as have separate median lines for the segments. First, we must calculate the medians for each department and segment.

```
medians <- aggregate(this.data$TotalSales,
                     list(Segment=this.data$Segment,
                          DepartmentName=this.data$DepartmentName),
                     FUN=median)
names(medians)[3] <- "TotalSales"
jit.val <- 0.6 # More jittering with segments
```

The function `aggregate` is convenient for summarizing one variable based on one or more other variables. We rename the column that holds the calculated medians to match the aesthetic we'll use in our plot.

As we shall see, it is quick to add a grouping variable that allows coloring of the points and the technique is analogous to what we did in Fig. 5.2. There we used grouping and color aesthetics to ensure that the department sales by month were joined by a line and common color. Here the variable `Segment` will fill that role.

```
ggplot(this.data,
      aes(x=TotalSales, y=DepartmentName, group=Segment, col=Segment))
  + theme_bw()
  + geom_point(position=position_jitter(h=jit.val),
            alpha=0.1)
  + ylab("")
  + scale_color_brewer(palette = "Set1")
  + xlab("Monthly Customer Spend - Log Scale")
  + scale_x_continuous(label=dollar,trans="log10", breaks=c(1,10,100,1000))
  + geom_segment(data = medians,
      aes(x = TotalSales,xend=TotalSales, group=Segment, col=Segment,
          y = y.val-0.5*jit.val, yend=y.val+0.5*jit.val), size=1.25)
```

What are the differences between this code and the previous code? Just the addition of the argument `group=Segment` and `col=Segment` in the `ggplot` aesthetic and `geom_segment` aesthetics, respectively. Other aspects of our plots remain the same.

5.4.7 Building Fig. 5.11

Our final example builds Fig. 5.11. The code is reproduced here, though there are few features we have not seen before.

```
set.seed(3939394)
this.data <- gdd[sample.int(nrow(gdd),size=25000,replace=F),]
this.data$DepartmentName <- reorder(this.data$DepartmentName,
                    this.data$TotalSales,
                    FUN=sum)
ggplot(this.data, aes( x = TotalSales, y = Items))
    + geom_point(alpha=0.1)
    + stat_smooth(se=F)
    + facet_wrap(~DepartmentName)
    + scale_x_continuous(label=dollar, limits=c(0,250),
          breaks=c(0,75,150,250))
    + scale_y_continuous(limits=c(0,100),label=comma)
    + theme_bw()
    + xlab("Spend")
    + ylab("Number of Items")
```

The first new feature is a statistic that stands on its own (as opposed to being hidden inside `geom_line` as we saw above). The function, `stat_smooth`, is quite useful, adding a line to a plot according to the specified method. In the code that generated Fig. 5.9, we added two smooths in two layers. One layer added the default smoothing method, loess, and the second added a fit-

ted linear regression line using the instruction `stat_smooth(method="lm")`. The default is to include standard error envelopes. We have suppressed the envelopes by passing the argument `se=F`.

The other new feature is `facet_wrap`, the function that controls the creation of the panels. The *wrap* version creates a rectangular array of panels. One panel is created for each level of the factor `DepartmentName`. Only observations associated with a particular level of the factor are graphed in the panel. Rows and columns can be controlled. Multiple variables can be set on the right-hand side of the tilde ($\sim$) to partition the data by combinations of levels of the variables. The analog of `facet_wrap` is `facet_grid`, which is designed to set up the panels in a tabular array, specifying row factors and column factors using the same tilde operator. One important parameter, not changed in this code, is the `scales` value, which can be set to `free`, `free_x`, `free_y`, and `fixed`. The default is the last one and it forces a single set of axes for every panel. The `free` varieties allow the axes to differ and generates the best visual fit for each panel. The `free` option is often a mistake if the purpose of the panels is to compare and contrast groups because the reader will have to account for different scales in different panels. The best visualization depends on the goals of the data scientist.

A final word of caution and encouragement. Plotting with `ggplot2` imposes an initially steep learning curve. The payoff is well worth the effort—do not underestimate the importance of communicating clearly and efficiently. Visualization is the best way to do it. The key to learning `ggplot` is to start simple and add complexity one layer at a time. Build a simple scatterplot with `geom_point`, then add a layer that identifies groups by color. Add a layer showing smooths. Label the axes clearly and adjust the font sizes. Every successful plot that we created for the first 6 months of working with `ggplot2` was made by incremental steps.

5.5 Exercises

5.1. Use the data set `grocery_data.txt` containing sales by month and build a dotchart of monthly sales by department. Put department on the y-axis and sales (in thousands) on the x-axis. Include the following features:

a. Sort the departments based on average sales.
b. Jitter the points vertically and set `alpha` to help with over-plotting.
c. Label the x-axis using dollars.
d. Suppress the y-axis label and use a sensible label for the x-axis.

What patterns emerge from the departments? How does the spread of sales vary by department? Why might this be?

5.2. Add vertical red lines representing the median monthly sales by department to the figure of Problem 5.1.

5.3. Build the middle panel from Fig. 5.3 with the bandwidth parameter set to the values 1, 5, and 25. What is the bandwidth returned by the "SJ" method? How does it compare to the `bw.nrd0` method[6]?

5.4. Continuing to use the data set `grocery_data.txt`, build a faceted graphic showing the empirical density of sales per month by department. Allow the x- and y-axes to vary within the panels. Use the SheatherJones bandwidth selection algorithm. Compare the densities.

5.5. Now we fine-tune the solution to Problem 5.4 to make it publication worthy. Add the following features:

a. Angle the x-axis labels to 45°. (Hint: `?theme` and search for `axis.text.x`.)
b. Use the black and white theme.
c. Remove the beer & wine segment from the plot since there are very few observations in the segment.
d. Shrink the font size of the facet titles (called `strip.text`) to make the longer names fit.

5.6. Repeat Problem 5.1 with violin plots. Add quartile lines and draw a red vertical line at the overall average monthly sales across all months and all departments. Use the parameter `scale="width"` in the violin plot to avoid the default behavior of the width being proportional to sample size.

Does this view of the data add to any insights beyond what you learned from Problem 5.1?

5.7. Build the mosaic plot shown in Fig. 5.6. This plot is *not* made by ggplot, but is made with the `mosaic` function in the package `vcd`. Controlling the orientation of the axis labels is tricky; we recommend replacing the department and segment names with abbreviations.

5.8. Use the `gdd` data to build the sum of sales by year and month, and segment. Display these in a line chart in chronological order, with each segment being its own line.

5.9. Use the `gdd` data to build the sum of sales by year and month and segment. Plot sum of sales against year and month, with each segment shown by a separate line. Include the following features in your plot:

a. Vertically align the year-month labels.
b. Plot the y-axis on a log scale and choose sensible breakpoints. Label the axis with the "dollar" formatting.
c. Label the x-axis every 12 months starting at 2010-1.
d. Add a linear trend line, by segment, to the plot. What information can you infer about the light segment from this graph?

[6] The purpose of these last two questions is to learn how to extract the bandwidth information from `R` directly.

Chapter 8
Cluster Analysis

Abstract Sometimes it's possible to divide a collection of observations into distinct subgroups based on nothing more than the observation attributes. If this can be done, then understanding the population or process generating the observations becomes easier. The intent of cluster analysis is to carry out a division of a data set into *clusters* of observations that are more alike within cluster than between clusters. Clusters are formed either by aggregating observations or dividing a single glob of observations into a collection of smaller sets. The process of cluster formation involves two varieties of algorithms. The first shuffles observations between a fixed number of clusters to maximize within-cluster similarity. The second process begins with singleton clusters and recursively merges the clusters. Alternatively, we may begin with one cluster and recursively split off new clusters. In this chapter, we discuss two popular cluster analysis algorithms (and representatives of the two varieties of algorithms): the k-means algorithm and hierarchical agglomerative clustering.

8.1 Introduction

Cluster analysis is a collection of methods for the task of forming groups where none exist. For example, we may ask whether there are distinct types of visitors to a grocery store, say, customers that purchase a few items infrequently, customers that regularly shop at a particular department, and customers that make frequent visits and purchase a wide variety of items. If so, then the visitors of the first and second group might be offered incentives aimed at shifting them to the third group. Realistically though, it's probably not possible to cleanly divide customers given that many customers

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0_8

exhibit multiple shopping behaviors. With a set of observations on customers and records of their purchases (receipts), cluster analysis may provide insight into the structure of the customer population and shopping behaviors.[1]

In this situation, the analyst does not have a set of data that is labeled according to group. The process of forming groups is therefore without benefit of supervision originating from a training set of labeled observations. Without knowledge of the number and arrangement of the groups, cluster formation proceeds by grouping observations that are most alike or by splitting groups based on the dissimilarity of the member observations. Similarity of observations and clusters is measured on the observation attributes. The only impetus driving cluster formation is similarity and dissimilarity.

Cluster analysis is driven by the mathematical objective of maximizing the similarity of observations within cluster. This objective does not provide much guidance on how to proceed. In contrast, linear regression is strongly driven by the linear model and the objective of minimizing the sum of squared differences between observations and fitted values. The mathematics and algorithms fall neatly into place from this starting point. Clustering algorithms on the other hand, have different approaches, all meritorious and occasionally useful. Even determining the appropriate number of clusters is difficult without prior knowledge of the population sub-divisions. Despite these weaknesses, cluster analysis is still a useful tool of data analytics since the ability to examine groups of similar observations often sheds light on the population or process generating the data.

In this chapter, we sidestep the difficult issues related to the application and interpretation of cluster analysis and focus instead on the mechanics of two somewhat different but fundamental algorithms of cluster analysis: hierarchical agglomerative and k-means clustering. By understanding these basic algorithms, we learn of the strengths and weaknesses of cluster analysis.

Let us begin by defining $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ to be the data set. As before, $\mathbf{x}_i$ is a vector of attributes measured on an observational unit. Unlike the data sets of Chap. 6, there is no target y_i to be predicted.

We begin the discussion of cluster analysis with a popular technique that recursively builds clusters by merging smaller clusters.

8.2 Hierarchical Agglomerative Clustering

The hierarchical agglomerative approach begins with each observation defining a singleton cluster. Therefore, the initial set of clusters may be represented by the singleton clusters $\{\mathbf{x}_1\}, \ldots, \{\mathbf{x}_n\}$, where n is the number of observations. The algorithm iteratively reduces the set of clusters by merging

[1] Section 10.6, Chap. 10 works with data originating from grocery store receipts.

similar clusters. On the ith iteration, two clusters A and B, say, are merged to form a cluster $A \cup B$. We write

$$(A, B) \longrightarrow A \cup B$$

to describe the merging of clusters A and B. The choice of clusters to merge is determined by computing a distance between clusters and merging the pair with the minimum inter-cluster distance. Consequently, a metric is needed to measure between-cluster distances. For example, the distance between clusters A and B may be defined to be the smallest distance between any vector belonging A and any vector belonging to B. Mathematically, this distance is defined as

$$d_1(A, B) = \min\{d_C(\mathbf{x}_k, \mathbf{x}_l) | \mathbf{x}_k \in A, \mathbf{x}_l \in B\},$$

where $d_C(\mathbf{x}, \mathbf{y})$ is the city-block distance between vectors $\mathbf{x}$ and $\mathbf{y}$ defined by Eq. (7.12). There's nothing special about city-block distance—other metrics, Euclidean distance, for example, also are popular. The minimum distance metric d_1 tends to produce chain-like clusters. More compact clusters result from a centroid-based metric that utilizes cluster centroids defined by

$$\overline{\mathbf{x}}_A = n_A^{-1} \sum_{\mathbf{x}_k \in A} \mathbf{x}_k, \tag{8.1}$$

where n_A is the numbers of observations in cluster A. Then, the distance between A and B is

$$d_{\text{ave}}(A, B) = d_C(\overline{\mathbf{x}}_A, \overline{\mathbf{x}}_B). \tag{8.2}$$

As described above, the algorithm will progressively merge clusters until there is a single cluster. Merging clusters into a single glob is only interesting if some of the intermediate cluster sets are interesting. If the clusters are to be used for some purpose, then it's necessary to inspect the intermediate sets of clusters to identify the most useful cluster set.

8.3 Comparison of States

We return to the BRFSS data and search for states that are alike with respect to the distribution of body mass index of residents. The distribution of body mass index for a particular state is represented by a histogram. A histogram, mathematically, is a set of intervals spanning the range of body mass index and the associated sample proportions of individuals belonging to each interval.[2] Each histogram amounts to an *empirical* distribution—that is, a distribution that has been constructed from data rather than a model.

[2] We worked with the mathematical form of the histogram in Chap. 3, Sect. 3.4.2.

As usual, data preparation requires a significant effort since we need to construct a histogram for each state, Puerto Rico, and the District of Columbia. Before delving into the computational problem, let us examine the empirical body mass index distributions for a handful of states.

Figure 8.1 shows the empirical distributions of body mass index for five states: Massachusetts, Colorado, Illinois, North Dakota, and Maryland.[3] The distributions from Massachusetts and Colorado are alike as are the distributions from Illinois, North Dakota, and Maryland. Massachusetts and Colorado are different from Illinois, North Dakota, and Maryland, principally because the Massachusetts and Colorado distributions tend to have relatively more values that are less than 27 kg/m^2 than Illinois, North Dakota, and Maryland, and fewer values that are greater than 27 kg/m^2. Interestingly, Colorado and Massachusetts previously were found to have the second and tenth smallest estimates of diabetes prevalence (Fig. 7.1).

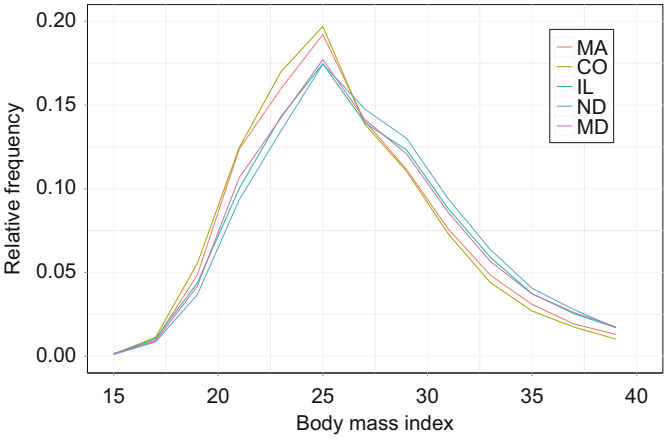

Fig. 8.1 Empirical body mass index distributions for five states. The scale has been truncated on the right so that differences among distributions may be more easily discerned

Recall that the numerical form of a conventional histogram is a set of pairs $H = \{(b_1, p_1), \ldots, (b_h, p_h)\}$, where $b_i = (l_i, u_i]$ is a bin or interval and p_i is the proportion of observations included in the interval.[4] If the data are a representative sample from the population, then H is constructed by defining a set of intervals and counting the number of sample observations falling into each interval. The BRFSS sampling design does not generate representative samples since observations are collected with unequal sampling probabilities. To correct for unequal sampling probabilities, the Centers of Disease Control and Prevention has attached a sampling weight to each observation. Usually, the sampling weight can be incorporated into an estimator with the effect of reducing or eliminating bias created by unequal sampling probabilities.

[3] The data shown in this figure may be plotted as a set of histograms. However, we use a simple line plot instead as it's easier to see the similarities among empirical distributions.

[4] Section 3.4.2 of Chap. 3 discusses histograms in details.

Section 3.4.2 of Chap. 3 presented an algorithm for constructing histograms from sampling weights. The adaption replaced the relative frequency of observations included in a particular interval with the sum of sampling weights associated with the observations.

Let x_j denote a measurement on the variable of interest (in this case, body mass index) for the jth observation (in this case, a respondent). Let w_j denote the sampling weight assigned to the observation. Let's suppose that $x_j, j = 1, 2, \ldots, n$ are from state A. Then, the total sampling weight associated with values belonging to interval b_i and originating from state A can be expressed as

$$s_{A,i} = \sum_{j=1}^{n} w_j I_i(x_j) \tag{8.3}$$

where $I_i(x_j)$ is an indicator variable taking on the value 1 if $x_j \in b_i$ is true (formula (3.10)).[5] If $x_j \in b_i$ is false, then the value of the indicator variable is 0. Transforming the sum $s_{A,i}$ to the height of the histogram bar is straightforward: we compute the estimated proportion of the population belonging to interval b_i as

$$p_{A,i} = \frac{s_{A,i}}{\sum_{k=1}^{h} s_{A,k}}. \tag{8.4}$$

Formula (8.4) was encountered in Chap. 3, formula (3.12). As with a conventional histogram, we iterate over the observations and accumulate the sums $s_{A,1}, \ldots, s_{A,h}$, and then form the histogram for state A as

$$H_A = \{(b_1, s_{A,1}), \ldots, (b_h, s_{A,h})\}, \tag{8.5}$$

A dictionary is maintained to keep track of the clusters as the algorithm progresses. A dictionary key is the cluster label A, and the value is the list of sampling weights $[s_{A,1}, \ldots, s_{A,h}]$. It's not necessary to store the intervals (the b_i's) with each histogram since the intervals are the same for every histogram.

When the cluster formation algorithm begins, the cluster singletons are individual states and the cluster histograms are the histograms for each state. The distance between clusters A and B is measured by the distance between the histograms associated with the clusters. This distance is defined to be

$$d_c(A, B) = \sum_{k=1}^{h} |p_{A,k} - p_{B,k}|,$$

where $p_{A,k}$ and $p_{B,k}$ are computed according to Eq. (8.4).

We also need a method for merging the histograms resulting from the merger of clusters A and B. The estimated proportion for interval k should not be a simple average of the interval proportions $p_{A,k}$ and $p_{B,k}$ since the average does not account for the differences in numbers of observations that are contained in clusters A and B. So instead, we compute a weighted average

[5] The statement $x_j \in b_i$ is true if $l_i < x_j \leq u_i$.

that utilizes the numbers of observations n_A and n_B in the respective clusters. The weighted mean is

$$\overline{p}_k = \frac{n_A p_{A,k} + n_B p_{B,k}}{n_A + n_B}.$$

The weights are the proportions $n_A/(n_A + n_B)$ and $n_B/(n_A + n_B)$.

In summary, the hierarchical clustering algorithm can be viewed as a sequence of mappings where a mapping reduces two clusters A and B to one cluster according to

$$\left.\begin{matrix}[p_{A,1}, \ldots, p_{A,h}] \\ [p_{B,1}, \ldots, p_{B,h}]\end{matrix}\right\} \longrightarrow [\overline{p}_1, \ldots, \overline{p}_h]\,. \tag{8.6}$$

We may also view the mapping as $(H_A, H_B) \longrightarrow H_A$. Instead of creating a new label $A \cup B$, we assign the label A to $A \cup B$ and delete B from the list of cluster labels. Metaphorically, A has devoured B.

Operationally, merging two clusters requires a search for the most similar pair of clusters. Once the pair (A, B) is identified, cluster A absorbs B by combining the two histograms as one according to the map (8.6). The combined histogram, replaces A. Cluster B is removed from the list of clusters.

The search for most similar clusters requires the inter-cluster distance between every cluster A and B. If the number of observations were large, we would not recompute all inter-cluster distances for every search, but instead maintain a list of inter-cluster distances and only update the distances between the merged cluster and all other clusters. For simplicity, the tutorial below recomputes all inter-cluster distances whenever a pair of clusters is merged.

8.4 Tutorial: Hierarchical Clustering of States

We'll write an algorithm for hierarchical agglomerative cluster formation. A substantial amount of data reduction is necessary before cluster formation can commence. The data reduction stage will map a set of BRFSS annual files to a dictionary in which each key is a state and the value is a list of the sampling weights $s_{A,1}, \ldots, s_{A,h}$ shown in Eq. (8.3). Most of the data reduction has been carried out in one form or another in previous tutorials. Each state in the dictionary will represent one of the initial singleton clusters. The cluster formation algorithm begins with this set of initial clusters. The algorithm iterates over a list of clusters and on each iteration, two clusters are merged as one. When two clusters are merged, the dictionary is updated by replacing one cluster with the merged cluster. The second cluster is removed from the dictionary.

1. At the top of your script, import modules to be used by the program:

```
import os
import importlib
import sys
sys.path.append('/home/.../parent')
from PythonScripts import functions
dir(functions)
```

2. Use the function `stateCodeBuild` developed in instruction 4 of Sect. 7.3, Chap. 7 to create a dictionary of state names and codes. We use the term state loosely as the dictionary will contain entries for the District of Columbia and Puerto Rico. The dictionary `stateCodeDict` uses the FIPS two-digit state codes as keys. The values are standard United States Postal Service two-letter state abbreviations. Call the function and create the dictionary. Also create a list (`namesList`) containing the two-digit abbreviations:

```
stateCodeDict = stateCodeBuild()
namesList = list(stateCodeDict.values())
noDataSet = set(namesList)
```

   The Behavioral Risk Factor Surveillance System collects data from three U.S. territories as well as the states and the District of Columbia. However, `stateCodeDict` contains the names of 57 states, territories, and the District of Columbia. Not all of these geographic units are sampled by the BRFSS survey. We'll identify the geographic units that have no data by removing names from the set `noDataSet` whenever a record is encountered from a particular geographic unit. What's left in `noDataSet` are geographic units with no data.
3. Create a dictionary containing the field positions of body mass index and sampling weight using the function `fieldDictBuild`. It was created in instruction 4 of Sect. 7.3, Chap. 7.

```
fieldDict = functions.fieldDictBuild()
print(fieldDict)
```

4. Create a dictionary of state histograms. Each histogram consists of a set of 30 sub-intervals spanning the interval $(12, 72]$ (kg/m^2) and an associated set of relative frequency measures. The histogram dictionary `histDict` maintains the histogram data for cluster A as a list containing the sampling weight sums $s_{A,1}, \ldots, s_{A,h}$ (Eq. (8.3)). The set of intervals is the same for each cluster and is stored as a single list of pairs named `intervals`.

```
nIntervals = 30
intervals = [(12+2*i,12+2*(i+1)) for i in range(nIntervals)  ]
histDict = {name:[0]*nIntervals for name in namesList}
```

Dictionary comprehension has been used to create `histDict`. The value associated with a `name` is a list of length 30 containing zeros.

5. We will use the same structure for processing a set of BRFSS files as was used previously, say instruction 9 of Chap. 7, Sect. 7.3.

```
n = 0
dataDict = {}
path = r'../Data/'  # Replace with your path.
fileList = os.listdir(path)
for filename in fileList:
    try:
        shortYear = int(filename[6:8])
        year = 2000 + shortYear
        fields = fieldDict[shortYear]
        sWt, eWt = fields['weight']
        sBMI, eBMI = fields['bmi']
        file = path + filename
    except(ValueError):
        pass
    print(n)
```

A `ValueError` will be thrown if `filename[6:8]` does not contain a string of digits. In that case, the remaining statements in the `try` branch of the exception handler are ignored and interpreter executes the `pass` statement.

6. Open each file in the `fileList` within the `try` branch of the exception handler and process the file records one at a time. Determine the respondent's state of residence and extract the state name from the `stateCodeDict`:

```
with open(file, encoding="utf-8", errors='ignore') as f:
    for record in f:
        stateCode = int(record[:2])
        stateName = stateCodeDict[stateCode]
```

It's difficult to resolve errors when an exception handler is invoked as errors are often not visible. Thus, it may be helpful to execute the code segment beginning with `with open(file,...` as you develop the code rather than allowing the exception handler to be invoked. If you proceed this way, you may find it helpful to temporarily set `file` to be one of the BRFSS data files in the list `fileList`.

7. Extract the sampling weight `weight` from `record` (see Sect. 7.3).
8. Translate the body mass index string to a float using the function `convertBMI`:

```
bmiString = record[sBMI-1:eBMI]
bmi = functions.convertBMI(bmiString,shortYear)
```

   The function `convertBMI` was build in the Chap. 3, Sect. 3.6 tutorial (see instruction 14). We're assuming that it resides in your functions module `functions.py`.
9. Increment the sampling weight total for the interval containing `bmi` in the histogram `histDict[stateName]`. This histogram corresponds to the respondent's state of residence.

```
for i, interval in enumerate(intervals):
    if interval[0] < bmi <= interval[1]:
        histDict[stateName][i] += weight
        break
noDataSet = noDataSet - set([stateName])
n += 1
```

   The next-to-last instruction computes the set difference between `noDataSet` and the singleton data set containing the respondent's state of residence. Allow the program to process all of the data sets.
10. Upon completion of the `for` loop iterating over `fileList`, remove the key-value pairs from `histDict` for which there are no data.

```
print(len(histDict))
print(noDataSet)
for stateName in noDataSet:
    del histDict[stateName]
print(len(histDict))
```

11. Transform the sums of weights contained `histDict` so that the sum over the intervals of a histogram is 1.

```
for state in histDict:
    sumWeights = sum(histDict[state])
    histDict[state] = [intervalTotal/sumWeights
                       for intervalTotal in histDict[state]]
```

12. We'll use the data in its reduced form of `histDict` in the next tutorial. It's convenient then to save the dictionary in a file so that `histDict` need

not be rebuilt. A standard method of saving data structures is the Python module pickle. Write histDict as a pickle file:

```
import pickle
picklePath = '../data/histDict.pkl'
pickle.dump(histDict, open(picklePath, "wb" ))
```

The file histDict.pkl will be read from disk into a Python dictionary at the beginning of the tutorial on the k-means algorithm.

13. Form an initial dictionary of clusters. Each cluster is a key-value pair where the key is a cluster label and a state abbreviation. The value is a singleton list containing the state abbreviation. For example, a dictionary entry will appear as $\{a : [a]\}$.

```
clusterDict = {state: [state] for state in histDict.keys()}
```

When two clusters A and B are merged, the list of states that belong to the cluster A will lengthen by adding those belonging to B. Cluster B will be deleted from the dictionary and the number of key-value pairs in clusterDict will be reduced by 1.

14. Begin building a function that merges clusters. It's best not to make it a function (with the keyword def and a return statement) until the code segment is complete and free of errors. Create a list of the state codes by extracting the keys of clusterDict.

```
stateList = list(clusterDict.keys())
```

15. We'll build the code segment that merges the closest two clusters. Three operations are necessary: build a list of all two-cluster sets, search the list for the closest two clusters, and merge the closest two as one cluster. Building the set of cluster pairs is accomplished using list comprehension:

```
setList = [{a,b} for i,a in enumerate(stateList[:-1])
                    for b in stateList[i+1:]]
```

The outer for loop iterates over all elements stateList except the last. The inner for loop iterates over the elements stateList[i] through the last (stateList[n-1]).

16. Begin the search for the closest pair by initializing the minimum distance between cluster to be 2.[6] Iterate over setList. With clusters A

[6] It can be proved that the distance between any two clusters will be less than 2.

and B, compute the distance between the associated histograms and if the distance is less than the minimum distance, update the minimum distance and save the set with the name `closestSet`. When the `for` loop is complete, `closestSet` will contain the labels of the clusters to merge.

```
mn = 2
for a, b in setList:
    abD = sum([abs(pai - pbi)
                for pai, pbi in zip(histDict[a], histDict[b])])
    if abD < mn:
        mn = abD
        closestSet = {a,b}
    print(closestSet,mn)
```

Our single-line function for computing the distance `abD` between clusters A and B takes the relative proportions from `histDict` and forms pairs $(p_{A,1}, p_{B,1}), \ldots, (p_{A,h}, p_{B,h})$. The `zip` function performs this operation. Then, the function iterates over the `zip` object and builds a list of the absolute differences, i.e., $[|p_{A,1} - p_{B,1}|, \ldots, |p_{A,h} - p_{B,h}|]$ using list comprehension. The last operation computes the sum of the absolute differences.

17. Let A and B denote the closest two clusters. They are merged by replacing the relative proportions for cluster A with the weighted average of the relative proportions from A and B. Hence, A consumes B. In this code segment, the relative proportions for cluster A are updated.

```
a, b = closestSet
na = len(clusterDict[a])
nb = len(clusterDict[b])
histDict[a] = [(na*pai + nb*pbi)/(na + nb) for pai, pbi
                           in zip(histDict[a], histDict[b])]
```

18. Extend the member list of A with the member list of B. Remove B from the cluster dictionary.

```
clusterDict[a].extend(clusterDict[b])
del clusterDict[b]
print(len(clusterDict))
```

You can test the merging operation by repeatedly running the code beginning with the instruction `stateList = list(histDict.keys())` and ending with `print(len(clusterDict))`. On each execution, the length of `clusterDict` will decrement by 1.

19. When the merging operation functions correctly, move it to a function `mergeClusters` that accepts `clusterDict` and `histDict` as arguments and returns `clusterDict` and `histDict`, say,

```
def mergeClusters(clusterDict, histDict):
    stateList = list(clusterDict.keys())
    ...
    del clusterDict[b]
    return clusterDict, histDict
```

20. Reduce the initial set of 54 singleton clusters to 5 clusters using a `for` loop that repeatedly calls `mergeClusters`.

```
while len(clusterDict) > 5:
    clusterDict, histDict = mergeClusters(clusterDict, histDict)
```

21. Print the clusters and the member states to the console.

```
for k,v in clusterDict.items():
    print(k,v)
```

22. We'll use `pyplot` from the `matplotlib` to draw the cluster histograms on a single figure. We've built almost the same plot in Chap. 3, instruction 26. To improve the readability, the histograms are truncated at the body mass index value of 51 kg/m^2.

```
intervals = [(12+2*i,12+2*(i+1)) for i in 30  ]

import matplotlib.pyplot as plt
x = [np.mean(pair) for pair in intervals][:19] # Ignore large BMI
      values.
for name in clusterDict:
    y = histDict[name][:19]
    plt.plot(x, y)
plt.legend([str(label) for label in range(6)], loc='upper right')
plt.show()
```

8.4.1 Synopsis

We have used a hierarchical clustering algorithm to form clusters of states that are similar with respect to the body mass index of adult residents.

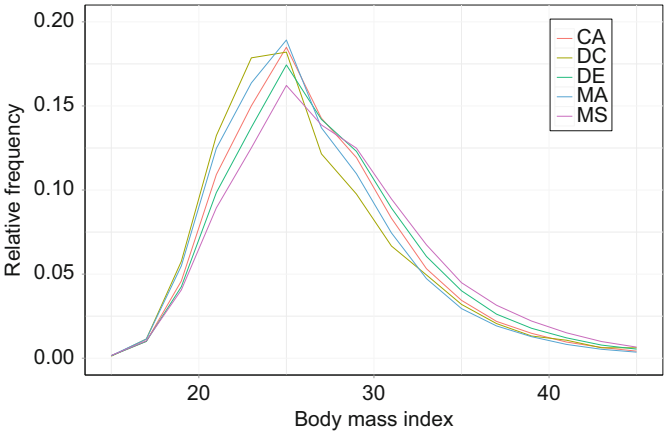

Fig. 8.2 Estimated distributions of body mass index for five clusters of states. Clusters are identified by state abbreviation

The analysis began with the estimation of body mass index distributions for each state. State to state and cluster to cluster similarity was measured by the distance between relative frequency histograms. When two clusters were merged, the merged cluster histogram was computed as a weighted average of the histograms belonging to the merged clusters. The weights reflected the numbers of states belonging to the respective clusters.

This application of cluster analysis corresponds to a more traditional set-up in which each observational unit would have provided a vector of observations on $h = 30$ variables. In our application, an observational unit is a state. The similarity calculation would have used the h variables. Some transformation, say standardization for instance, probably would have been used to try to account for differences in scale among the variables. Scaling was not necessary in this application.

Figure 8.2 shows that the cluster distributions are different though perhaps not to a striking extent. The difference in height between histograms for a fixed value of the horizontally-plotted variable (body mass index in this case) is the primary attribute reflecting differences in distributions. With this in mind, it's clear that the District of Columbia (DC) is very different than the Mississippi cluster (MS). For instance, all of the histograms have a mode at 25 kg/m^2 and the relative frequencies for the interval $[24, 26)$ are .182 and .162 for DC and MS respectively. The differences reverse in sign at the body mass index value 26 kg/m^2, reflecting a greater proportion of individuals with large values of body mass index in the Mississippi cluster than in the District of Columbia. The District of Columbia is a singleton cluster and so informally speaking, there are no states similar to it.

Though the differences are small for any single interval, there are 30 intervals, and the cumulative difference are large. In fact, 30.7% of the individuals in the Mississippi cluster have a body mass index at least 30 kg/m^2 and hence are classified as obese whereas only 21.7% of the District of Columbia sample is obese.

It's often difficult to judge whether a cluster analysis has successfully created clusters that are meaningfully different. In this example, we are able to do so by recalling that the motivation for analyzing body mass index is the prevalent view in public health circles that body mass index is associated with a number of chronic diseases (type 2 diabetes in particular). We may attempt to confirm that position by returning to the analysis of diabetes prevalence and incidence carried out in Chap. 7. In particular, we reconstructed Fig. 7.1 as Fig. 8.3 and identified cluster membership of states by color. States belonging to a particular cluster are usually near each other. Only the green cluster is not in a well-defined region.

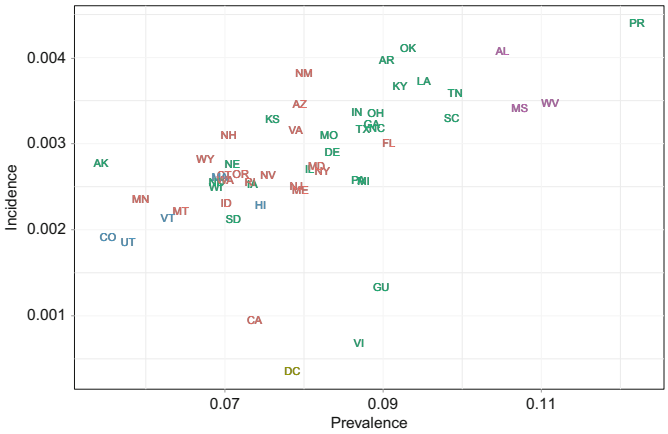

Fig. 8.3 Estimated incidence and prevalence of diabetes. States have been colored according to their membership in one of five clusters

8.5 The *k*-Means Algorithm

Our example of representative-based cluster analysis is the k-means algorithm. Now the number of clusters is determined by the analyst. The algorithm begins by randomly assigning the observation vectors in the data set $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ to k clusters. This initial set of clusters is denoted as $\{A_1, \ldots, A_k\}$. Then, the *centroids* are computed for each cluster. Traditionally, the centroids are the multivariate means of the observations belonging to the cluster. The centroids represent the cluster in the calculation of distance from observation to cluster.

The initial configuration is rarely satisfactory and so the bulk of computational effort is aimed at improving on it. The algorithm iterates between two steps: assign every observation to the nearest cluster, then recompute, or update the cluster centroids. If any observation is reassigned, that is, moved out of its currently assigned cluster and into another, then the centroids of the two clusters will change. Therefore, another iteration should take place.

The algorithm continues iterating between the two steps until no further reassignments are made. At that point, each observation belongs to the cluster to which it is closest. Starting from the initial random configuration, the within cluster sums-of-squares has been minimized. This is because the within sums-of-squares is equivalent to the sum of the Euclidean distances between observations and cluster centroids, and because moving any observation now will increase the sum of Euclidean distances (and the within sums-of-squares). It follows that the algorithm has reached a best possible assignment of observations to clusters and a best possible calculation of centroids.

The algorithm has considerable appeal because a popular objective function has been minimized by the algorithm (the within sums-of-squares). The drawback of the algorithm is that the initial configuration is random. A different configuration will often lead to a different set of clusters. If the analyst can begin with a selectively chosen initial configuration, then the algorithm may produce more satisfactory clusters.

Let's look at the details. The centroid of cluster A_i is the multivariate mean

$$\overline{\mathbf{x}}_i = n_i^{-1} \sum_{\mathbf{x}_j \in A_i} \mathbf{x}_j, \tag{8.7}$$

where n_i is the number of observations belonging to A_i and $\mathbf{x}_j = [x_{j,1} \cdots x_{j,h}]^T$. The number of attributes is h and the lth element of $\overline{\mathbf{x}}_i$ is

$$\overline{x}_{i,l} = n_i^{-1} \sum_{\mathbf{x}_j \in A_i} x_{j,l}, \tag{8.8}$$

for $l = 1, \ldots, h$. The distance between $\mathbf{x}_j \in D$ and A_i is defined to be the distance between $\mathbf{x}_j$ and the centroid of A_i, $\overline{\mathbf{x}}_i$. Since $\overline{\mathbf{x}}_i = [\overline{x}_{i,1} \ \cdots \ \overline{x}_{i,h}]^T$, the squared Euclidean distance is

$$d_E^2(\mathbf{x}_j, \overline{\mathbf{x}}_i) = \sum_{l=1}^{h} (x_{j,l} - \overline{x}_{i,l})^2. \tag{8.9}$$

We may use the squared Euclidean distance in the determination of the nearest centroid to $\mathbf{x}_j$ instead of Euclidean distance since the ordering, nearest to most distant, will be the same.

Upon completion of the initialization phase, the algorithm begins iterating between two steps.

The first step iterates over the data set D and computes the squared Euclidean distances between each $\mathbf{x}_j \in D$ and each cluster centroid according to Eq. (8.9). If an observation is found to be nearest to a different cluster than its currently assigned cluster, then it is reassigned to the nearest cluster. The second step updates the centroid of cluster A_i according Eq. (8.7). Every cluster that has changed membership in the last iteration must have an update computed to its centroid. That completes the two steps. If any observation has been reassigned, then another iteration commences. The algorithm terminates when no further changes in membership occur.

8.6 Tutorial: The *k*-Means Algorithm

We'll continue with the exercise of grouping states with respect to body mass index distribution of adult residents. The same data set of state body mass index histograms is used and so this tutorial begins not with data preparation but with programming the k-means algorithm. It's assumed that histogram representations of the body mass index have been computed in the course of the previous tutorial and are stored in the dictionary `histDict`. The keys of the dictionary are states[7] and the values are lists containing the estimated proportion of adult state residents with body mass indexes in the $h = 30$ intervals $(12, 14], (14, 16], \ldots, (70, 72]$.

We choose to create $k = 6$ clusters. Each observation is a state, as before. The contents of the observation vector are the same but we're changing the notation to be consistent with the development of the k-means algorithm. For the jth state, the estimated proportion of individuals in interval l is $x_{j,l}, l = 1, 2, \ldots, h$.[8] The vector of estimated proportions associated with observation j is now denoted generically by

$$\mathbf{x}_j = [x_{j,1} \ \cdots \ x_{j,h}]^T. \tag{8.10}$$

For cluster A_i, the centroid is $\overline{\mathbf{x}}_i = [\overline{x}_{i,1} \ \cdots \ \overline{x}_{i,h}]^T$. The centroid is computed using formula (8.7).

A note on programming the algorithm: the k-means algorithm is very fast and so it may not an advantage to write computationally efficient code if it requires a significant effort beyond writing simple and relatively slow code. For example, the first step of every iteration updates the cluster centroids. There are two ways to update the cluster centroids: recompute every centroid, or update the centroids that need to be updated by computing the centroid sum, subtracting from the sum the observation vectors that have been removed from the cluster and adding to the sum the observation vectors that have been added to the cluster. Then, divide by the number of observations in the cluster. Which is best? It depends on the use of the algorithm. Saving a few seconds of computation time is not worth an hour of programming time. We will recompute all of the centroids.

The program consists of two primary blocks: reading the data and initializing the clusters, and the k-means algorithm.

1. Load your pickle file containing the dictionary of state body mass index histograms and store the contents of the file in a dictionary with the name `histDict`.[9]

[7] Recall from the tutorial of Sect. 8.4 that there are actually 54 geographic entities that we are loosely referring to as state.

[8] The previous notation for the estimated proportion of individuals in interval l and observation j, was $p_{j,l}$.

[9] The pickle file was created in instruction 12 of the tutorial of Sect. 8.4.

```
import numpy as np
import pickle
picklePath = '../histDict.pkl'
histDict = pickle.load( open(picklePath, "rb" ) )
print(len(histDict))
```

2. Randomize the list of states so that the initial assignment of state to cluster will be random. Use the `Numpy` function `random.choice`. The essential arguments to be passed are a list from which to sample from (we pass the keys of `histDict`), the number of units to sample (`size = n`), and the type of sampling. Sampling must be *without* replacement so that a state does not appear in more than one of the k lists of cluster members.

```
k = 6
n = len(histDict)
randomizedNames = np.random.choice(list(histDict.keys()), size = n,
                    replace = False)
```

   The call to `randomize` returns a random sample of size n without replacement. The effect of this code segment is to randomly shuffle the order of the states in `randomizedNames`.
3. Build a dictionary `clusterDict` that contains the assignments of state to initial cluster. Use a `for` loop. The keys are computed as i mod k where $k = 6$ is the number of clusters and $i \in \{0, 1, \ldots, n-1\}$. The dictionary values are lists of states.

```
clusterDict = {}
for i, state in enumerate(randomizedNames):
    clusterDict.setdefault(i%k,[]).append(state)
print(clusterDict)
```

   The `setdefault` function was discussed in Sect. 4.6.2 and used in instruction 27 of Sect. 4.6.2. The `enumerate` function was described in instruction 32 of Sect. 4.6.2.
4. Write a function that will recompute a histogram for each cluster in the cluster dictionary `clusterDict`. The histogram for each cluster is computed by iterating over each state in the cluster and adding up the estimated proportions in each histogram interval (Eq. (8.8)). We begin creating a dictionary `clusterHistDict` to store the k cluster histograms. Then, a `for` loop iterates over clusters and a second inner `for` loop iterates over states belonging to clusters. The structure is

```
h = 30
clusterHistDict = dict.fromkeys(clusterDict,[0]*h)
for a in clusterDict:
    sumList = [0]*h
    for state in clusterDict[a]:
        print(a,state)
```

The list `sumList` will store the sum of estimated proportions for each interval as the program iterates over members of the cluster.

5. Terms are added to `sumList` by zipping `sumList` with the state histogram `histDict[state]`. Then, we use list comprehension to update `sumList` with the data stored in `histDict[state]`. The last step is to divide the sums by the number of states belonging to the cluster and store the list in the cluster histogram dictionary.

```
for state in clusterDict[a]:
    sumList = [(sumi + pmi) for sumi, pmi in
                  zip(sumList, histDict[state])]
na = len(clusterDict[a])
clusterHistDict[a] = [sumi/na for sumi in sumList]
```

This code segment executes *within* the `for` loop that iterates over `clusterDict` and replaces the last two lines of instruction 4.

6. Turn the code that updates the cluster centroids stored in `clusterHistDict` into a function:

```
def clusterHistBuild(clusterDict, histDict):
    ...
    return clusterHistDict
```

7. Move the function to the top of the script. Build the initial cluster centroids by calling the function

```
clusterHistDict = clusterHistBuild(clusterDict,histDict)
```

This instruction immediately follows the construction of `clusterDict` (instruction 3).

8. The next code segment is the start of the k-means algorithm iteration phase. There are two steps to be executed. The first step reassigns an observation (a state) to a different cluster if the state is closest to some other cluster besides the cluster that it is currently assigned to. The sec-

ond step updates the centroids stored in `clusterHistDict` if any states have been reassigned. These steps are repeated until no state is reassigned to a different cluster

A conditional structure using the `while` statement repeatedly executes the code within the structure. Program flow breaks out of the structure when the boolean variable `repeat` is false. The structure is so:

```
repeat = True
while repeat == True:
    updateDict = {}
    # Step 1:
    ...
    # Step 2:
    if clusterDict != updateDict:
         clusterDict = updateDict.copy()
         clusterHistDict = clusterHistBuild(clusterDict,histDict)
    else:
        repeat = False
```

The dictionary `updateDict` contains the updated clusters. It has the same structure as `clusterDict` and so there are k key-value pairs. A key is a cluster index ($i = 1, 2, \ldots, k$) and the value is a list of member states. It's built by determining the nearest cluster to each state and assigning the state to the nearest cluster.

If `clusterDict` is not equal to `updateDict` then at least one state has been reassigned to a different cluster. If this is the case, then we update the cluster dictionary `clusterDict` and the dictionary of cluster histograms `clusterHistDict`. If `clusterDict` is equal to `updateDict`, then the algorithm is complete and program flow is directed to the next statement following the conditional structure.

We must use the `.copy()` function when assigning the contents of `updateDict` to `clusterDict`. If we set `clusterDict = updateDict`, then the two objects will henceforth use same memory address. Any change to one dictionary immediately makes the same change to the other. Using `.copy()` writes the values of `updateDict` into the memory location of `clusterDict`. The dictionaries contain the same values but are different objects. A change to one has no effect on the other.

Step 2 is in place.

9. Let's program step 1. This code segment executes after `updateDict` is initialized and replaces the placeholder `...` in instruction 8. Its purpose is to build `updateDict`. Iterate over states, find the nearest cluster to a state, and assign the state to that nearest cluster.

```
for state in histDict:
    mnD = 2 # Initialize the nearest state-to-cluster distance.
    stateHist = histDict[state]
    for a in clusterDict:
        clusterHist = clusterHistDict[a]
        abD = sum([(pai-pbi)**2 for pai,pbi in zip(stateHist,
                   clusterHist)])
        if abD < mnD:
            nearestCluster = a
            mnD = abD
    updateDict.setdefault(nearestCluster,[]).append(state)
```

We initialize the nearest distance to be `mn = 2` since the distance between clusters cannot be larger than 2 (see exercise 8.1).

10. After the `else` branch (instruction 8), print out the current assignments of states to clusters.

```
for a in updateDict:
    print('Cluster =',a,' Size = ',len(updateDict[a]),updateDict[a])
```

This `for` statement should be aligned with the `if` and `else` keywords.

11. Plot the k histograms. The code below is only slightly different than that used to plot the cluster histograms in the tutorial of Sect. 3.6.

```
import matplotlib.pyplot as plt

nIntervals = 30
intervals = [(12+2*i,12+2*(i+1)) for i in range(nIntervals)  ]

x = [np.mean(pair) for pair in intervals][:19]
for name in clusterDict:
    print(name )
    y = clusterHistDict[name][:19]

    plt.plot(x, y)
plt.legend([str(label) for label in range(6)], loc='upper right')
plt.show()
```

12. Recall that the initial configuration is random and so the final output may differ if the algorithm executes more than once. To gain some insight into the effect of the initial configuration, execute the code that begins with the random assignment of names to clusters and ends with constructing the figure several times. Try a few values of k, say $k \in \{4, 5, 6\}$. You should observe that a few associations of states are formed with regularity.

8.6.1 Synopsis

The k-means algorithm has the desirable property of always producing the same number of clusters but it also has the unfortunate property of not being deterministic in the sense that the initial random cluster configuration usually affects the final configuration. Having been introduced to two completing methods of cluster analysis, hierarchical agglomerative and k-means algorithms, we are faced with a decision: which to use?

Generally, when choosing one method from among a set of candidate methods that may be used accomplish an analytical objective, we strive to evaluate the candidate methods with respect to the theoretical foundations motivating the methods.[10] The k-means algorithm is an example of a method that can be motivated as the solution to a compelling minimization problem [2].

Suppose that we set out to form clusters so that the members are as close as possible to their centers. To make the objective more concrete, we define the within-cluster sums of squares associated with a particular set of assignments of observations to clusters, call it C, as

$$S(C) = \sum_{i=1}^{k} \sum_{j=1}^{n_i} ||\mathbf{x}_{i,j} - \overline{\mathbf{x}}_i||^2, \tag{8.11}$$

where $\overline{\mathbf{x}}_i = n_i^{-1} \sum_j \mathbf{x}_{i,j}$ is the vector of means computed from the n_i members of cluster i, for $i = 1, \ldots, k$, and $||\mathbf{y}|| = (\mathbf{y}^T \mathbf{y})^{1/2}$ is the Euclidean norm of the vector $\mathbf{y}$.

Given a particular initial configuration, the k-means algorithm will produce a rearrangement C that minimizes the within sums-of-squares. A solution C^* (i.e., an arrangement of observations into clusters) found by the k-means algorithm does not necessarily achieve the global minimum of $S(\cdot)$ over all possible configurations. The global minimum is the smallest value of the within sums-of-squares over every possible arrangement of states into k clusters. However, the algorithm may be executed repeatedly using N different initial configurations. A best solution may be selected as the solution that yielded the smallest value of $S(C)$ among the set of solutions $S(C_1), \ldots, S(C_N)$.

There's a variety of variations on the k-means algorithm. For example, the k-medoids algorithm uses observation vectors as cluster centers. A somewhat more complex algorithm with a solid foundation in statistical theory does not assign observations to clusters but instead estimates the probability of cluster membership of every observation in each cluster. Adopting this probabilistic modeling approach leads to the finite mixtures model and an EM algorithm solution [39].

[10] Other criteria are usually considered and may outweigh theoretical considerations.

8.7 Exercises

8.7.1 Conceptual

8.1. In the Sect. 8.4 tutorial, we set the initial minimum distance between clusters to be 2 (instruction 15). Prove the maximum distance between any two clusters is no more than 2, thereby insuring that the final minimum distance will not be 2. Specifically, prove that

$$\sum_i |a_i - b_i| \leq 2, \tag{8.12}$$

where $\sum a_i = \sum_i b_i = 1$.

8.2. For a particular configuration C of observations to clusters, show that the sample mean vectors $\overline{\mathbf{x}}_i = n_i^{-1} \sum_j \mathbf{x}_{ij}$ minimize $S(C)$. Begin with the objective function

$$S(C) = \sum_{i=1}^{k} \sum_{j=1}^{n_i} ||\mathbf{x}_{ij} - \boldsymbol{\mu}_i||^2. \tag{8.13}$$

Determine the vector $\widehat{\boldsymbol{\mu}}_i$ that minimizes $S(C)$ with respect to $\boldsymbol{\mu}_i$. Argue that if $\widehat{\boldsymbol{\mu}}_i$ minimizes $\sum_{j=1}^{n_i} ||\mathbf{x}_{ij} - \boldsymbol{\mu}_i||^2$, then $\widehat{\boldsymbol{\mu}}_1, \ldots, \widehat{\boldsymbol{\mu}}_k$ minimize $S(C)$.

8.7.2 Computational

8.3. Change the k-means algorithm so that the distance between state histograms and the cluster centroid is computed using the city-block (or L_1) metric instead of the Euclidean (or L_2) metric. After finding a set of clusters using the Euclidean metric, run the algorithm again using the city-block metric. Use the final configuration from the Euclidean metric as the initial configuration for the city-block metric. How often does a change in metric result in a different arrangement?

8.4. Return to the tutorial of Sect. 8.4 and modify the program so that the inter-cluster distance between clusters A and B is computed according to

a. The smallest distance between any $\mathbf{x}_i \in A$ and any $\mathbf{x}_j \in B$, say,

$$d_{\min}(A, B) = \min_{\mathbf{x}_i \in A, \mathbf{x}_j \in B} d_c(\mathbf{x}_i, \mathbf{x}_j)$$

b. The largest distance between any $\mathbf{x}_i \in A$ and any $\mathbf{x}_j \in B$, say,

$$d_{\max}(A, B) = \max_{\mathbf{x}_i \in A, \mathbf{x}_j \in B} d_c(\mathbf{x}_i, \mathbf{x}_j)$$

c. The average distance between $\mathbf{x}_i \in A$ and $\mathbf{x}_j \in B$, say,

$$d_{\text{mean}}(A, B) = (n_A n_B)^{-1} \sum_{\mathbf{x}_i \in A, \mathbf{x}_j \in B} d_c(\mathbf{x}_i, \mathbf{x}_j)$$

Each one of these variants has a name [29]: single linkage (a), complete linkage (b), and average linkage (c). The metric programmed in Sect. 8.4 is known as centroid linkage.

8.5. Use the hierarchical agglomerative algorithm to build $k = 6$ clusters. Use these clusters as the initial configuration to the k-means algorithm. Compare the initial and final configurations.