Chapter 9
Reducing Data Complexity

Marketing datasets often have many variables—many *dimensions*—and it is advantageous to reduce these to smaller sets of variables to consider. For instance, we might have many questions (e.g. 9) on a consumer survey that reflect a smaller number (such as 3) of underlying concepts such as *customer satisfaction* with a service, *category leadership* for a brand, or *luxury* for a product. If we can reduce the data to its underlying dimensions, we can more clearly identify the underlying relationships among concepts.

In this chapter we consider three common methods to reduce data complexity by reducing the number of dimensions in the data. *Principal component analysis* (PCA) attempts to find uncorrelated linear dimensions that capture maximal variance in the data. *Exploratory factor analysis* (EFA) also attempts to capture variance with a small number of dimensions while seeking to make the dimensions interpretable in terms of the original variables. *Multidimensional scaling* (MDS) maps similarities among observations in terms of a low-dimension space such as a two-dimensional plot. MDS can work with metric data and with non-metric data such as categorical or ordinal data.

In marketing, PCA is often associated with *perceptual maps*, which are visualizations of respondents' associations among brands or products. In this chapter we demonstrate perceptual maps for brands using principal component analysis. We then look at ways to draw similar perceptual inferences from factor analysis and multidimensional scaling.

9.1 Consumer Brand Rating Data

We investigate dimensionality using a simulated dataset that is typical of consumer *brand perception* surveys. These data reflect consumer ratings of *brands* with regard to *perceptual adjectives* as expressed on survey items with the following form:

> On a scale from 1 to 10 — where 1 is *least* and 10 is *most* — how [ADJECTIVE] is *[BRAND A]*?

In these data, an observation is one respondent's rating of a brand on one of the adjectives. Two such items might be:

1. How *trendy* is *Intelligentsia Coffee*?
2. How much of a *category leader* is *Blue Bottle Coffee*?

Such ratings are collected for all the combinations of adjectives and brands of interest.

The data here comprise simulated ratings of 10 brands ("a" to "j") on 9 adjectives ("performance," "leader," "latest," "fun," and so forth), for N = 100 simulated respondents. The dataset is provided on this book's web site. We start by loading and checking the data:

```
In [1]: import pandas as pd
        brand_ratings = pd.read_csv('http://bit.ly/PMR-ch9')
        brand_ratings.head()
```

```
Out[1]:    perform  leader  latest  fun  serious  bargain  value  trendy  \
    0        2       4       8     8        2        9      7       4
    1        1       1       4     7        1        1      1       2
    2        2       3       5     9        2        9      5       1
    3        1       6      10     8        3        4      5       2
    4        1       1       5     8        1        9      9       1
```


J. S. Schwarz et al., *Python for Marketing Research and Analytics*, https://doi.org/10.1007/978-3-030-49720-0_9

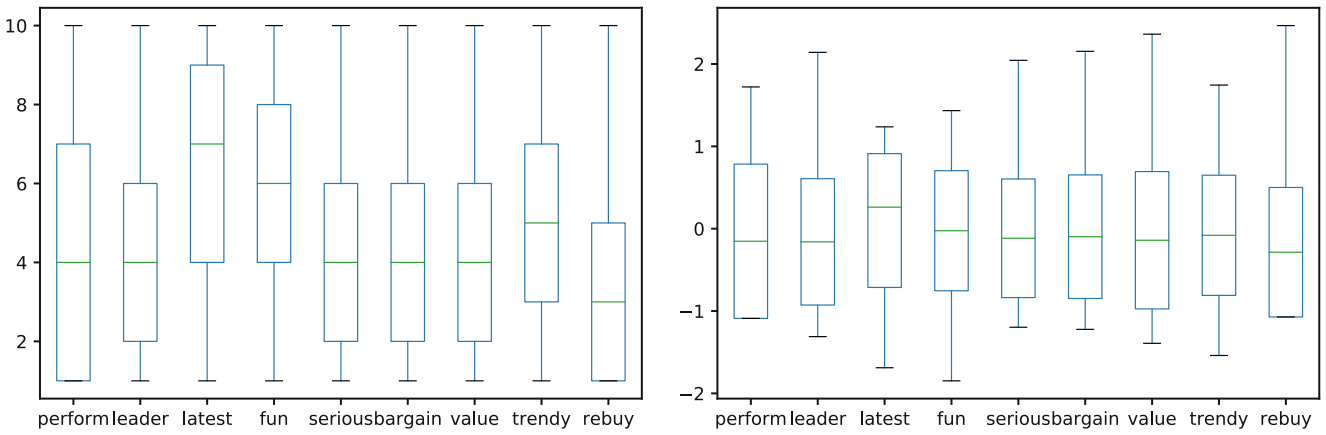

Fig. 9.1 Boxplots for the unscaled (left) and scaled (right) data offer a quick way to check the distribution of the dataset

```
   rebuy brand
0      6     a
1      2     a
2      6     a
3      1     a
4      1     a
```

```
In [2]: brand_ratings.tail()# Not shown
```

Each of the 100 simulated respondents has provided ratings for each of the 10 brands, so there are 1000 total rows. We inspect the `describe()` to check the data quality and structure:

```
In [3]: brand_ratings.describe().round(2)
```

```
Out[3]:         perform   leader   latest      fun  serious  bargain  \
        count   1000.00  1000.00  1000.00  1000.00  1000.00  1000.00
        mean       4.49     4.42     6.20     6.07     4.32     4.26
        std        3.20     2.61     3.08     2.74     2.78     2.67
        min        1.00     1.00     1.00     1.00     1.00     1.00
        25%        1.00     2.00     4.00     4.00     2.00     2.00
        50%        4.00     4.00     7.00     6.00     4.00     4.00
        75%        7.00     6.00     9.00     8.00     6.00     6.00
        max       10.00    10.00    10.00    10.00    10.00    10.00

                  value   trendy    rebuy
        count   1000.00  1000.00  1000.00
        mean       4.34     5.22     3.73
        std        2.40     2.74     2.54
        min        1.00     1.00     1.00
        25%        2.00     3.00     1.00
        50%        4.00     5.00     3.00
        75%        6.00     7.00     5.00
        max       10.00    10.00    10.00
```

We see in `describe()` that the ranges of the ratings for each adjective are 1–10. The data appear to be clean and formatted appropriately.

We can also use a boxplot for another view of the distribution of each variable, as displayed in the left panel of Fig. 9.1:

```
In [4]: brand_ratings.plot.box()
```

Table 9.1 Adjectives in the `brand_ratings` data and examples of survey text that might be used to collect rating data

Perceptual adjective (column name)	Example survey text:
Perform	*Brand* has strong performance
Leader	*Brand* is a leader in the field
Latest	*Brand* has the latest products
Fun	*Brand* is fun
Serious	*Brand* is serious
Bargain	*Brand* products are a bargain
Value	*Brand* products are a good value
Trendy	*Brand* is trendy
Rebuy	I would buy from *Brand* again

There are nine perceptual adjectives in this dataset. Table 9.1 lists the adjectives and the kind of survey text that they might reflect.

9.1.1 Rescaling the Data

It is often good practice to rescale raw data. This makes data more comparable across individuals and samples. A common procedure is to *center* each variable by subtracting its mean from every observation, and then *rescaling* those centered values as units of standard deviation. This is commonly called *standardizing*, *normalizing*, or *Z-scoring* the data (Sect. 7.3.3). Note that we use `np.arange()` here rather than `range()` because we want the output to be a NumPy array rather than a list so we can do vectorized operations on it.

In Python, data could be standardized with a mathematical expression using `mean()` and `sd()`:

```
In [5]: import numpy as np
        x = np.arange(1000)
        x_sc = (x - x.mean())/x.std()
        print('mean: {}\nmedian: {}\nmax: {}\nmin: {}'.format(x_sc.mean(),
                                                             np.median(x_sc),
                                                             x_sc.max(),
                                                             x_sc.min()))

mean: 0.0
median: 0.0
max: 1.7303196219213355
min: -1.7303196219213355
```

As we saw in Sect. 7.3.3, a simpler way is to use `sklearn.preprocessing.scale()` to rescale all variables at once. We never want to alter raw data, so we assign the raw values first to a new dataframe `brand_ratings_sc` and alter that:

```
In [6]: from sklearn.preprocessing import scale
        brand_ratings_sc = brand_ratings.copy()
        brand_ratings_sc.iloc[:, :-1] = scale(brand_ratings_sc.iloc[:, :-1])
        brand_ratings_sc.plot.box()
        brand_ratings_sc.describe().round(2)

Out[6]:         perform   leader   latest      fun  serious  bargain  \
        count  1000.00  1000.00  1000.00  1000.00  1000.00  1000.00
        mean     -0.00     0.00    -0.00     0.00    -0.00    -0.00
        std       1.00     1.00     1.00     1.00     1.00     1.00
        min      -1.09    -1.31    -1.69    -1.85    -1.20    -1.22
        25%      -1.09    -0.93    -0.71    -0.75    -0.84    -0.85
        50%      -0.15    -0.16     0.26    -0.02    -0.12    -0.10
```

```
75%       0.78     0.61     0.91     0.70     0.60     0.65
max       1.72     2.14     1.24     1.43     2.04     2.15

          value   trendy    rebuy
count   1000.00  1000.00  1000.00
mean      -0.00    -0.00     0.00
std        1.00     1.00     1.00
min       -1.39    -1.54    -1.07
25%       -0.97    -0.81    -1.07
50%       -0.14    -0.08    -0.29
75%        0.69     0.65     0.50
max        2.36     1.74     2.47
```

In this code we name the new dataframe with extension "`_sc`" to remind ourselves that observations have been scaled. We operate on columns 1–9 because the tenth column is a string variable for brand. (This is accomplished by selecting those columns using `.iloc[:, :-1]`, which excludes the last column of the dataframe.) We see that the mean of each adjective is correctly 0.00 across all brands because the data are rescaled. Observations on the adjectives have a spread (difference between `min` and `max`) of roughly 3 standard deviation units. This means the distributions are *platykurtic*, flatter than a standard normal distribution, because we would expect a range of more than 4 standard deviation units for a sample of this size. (Platykurtosis is a common property of survey data, due to floor and ceiling effects.)

For our initial exploration, we will use the unscaled data. It is generally a good idea to start with the unscaled data, as it's more interpretable. However differences in scale between the variables can complicate an analysis. In practice, we generally run our full analysis on unscaled and then scaled data and evaluate both. Generally, scaled data are "safer" to work with, as any effects from differences in scale between variables are eliminated. In this chapter we will start our exploration with the unscaled data and then move to the scaled data when we start the dimensionality reduction analyses.

9.1.2 Correlation Between Attributes

We can generate the correlation matrix using `corr()` and visualize it using Seaborn for initial inspection of bivariate relationships among the variables:

```
In [7]: import matplotlib.pyplot as plt
        import seaborn as sns

        sns.clustermap(brand_ratings.corr(), annot=True, fmt=".2f",
                       cmap=plt.cm.bwr)
```

Using `clustermap()` rather than `heatmap()`, reorders the rows and columns according to variables' similarity in a hierarchical cluster solution such that adjectives close to each other (such as rebuy, bargain, value) are plotted adjacent to each other (see Sect. 10.3.2 for more on hierarchical clustering). The `annot=True` argument adds the value in each grid cell and the `fmt="0.2f"` specifies the formatting as floats with two decimal places. The result is shown in Fig. 9.2, where we see that the ratings seem to group into three clusters of similar variables, a hypothesis we examine in detail in this chapter.

9.1.3 Aggregate Mean Ratings by Brand

Perhaps the simplest business question in these data is: "What is the average (mean) position of the brand on each adjective?" We can use `groupby()` (see Sects. 3.2.1 and 5.2) to find the mean of each variable by brand:

```
In [8]: brand_means = brand_ratings.groupby('brand').mean().round(3)
        brand_means
```

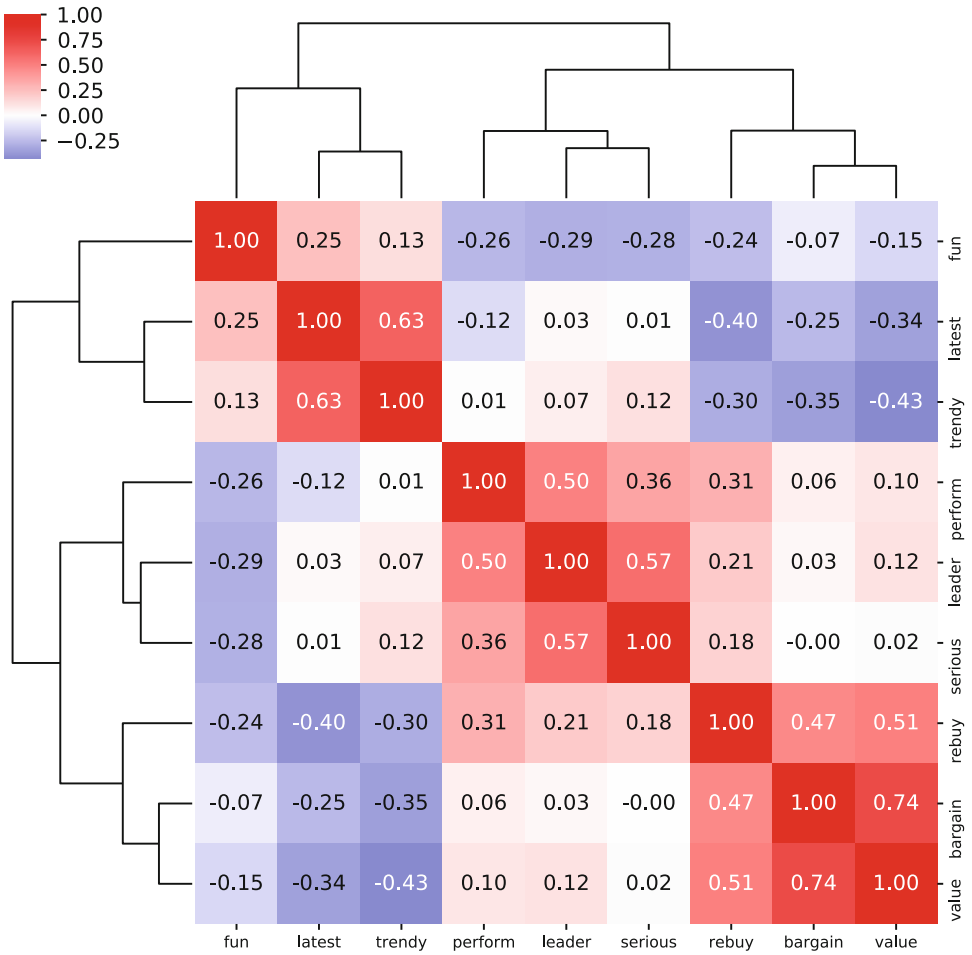

Fig. 9.2 Correlation plot for the simulated consumer brand ratings. This visualization of the basic data appears to show three general clusters that comprise *fun/latest/trendy*, *rebuy/bargain/value*, and *perform/leader/serious* respectively

```
Out[8]:        perform  leader  latest   fun  serious  bargain  value  \
brand
a                 1.65    3.04    7.46  7.87     1.77     4.83   4.78
b                 7.47    7.21    8.43  3.40     7.61     4.37   4.70
c                 6.57    7.45    5.88  3.75     7.72     2.64   3.28
d                 2.31    2.87    7.28  6.58     2.40     1.91   2.10
e                 2.68    4.92    7.60  6.88     4.44     5.73   5.34
f                 4.30    5.12    2.31  5.47     5.96     6.59   6.79
g                 7.43    3.98    2.24  4.65     2.84     6.65   7.35
h                 4.44    3.64    7.74  8.03     3.93     2.29   2.46
i                 5.56    3.58    7.29  7.20     3.91     3.58   2.41
j                 2.47    2.36    5.72  6.85     2.65     4.00   4.16

        trendy  rebuy
brand
a         3.78   2.21
b         7.25   4.33
c         5.29   3.39
d         7.24   2.47
e         5.60   3.82
f         2.99   7.18
```

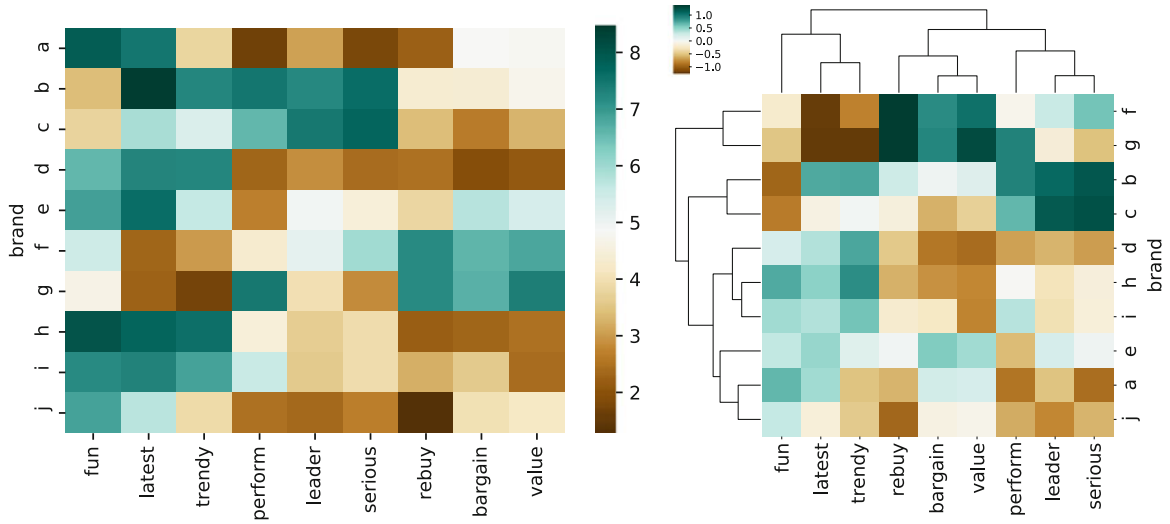

Fig. 9.3 A heatmap (left) and clustermap (right) of the mean of each adjective by brand. Brands *f* and *g* are similar—with high ratings (in green) for *rebuy* and *value* but low ratings for *latest* and *fun*. Other groups of similar brands are *b/c*, *i/h/d*, and *a/j*

```
g        1.72    7.19
h        7.59    2.19
i        6.84    3.21
j        3.90    1.28
```

A *heatmap* is a useful way to examine such results because it colors data points by the intensities of their values. We use `heatmap()` from the `seaborn` package (Waskom et al. 2018). We specify the colormap using the `cm` module from matplotlib.

```
In [9]: from matplotlib import cm

        sns.heatmap(brand_means[['fun', 'latest', 'trendy', 'perform',
                                 'leader', 'serious', 'rebuy', 'bargain',
                                 'value']], cmap=cm.BrBG)
```

The resulting heatmap is shown in the left panel of Fig. 9.3. In this chart's brown-to-blue-green (`"BrBG"`) palette a brown color indicates a low value and dark green indicates a high value; lighter colors are for values in the middle of the range. The brands are clearly perceived differently with some brands rated high on performance and leadership (brands *b* and *c*) and others rated high for value and intention to rebuy (brands *f* and *g*). We order the columns to match the correlation plot above, but the rows are in alphabetical order, i.e. arbitrary order.

We can use `clustermap()`, which sorts the columns and rows in order to emphasize similarities and patterns in the data, which can be seen in the right panel of Fig. 9.3:

```
In [10]: sns.clustermap(brand_means, cmap=cm.BrBG)
```

It does this using a form of hierarchical clustering (see Sect. 10.3.2). We should use some caution interpreting those clusters, but it can be a helpful way to visualize groupings in the data, and is a good preliminary analysis to a more formal cluster analysis. See Chap. 10 for an introduction to clustering.

Looking at Figs. 9.2 and 9.3 we could guess at the groupings and relationships of adjectives and brands. For example, there is similarity in the color pattern across columns for the *bargain/value/rebuy*; a brand that is high on one tends to be high on another. But it is better to formalize such insight, and the remainder of this chapter discusses how to do so.

9.2 Principal Component Analysis and Perceptual Maps

Principal Component Analysis (PCA) recomputes a set of variables in terms of linear equations, known as *components*, that capture linear relationships in the data (Jolliffe 2002). The first component captures as much of the variance as possible from all variables as a single linear function. The second component captures as much variance as possible that remains after

the first component. This continues until there are as many components as there are variables. We can use this process to reduce data complexity by then retaining and analyzing only a subset of those components—such as the first one or two components—that explain a large proportion of the variation in the data.

By extracting components, one can derive a reduced set of variables that captures as much of the variance as desired, yet where each of the measures is independent of the others, which can help us better interpret high-dimensional data.

9.2.1 PCA Example

We explore PCA first with a simple dataset to see and develop intuition about what is happening. We create highly correlated data by copying a random vector `xvar` to a new vector `yvar` while replacing half of the data points. Then we repeat that procedure to create `zvar` from `yvar`:

```
In [11]: np.random.seed(98286)
         xvar = np.random.randint(low=0, high=10, size=100)
         yvar = xvar.copy()
         yvar[:50] = np.random.randint(low=0, high=10, size=50)
         zvar = yvar.copy()
         zvar[25:75] = np.random.randint(low=0, high=10, size=50)
         myvars = np.array([xvar, yvar, zvar])
```

`yvar` will be correlated with `xvar` because 50 of the observations are identical while 50 are newly sampled random values. Similarly, `zvar` keeps 50 values from `yvar` (and thus also inherits some from `xvar`, but fewer). We compile those three vectors into a matrix. Note that we use these data only to have a simple demonstration; this procedure is not a good way to generate formally correlated variables. If you want to generate correlated data, see `np.random.multivariate_normal()`.

We check one of the three possible bivariate plots along with the correlation matrix. If we simply plotted the raw data, there would be many overlapping values because the responses are discrete (integers 1–10). To separate and visualize multiple points with the same values, we use `swarmplot()`, which adjusts points so you can see how many points there are at the same value (similar to jittering, see Sect. 4.6.1):

```
In [11]: sns.swarmplot(x=xvar, y=yvar, color='k')
```

The bivariate plot in Fig. 9.4 shows a clear linear trend for `yvar` vs. `xvar` on the diagonal.

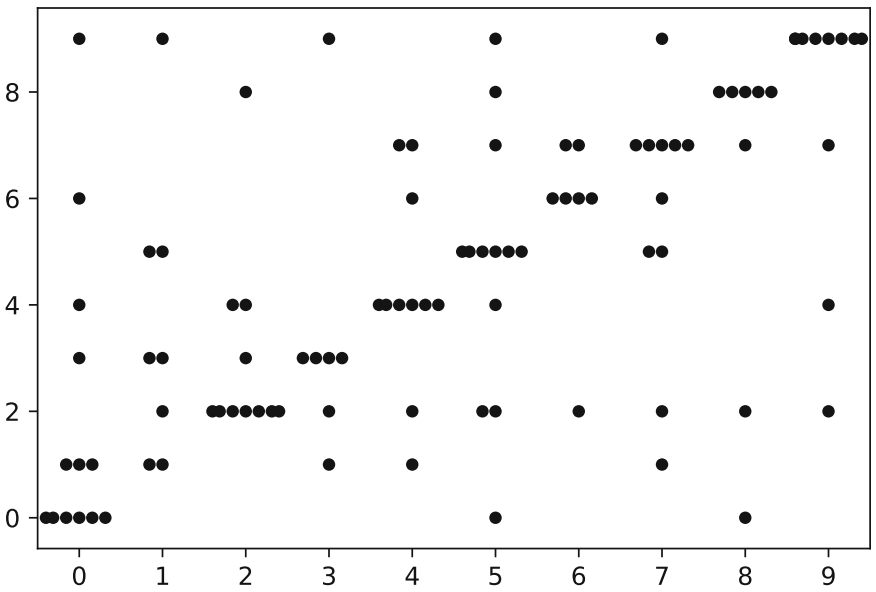

Fig. 9.4 Scatterplot of correlated data with discrete values, using `swarmplot()` to separate the values slightly for greater visual impact of overlapping points

```
In [12]: np.corrcoef(myvars)
```

```
Out[12]: array([[1.        , 0.5755755 , 0.23837089],
                [0.5755755 , 1.        , 0.48224687],
                [0.23837089, 0.48224687, 1.        ]])
```

In the correlation matrix, `xvar` correlates highly with `yvar` and less so with `zvar`, as expected, and `yvar` has strong correlation with `zvar` (using the rules of thumb from Sect. 4.5).

Using intuition, what would we expect the components to be from these data? First, there is shared variance across all three variables because they are positively correlated. So we expect to see one component that picks up that association of all three variables. After that, we expect to see a component that shows that `xvar` and `zvar` are more differentiated from one another than either is from `yvar`. That implies that `yvar` has a unique position in the dataset as the only variable to correlate highly with both of the others, so we expect one of the components to reflect this uniqueness of `yvar`.

Let's check the intuition. We use `PCA()` from the `sklearn.decomposition` library to perform PCA:

```
In [13]: from sklearn import decomposition
         my_pca = decomposition.PCA().fit(myvars.T)
```

We create a helper function to print some relevant statistics from the PCA:

```
In [14]: def pca_summary(pca, round_dig=3):
           '''Print a summary of the PCA fit'''
           return pd.DataFrame(
               [pca.explained_variance_,
                pca.explained_variance_ratio_,
                np.cumsum(pca.explained_variance_ratio_)],
               columns=['pc{}'.format(i) for i in
                        range(1, 1+len(pca.explained_variance_))],
               index=['variance', 'proportion of variance explained',
                      'cumulative proportion']
                      ).round(round_dig)
         pca_summary(my_pca)
```

```
Out[14]:                                      pc1    pc2    pc3
         variance                          16.473  7.050  3.042
         proportion of variance explained   0.620  0.265  0.114
         cumulative proportion              0.620  0.886  1.000
```

There are three components because we have three variables. The first component accounts for 62% of the explainable linear variance, while the second accounts for 27%, leaving 11% for the third component. How are those components related to the variables? We check the rotation matrix, using another helper function to format the output:

```
In [15]: def pca_components(pca, variable_names):
           '''Return loading of variables on specific components in the PCA'''
           return pd.DataFrame(pca.components_,
                               index=['pc{}'.format(i+1)
                                      for i in range(len(pca.components_))],
                               columns=variable_names).T
         my_pca_components = pca_components(my_pca, ['xvar', 'yvar', 'zvar'])
         my_pca_components.round(3)
```

```
Out[15]:        pc1    pc2    pc3
         xvar -0.544  0.637  0.545
         yvar -0.622  0.129 -0.772
         zvar -0.563 -0.760  0.326
```

Interpreting PCA rotation loadings is difficult because of the multivariate nature—factor analysis is a better procedure for interpretation, as we will see later in this chapter—but we examine the loadings here for illustration and comparison to our expectations. In component 1 (PC1) we see loading on all three variables as expected from their overall shared variance (the negative direction is not important; the key is that they are all in the same direction).

In component 2, we see that `xvar` and `zvar` are differentiated from one another as expected, with loadings in opposite directions. Finally, in component 3, we see residual variance that differentiates `yvar` from the other two variables and is consistent with our intuition about `yvar` being unique.

In addition to the loading matrix, we can use the PCA to compute scores for each of the principal components that express the underlying data in terms of its loadings on those components using the `transform()` method on the `PCA` object. The columns (`[:, 0]`, `[:, 1]`, and so forth) may be used to obtain the values of the components for each observation. We can use a small number of those columns in place of the original data to obtain a set of observations that captures much of the variation in the data.

A less obvious feature of PCA, but implicit in the definition, is that extracted PCA components are *uncorrelated* with one another, because otherwise there would be more linear variance that could have been captured. We see this in the transformed values returned for observations in a PCA model, where the correlations (off-diagonal values) are effectively zero (approximately 10^{-16} as shown in scientific notation):

```
In [16]: myvars_transformed = my_pca.transform(myvars.T)
         np.corrcoef(myvars_transformed.T)

Out[16]: array([[1.00000000e+00, 1.36938301e-16, 3.98753086e-16],
                [1.36938301e-16, 1.00000000e+00, 7.27332599e-17],
                [3.98753086e-16, 7.27332599e-17, 1.00000000e+00]])
```

9.2.2 Visualizing PCA

A good way to examine the results of PCA is to map the first few components, which allows us to visualize the data in a lower-dimensional space. A common visualization is a *biplot*, a two-dimensional plot of data points with respect to the first two PCA components, overlaid with a projection of the variables on the components. We can produce a biplot by first plotting the data points in a scatter plot:

```
In [17]: import matplotlib.pyplot as plt
         plt.scatter(x=myvars_transformed[:,0],
                     y=myvars_transformed[:,1],
                     color='k')
         plt.xlabel('PC1')
         plt.ylabel('PC2')
```

The result is Fig. 9.5. To that we add the vectors representing the loading of each variable, shown in Fig. 9.6.

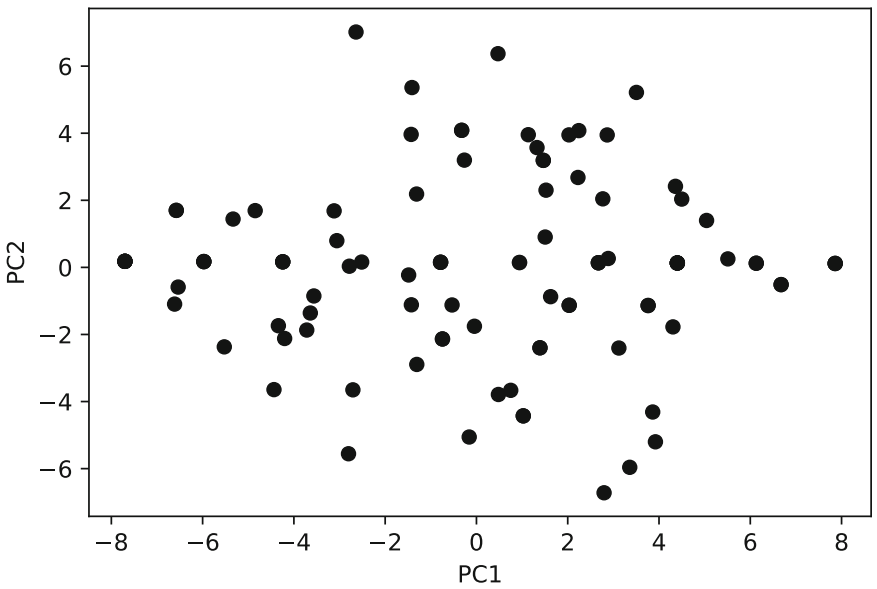

Fig. 9.5 A scatterplot showing data points plotted on the first two components

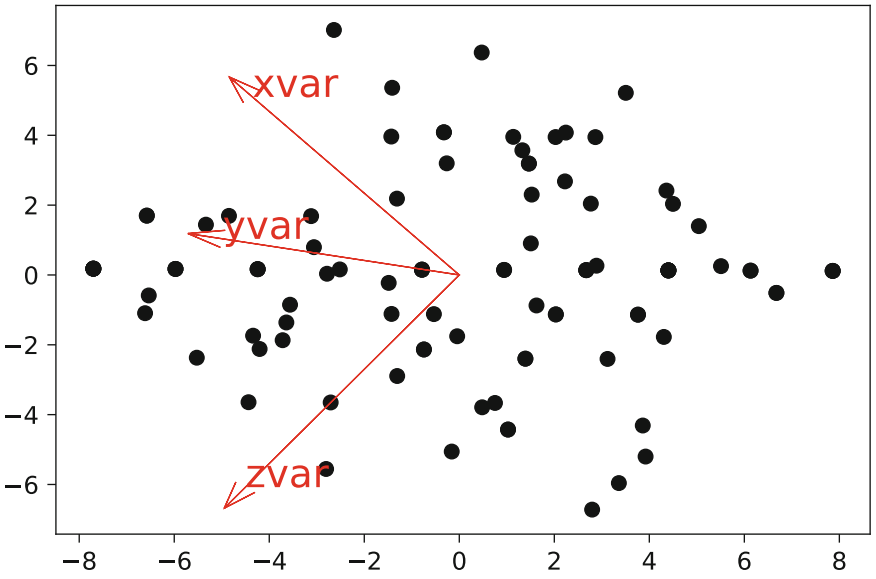

Fig. 9.6 A simple *biplot()* of a principal component analysis solution for the simple, constructed example, showing data points plotted on the first two components

```
In [18]: def plot_arrow_component(pca_components, variable, scale=1):
           '''Plot an arrow of component dimensions in PCA space'''
           plt.arrow(x=0, y=0,
                     dx=pca_components.loc[variable]['pc1'] * scale,
                     dy=pca_components.loc[variable]['pc2'] * scale,
                     color='r',
                     head_width=.5, overhang=1)
           plt.text(x=pca_components.loc[variable]['pc1'] * scale,
                    y=pca_components.loc[variable]['pc2'] * scale,
                    s=variable,
                    color='r',
                    fontsize=16)

           plt.scatter(x=myvars_transformed[:,0],
                       y=myvars_transformed[:,1],
                       color='k')

         for v in my_pca_components.index:
           plot_arrow_component(my_pca_components, v, 8)
```

Finally, we produce a function `biplot()`, which plots the datapoints and the vectors and additionally labels each data point with its index, as in Fig. 9.7. Such plots are especially helpful when there are a smaller number of points (as we will see below for brands) or when there are clusters (as we see in Chap. 10).

```
In [19]: def biplot(values_transformed, pca_components, label=[]):
          '''Create a biplot, a scatterplot of points in PCA space with arrows
          representing the loadings of each variable.
          Points can optionally be labelled'''
          scale = 1.2* np.max(values_transformed[:,1])
          plt.figure(figsize=(10, 10))
          for v in pca_components.index:
            plot_arrow_component(pca_components, v, scale)
          plt.scatter(x=values_transformed[:,0],
                      y=values_transformed[:,1],
```

```
                color='gray', s=4)
        if len(label) == values_transformed.shape[0]:
          for i, txt in enumerate(label):
            plt.text(s=txt,
                     x=values_transformed[i,0]+.01*scale,
                     y=values_transformed[i,1]+.01*scale,
                     fontsize=14)
        plt.xlabel('PC1')
        plt.ylabel('PC2')

In [20]: biplot(myvars_transformed, my_pca_components,
                label=range(myvars.shape[1]))
```

In Fig. 9.7, there are arrows that show the best fit of each of the variables on the principal components—a projection of the variables onto the 2-dimensional space of the first two PCA components, which explain a large part of the variation in the data. These are useful to inspect because the *direction* and *angle* of the arrows reflect the relationship of the variables; a closer angle indicates higher positive association, while the relative direction indicates positive or negative association of the variables.

In the present case, we see in the variable projections (arrows) that `yvar` is closely aligned with the first component (X axis). In the relationships among the variables themselves, we see that `xvar` and `zvar` are more associated with `yvar`, relative to the principal components, than either is with the other. Thus, this visually matches our interpretation of the correlation matrix and loadings above.

By plotting against principal components, a biplot benefits from the fact that components are uncorrelated; this helps to disperse data on the chart because the x- and y-axes are independent. When there are several components that account for substantial variance, it is also useful to plot components beyond the first and second.

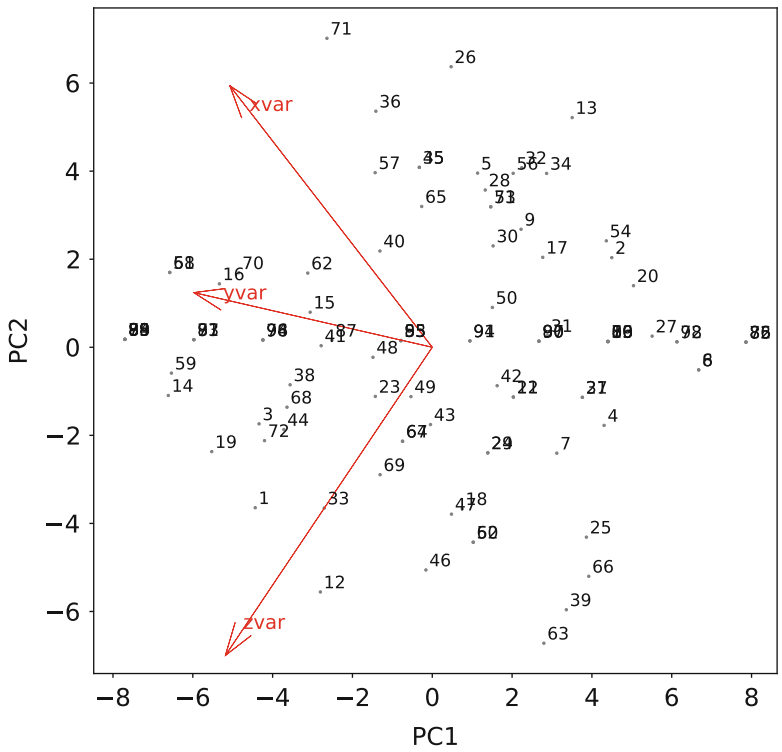

Fig. 9.7 A *biplot()* with point labels, generated from our `biplot()` function

9.2.3 PCA for Brand Ratings

Let's look at the principal components for the brand rating data (refer to Sect. 9.1 above if you need to load the data). We find the components with `PCA.fit()`, selecting just the rating columns, for which we will make two new variables, `brand_rating_names` for the rating names and `brand_rating_sc_vals` for the values (as we will be using these several more times):

```
In [21]: brand_rating_names = brand_ratings_sc.columns[:-1]
         brand_ratings_sc_vals = brand_ratings_sc[brand_rating_names]
         brand_pca = decomposition.PCA().fit(brand_ratings_sc_vals)
```

We can use our `pca_summary()` function defined in the previous section to see the variance covered by each component:

```
In [22]: pca_summary(brand_pca)
```

```
Out[22]:                                      pc1    pc2    pc3    pc4    pc5  \
         variance                           2.982  2.099  1.080  0.728  0.638
         proportion of variance explained   0.331  0.233  0.120  0.081  0.071
         cumulative proportion              0.331  0.564  0.684  0.765  0.836

                                               pc6    pc7    pc8    pc9
         variance                            0.535  0.390  0.312  0.243
         proportion of variance explained    0.059  0.043  0.035  0.027
         cumulative proportion               0.895  0.938  0.973  1.000
```

An important plot when analyzing a PCA is a *scree plot*, which shows the successive proportion of additional variance that each component adds. This corresponds to the `PCA.explained_variance_` parameter, which we can plot:

```
In [23]: plt.plot(1+np.arange(len(brand_pca.explained_variance_)),
                  brand_pca.explained_variance_, 'o-')
         plt.xlabel('Component')
         plt.ylabel('Variance')
```

The result is Fig. 9.8. A scree plot is often interpreted as indicating where additional components are not worth the complexity; this occurs where the line has an *elbow*, a kink in the angle of bending, a somewhat subjective determination.

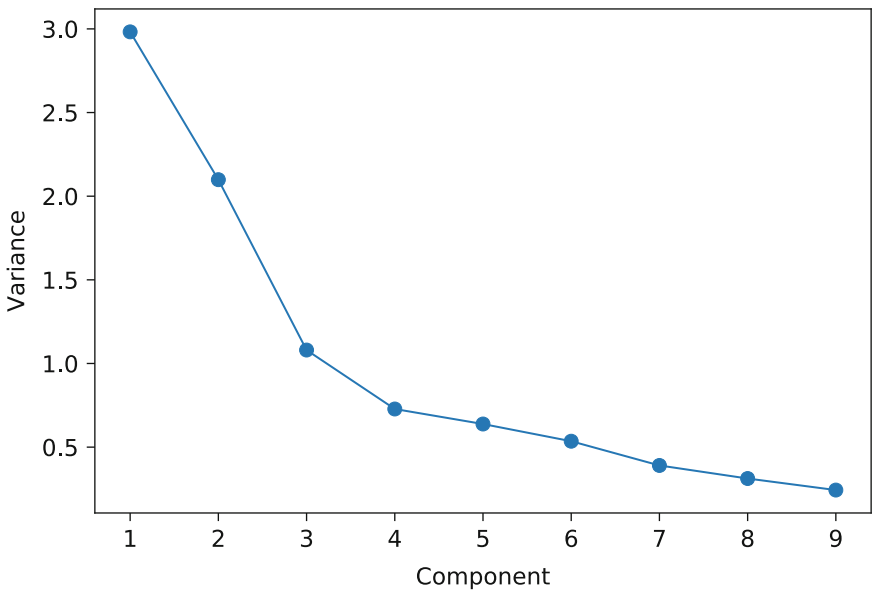

Fig. 9.8 A scree plot of a PCA solution shows the successive variance accounted by each component. For the brand rating data, the proportion largely levels out after the third component

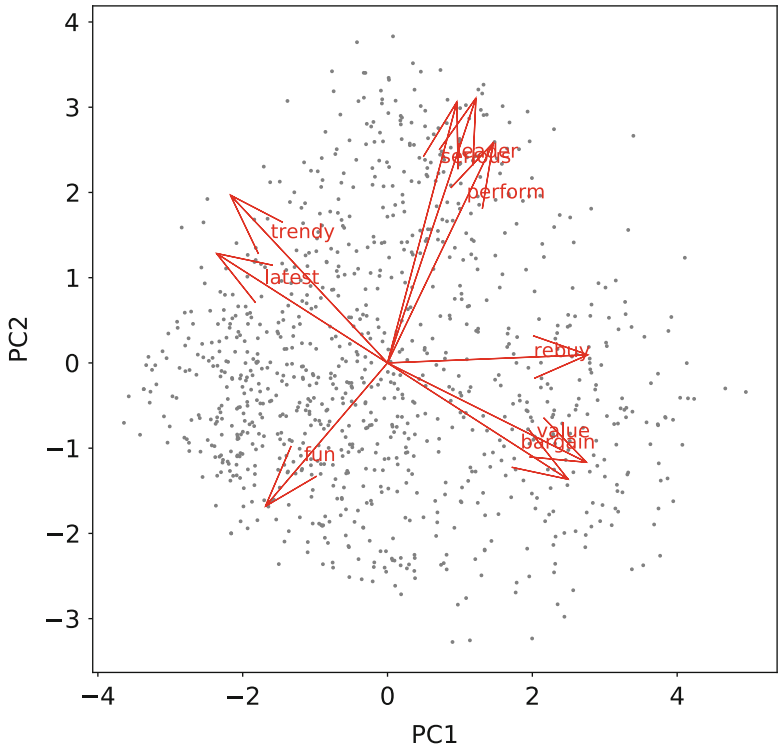

Fig. 9.9 A biplot of an initial attempt at principal component analysis for consumer brand ratings. Although we see adjective groupings on the variable loading arrows in red, and gain some insight into the areas where ratings cluster (as dense areas of observation points), the chart would be more useful if the data were first aggregated by brand

In Fig. 9.8, the elbow occurs at either component 3 or 4, depending on interpretation; and this suggests that the first two or three components explain most of the variation in the observed brand ratings.

We can use several of our helper functions from the previous section to plot a biplot of the first two principal components reveals how the rating adjectives are associated:

```
In [24]: brand_ratings_sc_trans = brand_pca.transform(brand_ratings_sc_vals)
         brand_pca_components = pca_components(brand_pca, brand_rating_names)
         biplot(brand_ratings_sc_trans, brand_pca_components)
```

We see the result in Fig. 9.9, where adjectives map in four regions: category leadership ("serious," "leader," and "perform" in the upper right), value ("rebuy," "value," and "bargain"), trendiness ("trendy" and "latest"), and finally "fun" on its own.

But there is a problem: the plot of individual respondents' ratings is too dense and it does not tell us about the brand positions! A better solution is to construct a biplot that shows the *aggregated* ratings by brand.

9.2.4 Perceptual Map of the Brands

First we compile the mean scaled rating of each adjective by brand as we found above using `groupby()` (see Sect. 9.1).

```
In [25]: brand_means_sc = brand_ratings_sc.groupby('brand').mean()
         brand_means_sc.head()
```

```
Out[25]:         perform    leader    latest       fun   serious   bargain  \
         brand
         a     -0.886362 -0.528168  0.411179  0.656974 -0.919400  0.214203
         b      0.931336  1.071294  0.726470 -0.972701  1.183733  0.041640
         c      0.650249  1.163350 -0.102388 -0.845098  1.223346 -0.607347
         d     -0.680231 -0.593373  0.352671  0.186665 -0.692521 -0.881197
```

```
e     -0.564673  0.192933  0.456685  0.296039  0.042135  0.551826

          value     trendy     rebuy
brand
a      0.184785 -0.525407 -0.596465
b      0.151415  0.740679  0.237092
c     -0.440898  0.025541 -0.132504
d     -0.933102  0.737030 -0.494236
e      0.418373  0.138649  0.036566
```

Prior to creating the biplot, we rescale the data; even though the raw data were already rescaled, the aggregated means have a somewhat different scale than the standardized data itself.

Note, that we manually scaled the data using `mean()` and `std()`. Since the data were held in a dataframe with a relevant index, this simplified the process. Using `sklearn.preprocessing.scale` would have returned a NumPy matrix, from which we could have created a dataframe, but by doing it manually we maintained those labels.

A biplot of the PCA-transformed mean ratings gives an interpretable *perceptual map*, showing where the brands are placed with respect to the first two principal components:

```
In [26]: brand_means_sc = (
           ((brand_means_sc - brand_means_sc.mean()) / brand_means_sc.std()))
         brand_means_sc_transformed = brand_pca.transform(brand_means_sc)
         biplot(brand_means_sc_transformed, brand_pca_components,
                label=brand_means.index)
```

The result is Fig. 9.10.

What does the map tell us? First we interpret the adjective clusters and relationships and see four areas with well-differentiated sets of adjectives and brands that are positioned in proximity. Brands *f* and *g* are high on "value," for instance, while *a* and *j* are relatively high on "fun," which is opposite in direction from leadership adjectives ("leader" and "serious").

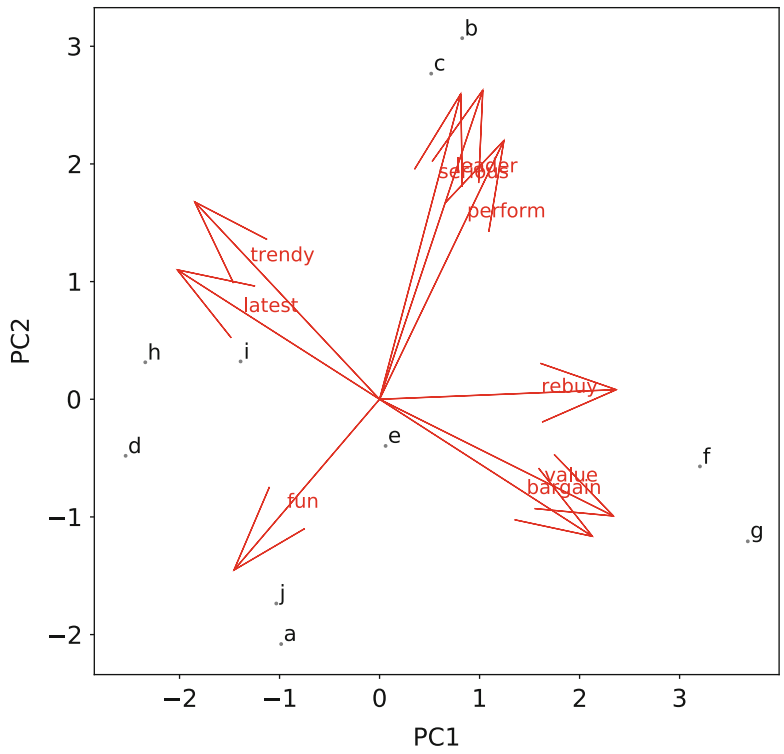

Fig. 9.10 A perceptual map of consumer brands with `biplot()` for aggregate mean rating by brand. This shows components almost identical to those in Fig. 9.9 but the mean brand positions are clear

With such a map, one might form questions and then refer to the underlying data to answer them. For instance, suppose that you are the brand manager for brand *e*. What does the map tell you? For one thing, your brand is in the center and thus appears not to be well-differentiated on any of the dimensions. That could be good or bad, depending on your strategic goals. If your goal is to be a safe brand that appeals to many consumers, then a relatively undifferentiated position like *e* could be desirable. On the other hand, if you wish your brand to have a strong, differentiated perception, this finding would be unwanted (but important to know).

What should you do about the position of your brand *e*? Again, it depends on the strategic goals. If you wish to increase differentiation, one possibility would be to take action to shift your brand in some direction on the map. Suppose you wanted to move in the direction of brand *c*. You could look at the specific differences from *c* in the data:

```
In [27]: brand_means_sc.loc['c'] - brand_means_sc.loc['e']

Out[27]: perform    1.775362
         leader     1.440484
         latest    -0.774450
         fun       -1.886670
         serious    1.544750
         bargain   -1.811159
         value     -1.131735
         trendy    -0.151604
         rebuy     -0.212361
         dtype: float64
```

This shows you that *e* is relatively stronger than *c* on "bargain" and "fun", which suggests dialing down messaging or other attributes that reinforce those (assuming, of course, that you truly want to move in the direction of *c*). Similarly, *c* is stronger on "perform" and "serious," so those could be aspects of the product or message for *e* to strengthen.

Another option would be *not* to follow another brand but to aim for differentiated space where no brand is positioned. In Fig. 9.10, there is a large gap between the group *b* and *c* on the bottom of the chart, versus *f* and *g* on the upper right. This area might be described as the "value leader" area or similar.

How do we find out how to position there? Let's assume that the gap reflects approximately the average of those four brands (see Sect. 9.2.5 for some of the risks with this assumption). We can find that average as the `mean` of the brands' rows, and then take the difference of *e* from that average:

```
In [28]: brand_means_sc.loc[['b','c','f','g']].mean(axis=0) - brand_means_sc.loc['e']

Out[28]: perform    1.717172
         leader     0.580749
         latest    -1.299004
         fun       -1.544598
         serious    0.750005
         bargain   -0.391245
         value      0.104383
         trendy    -0.629646
         rebuy      0.840802
         dtype: float64
```

This suggests that brand *e* could target the gap by increasing its emphasis on performance while reducing emphasis on "latest" and "fun."

To summarize, when you wish to compare several brands across many dimensions, it can be helpful to focus on just the first two or three principal components that explain variation in the data. You can select how many components to focus on using a scree plot, which shows how much variation in the data is explained by each principal component. A perceptual map plots the brands on the first two principal components, revealing how the observations related to the underlying dimensions (the components).

PCA may be performed using survey ratings of the brands (as we have done here) or with objective data such as price and physical measurements, or with a combination of the two. In any case, when you are confronted with multidimensional data on brands or products, PCA visualization is a useful tool for understanding how they differ from one another in the market.

9.2.5 Cautions with Perceptual Maps

There are three important caveats in interpreting perceptual maps. First, you must choose the level and type of aggregation carefully. We demonstrated the maps using mean rating by brand, but depending on the data and question at hand, it might be more suitable to use median (for ordinal data) or even modal response (for categorical data). You should check that the dimensions are similar for the full data and aggregated data before interpreting aggregate maps. You can do this by examining the variable positions and relationships in biplots of both aggregated data (such as means) and raw data (or a random subset of it), as we did above.

Second, the relationships are strictly relative to the product category and the brands and adjectives that are tested. In a different product category, or with different brands, adjectives such as "fun" and "leader" could have a very different relationship. Sometimes simply adding or dropping a brand can change the resulting map significantly because the positions are relative. In other words, if a new brand enters the market (or one's analysis), the other positions may change substantially. One must also be confident that all of the key perceptions (adjectives, in this example) have been assessed. One way to assess sensitivity here is to run PCA and biplot on a few different samples from your data, such as 80% of your observations, and perhaps dropping an adjective each time. If the maps are similar across those samples, you may feel more confident in their stability.

Third, it is frequently misunderstood that the positions of brands in such a map depend on their relative positioning in terms of the principal components, which are constructed composites of all dimensions. This means that *the strength of a brand on a single adjective cannot be read directly from the chart*. For instance, in Fig. 9.10, it might appear that brands *b* and *c* are weaker than *d*, *h*, and *i* on "latest" but are similar to one another. In fact, *b* is the single strongest brand on "latest" while *c* is weak on that adjective. Overall, *b* and *c* are quite similar to one another in terms of their scores on the two components that aggregate all of the variables (adjectives), but they are not necessarily similar on any single variable. When we use PCA to focus on the first one or two dimensions in the data, we are looking at the largest-magnitude similarities, and that may obscure smaller differences that do not show up strongly in the first one or two dimensions.

This last point is a common area of confusion with analysts and stakeholders who want to read adjective positions directly from a biplot. We recommend to explain that positions are not absolute but are *relative*. We often explain positions with language such as, "compared to its position on other attributes, brand X is *relatively* differentiated by perceptions of strength (or weakness) on such-and-such attribute."

Despite these caveats, perceptual maps can be a valuable tool. We use them primarily to form hypotheses and to provide material to inform strategic analyses of brand and product positioning. If they are used in that way—rather than as absolute assessments of position—they can contribute to engaging discussions about position and potential strategy.

Although we illustrated PCA with brand position, the same kind of analysis could be performed for product ratings, position of consumer segments, ratings of political candidates, evaluations of advertisements, or any other area where you have metric data on multiple dimensions that is aggregated for a modest number of discrete entities of interest.

9.3 Exploratory Factor Analysis

Exploratory factor analysis (EFA) is a family of techniques to assess the relationship of *constructs* (concepts) in surveys and psychological assessments. Factors are regarded as *latent variables* that cannot be observed directly, but are imperfectly assessed through their relationship to other variables.

In psychometrics, canonical examples of factors occur in psychological and educational testing. For example, "intelligence," "knowledge of mathematics," and "anxiety" are all abstract concepts (constructs) that are not directly observable in themselves. Instead, they are observed empirically through multiple behaviors, each one of which is an imperfect indicator of the presumed underlying latent variable. These observed values are known as *manifest variables* and include indicators such as test scores, survey responses, and other empirical behaviors. Exploratory factor analysis attempts to find the degree to which latent, composite *factors* account for the observed variance of those manifest variables.

In marketing, we often observe a large number of variables that we believe should be related to a smaller set of underlying constructs. For example, we cannot directly observe *customer satisfaction* but we might observe responses on a survey that asks about different aspects of a customer's experience, jointly representing different facets of the underlying construct *satisfaction*. Similarly, we cannot directly observe *purchase intent*, *price sensitivity*, or *category involvement* but we can observe multiple behaviors that are related to them.

In this section, we use EFA to examine respondents' attitudes about brands, using the brand rating data from above (Sect. 9.1) and to uncover the latent dimensions in the data. Then we assess the brands in terms of those estimated latent factors.

9.3.1 Basic EFA Concepts

The result of EFA is similar to PCA: a matrix of factors (similar to PCA components) and their relationship to the original variables (*loadings* of the factors on the variables). Unlike PCA, EFA attempts to find solutions that are maximally *interpretable* in terms of the manifest variables. In general, it attempts to find solutions in which a small number of loadings for each factor are very high, while other loadings for that factor are low. When this is possible, that factor can be interpreted in terms of that small set of variables.

To accomplish this, EFA uses *rotations* that start with an uncorrelated (*orthogonal*) mathematical solution and then mathematically alter the solution to explain an identical amount of variance but with different loadings on the original variables. There are many such rotations available, and they typically share the goals of maximizing the loadings on a few variables while making factors as distinct as possible from one another.

Instead of reviewing that mathematically (see Mulaik 2009), let's consider a loose analogy. One might think about EFA in terms of a pizza topped with large items such as tomato slices and mushrooms that will be cut into a certain number of slices. The pizza could be rotated and cut in an infinite number of ways that are all mathematically equivalent insofar as they divide up the same underlying structure.

However, some rotations are more useful than others because they fall in-between the large items rather than dividing them. When this occurs, one might have a "tomato slice," a "mushroom slice," a "half-and-half tomato and mushroom slice," and so forth. By rotating and cutting differently, one makes the underlying substance more interpretable relative to one's goals (such as having differentiated pizza slices). No rotation is inherently better or worse, but some are more useful than others. Similarly, the manifest variables in EFA can be sliced in many ways according to one's goals for interpreting the latent factors. For example, in our dataset, a rotation that associates "value" and "bargain" together might be more interpretable than one that placed them apart, as we can imagine that they both represent a similar underlying concept. We will see how this works in Sect. 9.3.3.

Because EFA produces results that are interpretable in terms of the original variables, an analyst may be able to interpret and act on the results in ways that would be difficult with PCA. For instance, EFA can be used to refine a survey by keeping items with high loading on factors of interest while cutting items that do not load highly. EFA is also useful to investigate whether a survey's items actually go together in a way that is consistent with expectations.

For example, if we have a 10-item survey that is supposed to assess the single construct *customer satisfaction*, it is important to know whether those items in fact go together in a way that can be interpreted as a single factor, or whether they instead reflect multiple dimensions that we might not have considered. Before interpreting multiple items as assessing a single concept, one might wish to test that it is appropriate to do so. In this chapter, we use EFA to investigate such structure.

EFA serves as a data reduction technique in three broad senses:

1. In the technical sense of dimensional reduction, we can use *factor scores* instead of a larger set of items. For instance, if we are assessing satisfaction, we could use a single satisfaction score instead of several separate items. (In Sect. 8.1.2 we review how this is also useful when observations are correlated.)
2. We can reduce uncertainty. If we believe *satisfaction* is imperfectly manifest in several measures, the combination of those will have less noise than the set of individual items.
3. We might also reduce data collection by focusing on items that are known to have high contribution to factors of interest. If we discover that some items are not important for a factor of interest, we can discard them from data collection efforts.

In this chapter we use the brand rating data to ask the following questions: How many latent factors are there? How do the survey items map to the factors? How are the brands positioned on the factors? What are the respondents' factor scores?

9.3.2 Finding an EFA Solution

The first step in exploratory factor analysis is to determine the number of factors to estimate. There are various ways to do this, and two traditional methods are to use a scree plot (Sect. 9.2.3), and to retain factors where the *eigenvalue* (a metric

for proportion of variance explained) is greater than 1.0. An eigenvalue of 1.0 corresponds to the amount of variance that might be attributed to a single independent variable; a factor that captures less variance than such an item may be considered relatively uninteresting.

As we saw in Sect. 9.2, a scree plot of the brand rating data suggests two or three components. We can also examine the eigenvalues using `numpy.linalg.eig()` on a correlation matrix:

```
In [29]: np.linalg.eig(np.corrcoef(brand_ratings_sc_vals.T))[0]

Out[29]: array([2.97929556, 2.09655168, 1.07925487, 0.72721099, 0.63754592,
                0.53484323, 0.39010444, 0.24314689, 0.31204642])
```

The first three eigenvalues are greater than 1.0, although barely so for the third value. This again suggests three or possibly two factors.

The final choice of a model depends on whether it is useful. For EFA, a best practice is to check a few factor solutions, including the ones suggested by the scree and eigenvalue results. Thus, we test a 3-factor solution and a 2-factor solution to see which one is more useful.

`sklearn` has a factor analysis module `sklearn.decomposition.FactorAnalysis()`. However, it is a bit limited in functionality, so we will use the `factor_analyzer` package (Biggs 2017), which we can install using `pip` if necessary:

```
In [30]: !pip install factor_analyzer
```

We can then instantiate a `FactorAnalyzer()` object and analyze the scaled brand ratings:

```
In [31]: import factor_analyzer

         fa = factor_analyzer.FactorAnalyzer(n_factors=2, rotation='varimax')
         fa.fit(brand_ratings_sc_vals)
         pd.DataFrame(fa.loadings_, index=brand_rating_names).round(2)
```

Out[31]:

	0	1
perform	0.09	0.60
leader	-0.02	0.81
latest	-0.59	-0.04
fun	-0.19	-0.39
serious	-0.07	0.68
bargain	0.69	0.05
value	0.78	0.11
trendy	-0.65	0.10
rebuy	0.60	0.33

In the 2-factor solution, factor 0 loads strongly on “bargain” and “value,” and therefore might be interpreted as a “value” factor while factor 1 loads on “leader” and “serious” and thus might be regarded as a “category leader” factor.

This is not a bad interpretation, but let’s compare it to a 3-factor solution:

```
In [32]: fa = factor_analyzer.FactorAnalyzer(n_factors=3, rotation='varimax')
         fa.fit(brand_ratings_sc_vals)
         pd.DataFrame(fa.loadings_, index=brand_rating_names).round(2)
```

Out[32]:

	0	1	2
perform	0.07	0.60	-0.06
leader	0.06	0.80	0.10
latest	-0.16	-0.08	0.98
fun	-0.07	-0.41	0.21
serious	-0.01	0.68	0.08
bargain	0.84	-0.00	-0.11
value	0.85	0.08	-0.21
trendy	-0.35	0.08	0.59
rebuy	0.50	0.32	-0.30

The 3-factor solution retains the "value" and "leader" factors (factors 0 and 1 in the output) and adds a clear "latest" factor (factor 2) that loads strongly on "latest" and "trendy." This adds an easily interpretable concept to our understanding of the data. It also aligns with the bulk of suggestions from the scree and eigen tests, and fits well with the perceptual maps we saw in Sect. 9.2.4, where those adjectives were in a differentiated space. So we regard the 3-factor model as superior to the 2-factor model because the factors are more interpretable. Solutions may be formally compared using confirmatory factor analysis (CFA), which we do not cover in this book (see Chapman and Feit 2019, Chap. 10).

9.3.3 EFA Rotations

As we described earlier, a factor analysis solution can be rotated to have new loadings that account for the same proportion of variance. Although a full consideration of rotations is out of scope for this book, there is one issue worth considering in any EFA: do you wish to allow the factors to be *correlated* with one another or not?

You might think that one should let the data decide. However, the question of whether to allow correlated factors is less a question about the *data* than it is about your *concept* of the underlying latent factors. Do you think the factors should be conceptually independent, or does it make more sense to consider them to be related? (To return to the pizza analogy, we could choose to slice our pizza such that mushrooms tend to appear alongside tomatoes, or we could cut it such that the two tend to be separated.) An EFA rotation can be obtained under either assumption.

The default in general is to find factors that have zero correlation (using a *varimax* rotation). In case you're wondering how this differs from PCA, it differs mathematically because EFA finds latent variables that may be observed with error (see Mulaik 2009) whereas PCA simply recomputes transformations of the observed data. In other words, EFA focuses on the underlying latent dimensions, whereas PCA focuses on transforming the dimensionality of the data.

Returning to our present data, we might judge that *value* and *leader* are reasonably expected to be related; in many categories, the leader can command a price premium, and thus we might expect those two latent constructs to be negatively correlated rather than independent of one another. This suggests that we could allow correlated factors in our solution. This is known as an *oblique* rotation ("oblique" because the dimensional axes are not perpendicular but are skewed by the correlation between factors).

A common oblique rotation is the "oblimin" rotation. We add that to our 3-factor model with `rotation="oblimin"`:

```
In [33]: fa = factor_analyzer.FactorAnalyzer(n_factors=3, rotation='oblimin')
         fa.fit(brand_ratings_sc_vals)
         fa_loadings_df = pd.DataFrame(fa.loadings_,
                                       index=brand_rating_names)
         fa_loadings_df.round(2)
```

```
Out[33]:              0     1     2
         perform   0.01  0.60 -0.09
         leader    0.02  0.81  0.07
         latest    0.03 -0.00  1.01
         fun       0.00 -0.39  0.24
         serious  -0.05  0.68  0.03
         bargain   0.88 -0.05  0.07
         value     0.86  0.03 -0.04
         trendy   -0.26  0.14  0.54
         rebuy     0.45  0.28 -0.22
```

When we compare this oblimin result to the default varimax rotation above, we see the loadings are slightly different for the relationships of the factors to the adjectives. However, the loadings are similar enough in this case that there is no substantial change in how we would interpret the factors. There are still factors for "value," "leader," and "latest."

We can check the factor correlation matrix showing the relationships between the estimated latent factors:

```
In [34]: np.corrcoef(fa.transform(brand_ratings_sc_vals).T)
```

```
Out[34]: array([[ 1.        ,  0.12904599, -0.41410012],
                [ 0.12904599,  1.        , -0.04888392],
                [-0.41410012, -0.04888392,  1.        ]])
```

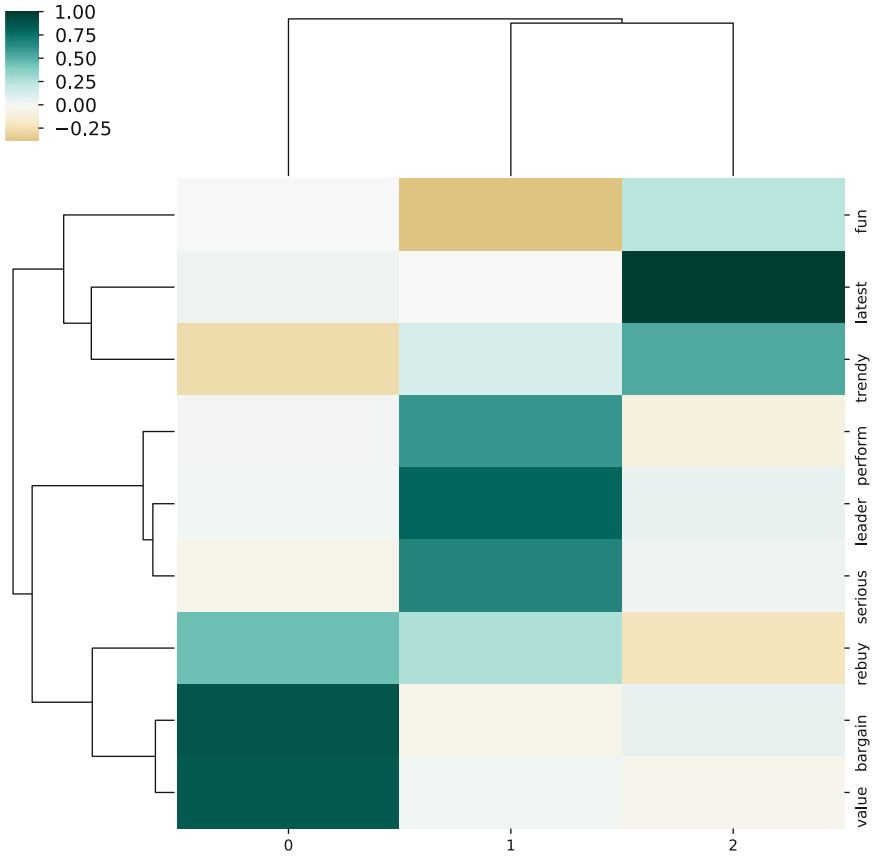

Fig. 9.11 A clustermap of item-factor loadings

Factor 1 (value) is negatively correlated with Factor 3 (latest), $r = -0.41$, and is essentially uncorrelated with Factor 2 (leader), $r = 0.13$.

The negative correlation between factors 1 and 3 is consistent with our theory that brands that are value brands are less likely to be trendy, and thus we think this is a more interpretable result. However, in other cases a correlated rotation may or may not be a better solution than an orthogonal one; that is largely an issue to be decided on the basis of domain knowledge and interpretive utility rather than statistics.

In the output above, the item-to-factor loadings are displayed. In the returned model object, those are present as the `loadings` parameter. We can then visualize item-factor relationships with a clustermap of `loadings`:

```
In [35]: sns.clustermap(fa_loadings_df, cmap=cm.BrBG, center=0)
```

The result is Fig. 9.11, which shows a distinct separation of items into three factors, which are roughly interpretable as *value*, *leader*, and *latest*. Note that the item `rebuy`, which reflects stated intention to repurchase, loads on both Factor1 (*value*) and Factor2 (*leader*). This suggests that in our simulated data, consumers say they would rebuy a brand for either reason, because it is a good value or because it is a leader.

Overall, the result of the EFA for this dataset is that instead of using nine distinct variables, we might instead represent the data with three underlying latent factors. We have seen that each factor maps to 2–4 of the manifest variables. However, this only tells us about the relationships of the rating variables among themselves in our data; in the next section, we use the estimated factor scores to learn about the *brands*.

9.3.4 Using Factor Scores for Brands

In addition to estimating the factor structure, EFA will also estimate latent factor *scores* for each observation. In the present case, this gives us the best estimates of each respondent's latent ratings for the "value," "leader," and "latest" factors. We

can then use the factor scores to determine brands' positions on the factors. Interpreting factors eliminates the separate dimensions associated with the manifest variables, allowing us to concentrate on a smaller, more reliable set of dimensions that map to theoretical constructs instead of individual items.

Factor scores are calculated from the `FactorAnalyzer()` object using the `transform()` method, which we can store as a separate dataframe:

```
In [36]: fa = factor_analyzer.FactorAnalyzer(n_factors=3, rotation='oblimin')
         brand_ratings_fa_trans = fa.fit_transform(brand_ratings_sc_vals)
         brand_rating_fa_scores = pd.DataFrame(brand_ratings_fa_trans)
         brand_rating_fa_scores['brand'] = brand_ratings_sc.brand
         brand_rating_fa_scores.head()
```

```
Out[36]:           0         1         2 brand
         0  1.388590 -0.491354  0.531693     a
         1 -1.188916 -1.352280 -0.658905     a
         2  1.038597 -0.801256 -0.372207     a
         3  0.037803 -0.318029  1.190962     a
         4  1.688281 -1.525753 -0.453958     a
```

The result is an estimated score for each respondent on each factor and brand. If we wish to investigate individual-level correlates of the factors, such as their relationship to demographics or purchase behavior, we could use these estimates of factor scores. This can be very helpful in analyses such as regression and segmentation because it reduces the model complexity (number of dimensions) and uses more reliable estimates (factor scores that reflect several manifest variables). Instead of nine items, we have three factors.

To find the overall position for a brand, we aggregate the individual scores by brand as usual using `groupby()`:

```
In [37]: brand_rating_fa_mean = brand_rating_fa_scores.groupby('brand').mean()
         brand_rating_fa_mean.columns = ['Value', 'Leader', 'Latest']
         brand_rating_fa_mean.round(3)
```

```
Out[37]:        Value  Leader  Latest
         brand
         a      0.147  -0.863   0.388
         b      0.067   1.205   0.710
         c     -0.492   1.120  -0.077
         d     -0.921  -0.625   0.368
         e      0.416  -0.035   0.437
         f      1.048   0.406  -1.265
         g      1.236   0.086  -1.326
         h     -0.804  -0.271   0.528
         i     -0.555  -0.169   0.388
         j     -0.142  -0.854  -0.150
```

Finally, a clustermap graphs the scores by brand:

```
In [38]: sns.clustermap(brand_rating_fa_mean, cmap=cm.BrBG, center=0)
```

The result is Fig. 9.12. When we compare this to the chart of brand by adjective in Fig. 9.3, we see that the chart of factor scores is significantly simpler than the full adjective matrix. The brand similarities are evident again in the factor scores, for instance that *f* and *g* are similar, as are *b* and *c*, and so forth.

We conclude that EFA is a valuable way to examine the underlying structure and relationship of variables. When items are related to underlying constructs, EFA reduces data complexity by aggregating variables to create simpler, more interpretable latent variables.

In this exposition, we have only explored a small number of the possibilities for factor analysis; to learn more, see Sect. 9.5.

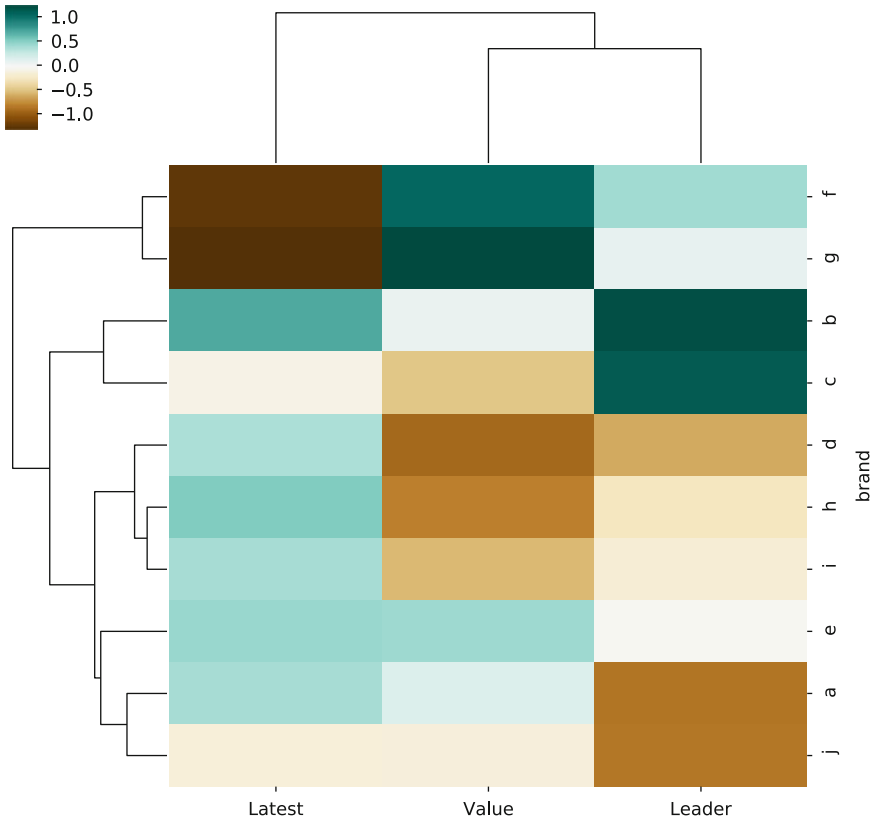

Fig. 9.12 A heatmap of the latent factor scores for consumer brand ratings, by brand

9.4 Multidimensional Scaling

Multidimensional scaling (MDS) is a family of procedures that can also be used to find lower-dimensional representations of data. Instead of extracting underlying components or latent factors, MDS works instead with a *distance matrix* (also known as a *similarity matrix*). MDS attempts to find a lower-dimensional map that best preserves all the observed similarities between items.

The `sklearn.manifold.MDS` module works with vectors directly: it calculates pairwise Euclidean distances and then looks for lower-dimensional representations:

```
In [39]: from sklearn import manifold

         np.random.seed(889783)
         brand_mds = manifold.MDS().fit_transform(brand_means)
         brand_mds

Out[39]: array([[ 0.59217926,  5.19146726],
                [-2.74412002, -6.30675543],
                [-0.51645595, -5.64921129],
                [-4.33444294,  3.51765049],
                [ 0.69182752,  0.80286252],
                [ 6.14100233, -2.84581818],
                [ 8.40903503, -0.33459353],
                [-4.67731306,  1.14429619],
                [-3.16204417,  0.12390567],
                [-0.399668  ,  4.35619632]])
```

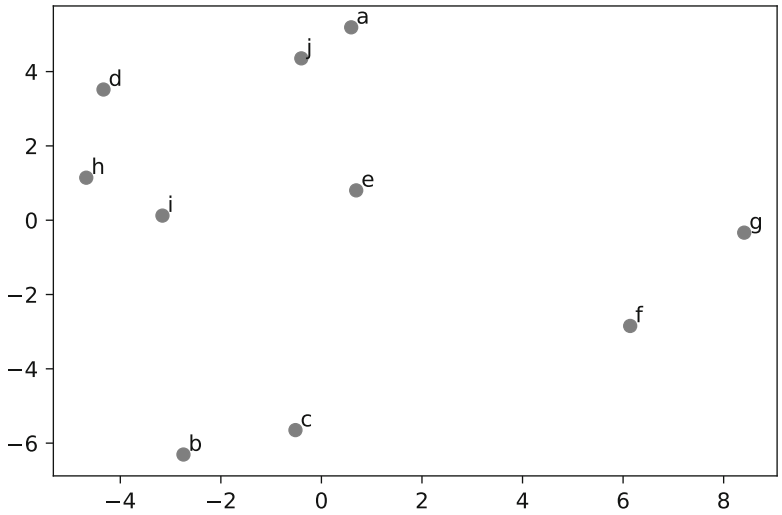

Fig. 9.13 A metric multidimensional scaling chart for mean brand rating, using `sklearn.manifold.MDS()`. The brand positions are quite similar to those seen in the `biplot()` in Fig. 9.10

The result of `MDS.fit_transform()` is a list of X and Y dimensions indicating 2-dimensional estimated plot coordinates for entities (in this case, brands). We see the plot locations for brands *a* and *b* in the output above. Given those coordinates, we can simply `scatter()` the values and label them:

```
In [40]: plt.scatter(x=brand_mds[:,0],
                     y=brand_mds[:,1],
                     color='grey')
         for i,p in enumerate(brand_mds):
           plt.annotate(s=brand_means.index[i], xy=p+.1)
```

In this code, `plot(plt.annotate())` adds the text brand annotation to each point (as before in our `biplot()` code. The result is Fig. 9.13. The relative brand positions are grouped nearly identically to what we saw in the perceptual map in Fig. 9.10.

9.4.1 Non-metric MDS

For *non-metric* data such as rankings or categorical variables, you simply pass the `metric=False` argument to the sklearn.manifold.MDS() instantiation.

For purposes of illustration, let's convert the mean ratings to rankings instead of raw values; this will be non-metric, ordinal data. We apply `argsort()` to the columns using `apply()` which codes each resulting column as its rank rather than overall rating:

```
In [41]: brand_ranks = brand_means.apply(lambda col: col.argsort().argsort())
         brand_ranks
```

```
Out[41]:        perform  leader  latest  fun  serious  bargain  value  \
         brand
         a            0       2       6    8        0        6      6
         b            9       8       9    0        8        5      5
         c            7       9       3    1        9        2      3
         d            1       1       4    4        1        0      0
         e            3       6       7    6        6        7      7
         f            4       7       1    3        7        8      8
         g            8       5       0    2        3        9      9
         h            5       4       8    9        5        1      2
```

```
i            6       3       5    7       4       3       1
j            2       0       2    5       2       4       4

       trendy  rebuy
brand
a           2      2
b           8      7
c           4      5
d           7      3
e           5      6
f           1      8
g           0      9
h           9      1
i           6      4
j           3      0
```

We then pass that brand rank matrix to `MDS()` and plot the result:

```
In [42]: brand_mds_nonmetric = manifold.MDS(metric=False).fit_transform(brand_ranks)

In [43]: plt.scatter(x=brand_mds_nonmetric[:,0],
                     y=brand_mds_nonmetric[:,1],
                     color='grey')
         for i,p in enumerate(brand_mds_nonmetric):
           plt.annotate(s=brand_means.index[i], xy=p+.01)
```

The resulting chart is shown in Fig. 9.14. Compared to Fig. 9.13, we see that brand positions in the non-metric solution are considerably different, but most of the nearest neighbors of brands are largely consistent with the exception of brands *b* and *c*, which are separated quite a bit more than in the metric solution. (This occurs because the rank-order procedure loses some of the information that is present in the original metric data solution, resulting in a slightly different map.)

We generally recommend principal component analysis as a more informative procedure than multidimensional scaling for typical metric or near-metric (e.g., survey Likert scale) data. However, PCA will not work with non-metric data. In those cases, multidimensional scaling is a valuable alternative.

MDS may be of particular interest when handling text data such as consumers' feedback, comments, and online product reviews, where text frequencies can be converted to distance scores. For example, if you are interested in similarities between brands in online reviews, you could count how many times various pairs of brands occur together in consumers' postings.

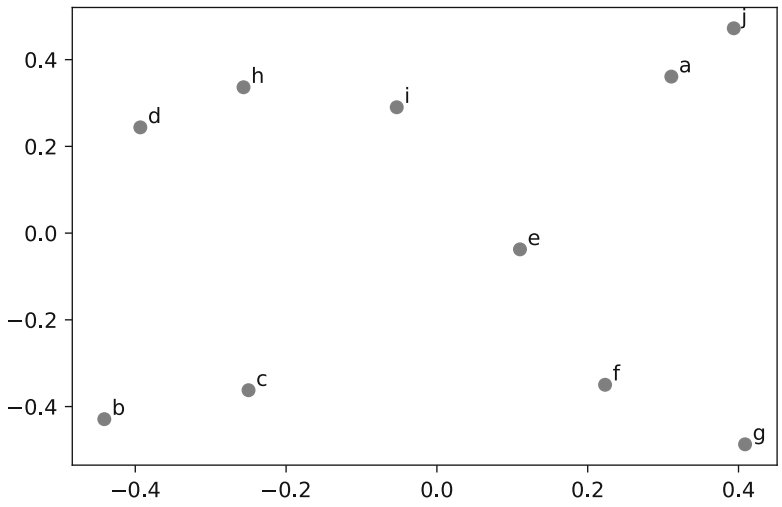

Fig. 9.14 A non-metric multidimensional scaling chart for mean brand ratings expressed as ordinal ranks. The brand groupings are similar to but more diffuse than those in Fig. 9.13

The co-occurrence matrix of counts—brand A mentioned with brand B, with brand C, and so forth—could be used as a measure of similarity between the two brands and serve as the distance metric in MDS (see Netzer et al. 2012).

9.4.2 Visualization Using Low-Dimensional Embeddings

Visualizing high-dimensional data is difficult, as we are effectively limited to two dimensions. PCA can be used to reduce dimensionality to two dimensions, but the resulting scatterplots are often very difficult to interpret. There are several non-linear dimensionality reduction tools that are explicitly tailored to visualization, representing high-dimensional structure in two dimensions.

t-SNE

t-SNE, t-distributed Stochastic Neighbor Embedding (McInnes et al. 2008), is a nonlinear dimensionality technique that is primarily used for visualizing high-dimensional systems, such as the high-level representations learned by a neural network architecture. It is sensitive to the specific parameters, and is stochastic, so each time it is run the representation will be different. But it does an excellent job at highlighting high-dimensional structure in the data.

A t-SNE method is included in `sklearn` in the `manifold` library:

```
In [44]: brand_tsne = manifold.TSNE().fit_transform(brand_ratings_sc_vals)
         brand_tsne_df = pd.DataFrame(brand_tsne, columns=['x', 'y'])
         brand_tsne_df['brand'] = brand_ratings_sc.brand
```

We get each response transformed into the t-SNE fitted space. If we overlay the brand on a scatterplot of all the points, we can see the relative position of the different brands in the t-SNE space in the upper-left panel of Fig. 9.15:

```
In [45]: sns.pairplot(brand_tsne_df, x_vars=['x'], y_vars=['y'],
                      hue='brand', size=10,
                      palette=sns.color_palette('Paired', n_colors=10))
```

UMAP

We can also use a similar technique, Uniform Manifold Approximation and Projection (McInnes et al. 2018), or UMAP, another dimensionality reduction technique for visualizing high-dimensional structure in two dimensions. The mechanics of training a UMAP model and visualizing the transformed data are similar to t-SNE:

```
In [46]: import umap

         brand_embedding = umap.UMAP().fit_transform(brand_ratings_sc_vals)
         brand_umap_df = pd.DataFrame(brand_embedding, columns=['x', 'y'])
         brand_umap_df['brand'] = brand_ratings_sc.brand

In [47]: sns.pairplot(brand_umap_df, x_vars=['x'], y_vars=['y'],
                      hue='brand', size=10,
                      palette=sns.color_palette('Paired', n_colors=10))
```

The UMAP-transformed data can be seen in the upper-right panel of Fig. 9.15.

The results between the models are similar. UMAP is faster and tends to "collapse" the clusters more. Unlike t-SNE, the trained model can also be saved and applied to new (but similar) data, which can be useful.

Comparing the t-SNE and UMAP representations to the PCA, EFA and MDS in Fig. 9.15, the additional structure provided by the non-linear dimensionality reduction algorithms is clear.

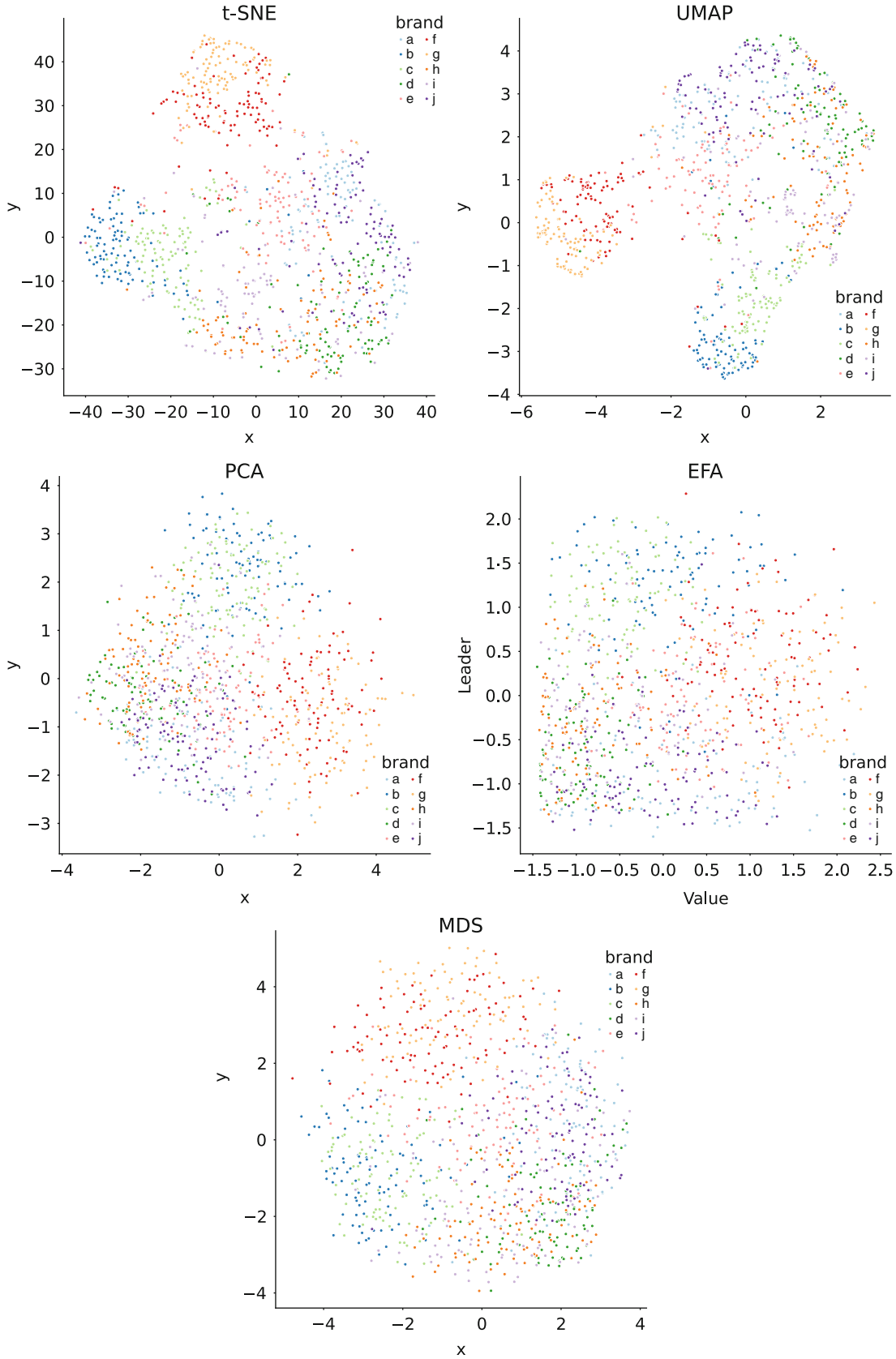

Fig. 9.15 t-SNE (upper-left), UMAP (upper-right), PCA (lower-left), EFA (lower-right), and MDS (bottom) representations of the brand ratings. The similarities between brands *b* and *c* as well as brands *f* and *g* are very evident, but the structure is far more apparent in the t-SNE and UMAP representations than in the others

9.5 Learning More*

Principal Component Analysis There is a large literature describing many procedures, options, and applications for each of the analyses in this chapter. With perceptual mapping, a valuable resource is Gower et al. (2010) which describes common problems and best practices for perceptual maps. Jolliffe (2002) provides a comprehensive text on the mathematics and applications of principal component analysis.

Factor Analysis A good conceptual overview of exploratory factor analysis with procedural notes is Fabrigar and Wegener, *Exploratory Factor Analysis* (Fabrigar and Wegener 2011). A modestly more technical volume that covers exploratory and confirmatory models together, with a social science (psychology) point of view, is Thompson, *Exploratory and Confirmatory Factor Analysis* (Thompson 2004). For examination of the mathematical bases and procedures of factor analysis, a standard text is Mulaik, *Foundations of Factor Analysis* (Mulaik 2009).

A companion to *exploratory* factor analysis is *confirmatory* factor analysis. Whereas EFA infers factor structure from a dataset, CFA tests a proposed model to see whether it corresponds well to observed data. A common use of EFA is to select items that load highly on underlying dimensions of interest. CFA allows you to confirm that the relationships between items and factors are maintained in new datasets. An introduction to CFA is given in this book's companion text, Chapman and Feit (2019), Chap. 10.

Multidimensional Scaling There are many uses and options for multidimensional scaling beyond those considered in this chapter. A readable introduction to the methods and applications is Borg, Groenen, and Mair, *Applied Multidimensional Scaling* (Borg et al. 2018). The statistical foundations and methods are detailed in Borg and Groenen, *Modern Multidimensional Scaling* (Borg and Groenen 2005).

Visualization Visualizing high-dimensional data is always a challenge. Embedding algorithms, such as t-SNE and UMAP, enable visualization through non-linear transforms of the data that maximize local structure and topography, often generating more interpretable visualizations than a PCA. See McInnes et al. (2008) and McInnes et al. (2018) for more information.

9.6 Key Points

Investigation of data complexity has several benefits. It allows inspection of the underlying dimensional relationships among variables, investigation of how observations such as brands or people vary on those dimensions, and estimation of a smaller number of more reliable dimensional scores. The following key points will assist you to investigate the underlying dimensions of your data.

Principal Component Analysis

- Principal component analysis (PCA) finds *linear functions* that explain maximal variance in observed data. A key concept is that such components are *orthogonal* (uncorrelated). A basic Python module is `sklearn.decomposition.PCA` (Sect. 9.2.1).
- A common use for PCA is a *biplot* of aggregate scores for brands or people to visualize relationships. When this is done for attitudinal data such as brand ratings it is called a *perceptual map*. This is created by aggregating the statistic of interest by entity and charting a biplot (Sect. 9.2.2).
- Because PCA components often load on many variables, the results must be inspected cautiously and in terms of relative position. It is particularly difficult to read the status of individual items from a PCA biplot (Sect. 9.2.5).

Exploratory Factor Analysis

- Exploratory factor analysis (EFA) models *latent variables* (factors) that are not observed directly but appear indirectly as observed *manifest variables*. A valuable library is `factor_analysis` (Sect. 9.3.1).
- A fundamental decision in EFA is the *number of factors* to extract. Common criteria involve inspection of a *scree* plot and extraction of factors such that all *eigenvalues* are greater than 1.0. The final determination depends on one's theory and the utility of results (Sect. 9.3.2).
- EFA uses *rotation* to adjust an initial solution to one that is mathematically equivalent but more interpretable according to one's aims. Another key decision in EFA is whether one believes the underlying latent variables should be uncorrelated (calling for an *orthogonal* rotation such as `varimax`) or correlated (calling for an *oblique* rotation such as `oblimin`) (Sect. 9.3.3).

- After performing EFA, you can extract *factor scores* that are the best estimates for each observation (respondent) on each factor. These can be extracted from the `FactorAnalysis()` object using the `get_scores()` method (Sect. 9.3.4).

Multidimensional Scaling

- Multidimensional scaling (MDS) is similar to principal component analysis but is able to work with both *metric* and *non-metric* data. MDS scaling can be performed using `sklearn.manifold.MDS()` for (Sect. 9.4).
- One advantage of MDS is that the dimensionality is constrained: you can limit it to two dimensions, in which case 100% of the variance will be represented in those dimensions
- Another advantage of MDS is that the relationship between variables is preserved, so if there is clear structure in the data, it is less likely to be lost
- However, the MDS model itself is generally less useful for subsequent analysis, unlike for the PCA or EFA

Low-Dimensional Embedding Visualization

- Like MDS, t-SNE and UMAP represent two algorithms for projecting high-dimensional structure into two dimensions (or an arbitrary number), which can be a way to understand a single observation's position within the environment of all other observations as well as visualizing overall structure within the system (Sect. 9.4.2)
- Unlike MDS, t-SNE and UMAP do not maintain relationships between observations, but rather make apparent topography present in higher dimensions. This means that clusters stand out more clearly in these representations than in the others discussed in the chapter