Chapter 10
Segmentation: Unsupervised Clustering Methods for Exploring Subpopulations

In this chapter, we tackle a canonical marketing research problem: finding, assessing, and predicting customer segments. In previous chapters we've seen how to assess relationships in data (Chap. 4), compare groups (Chap. 5), and assess models (Chap. 7). In a real segmentation project, one would use those methods to ensure that data have appropriate multivariate structure, and then begin segmentation analysis.

Segmentation is not a well-defined process and analysts vary in their definitions of segmentation as well as their approaches and philosophies. This chapter demonstrates our approach using basic models in Python. As always, this should be supplemented by readings we suggest at the end of the chapter.

We start with a warning: we have definite opinions about segmentation and what we believe are common misunderstandings and poor practices. We hope you'll be convinced by our views—but even if not, the methods here will be useful to you.

10.1 Segmentation Philosophy

The general goal of market segmentation is to find groups of customers that differ in important ways that are associated with product interest, market participation, or response to marketing efforts. By understanding the differences among groups, a marketer can make better strategic choices about opportunities, product definition, and positioning, and can engage in more effective promotion.

10.1.1 The Difficulty of Segmentation

The definition of segmentation above is a textbook description and does not reflect what is most difficult in a segmentation project: finding actionable business outcomes. It is not particularly difficult to find *groups* within consumer data; indeed, in this chapter we see several ways to do this, all of which "succeed" according to one statistical criterion or another. Rather, the difficulty is to ensure that the outcome is *meaningful* for a particular business need.

It is outside the range of this book to address the question of business need in general. However, we suggest that you ask a few questions along the following lines. If you were to find segments, what would you do about them? Would anyone in your organization use them? Why and how? Are the differences between segments large enough to be meaningful for your business? Among various solutions you might find, are there organizational efforts or politics that would make one solution more or less influential than another?

There is no magic bullet to find the "right" answer. For segmentation this means that there is no all-purpose method or algorithm that is a priori preferable to others. This does not mean that the choice of a method is irrelevant or arbitrary; rather, one cannot necessarily determine in advance which approach will work best for a novel problem. As a form of optimization, segmentation is likely to require an iterative approach that successively tests and improves its answer to a business need.

Segmentation is like slicing a pie, and any pie might be sliced in an infinite number of ways. Your task as an analyst is to consider the infinity of possible data that might be gathered, the infinity of possible groupings of that data, and the infinity of possible business questions that might be addressed. Your goal is to find a solution within those infinities that represents real differences in the data and that informs and influences business decisions.

J. S. Schwarz et al., *Python for Marketing Research and Analytics*, https://doi.org/10.1007/978-3-030-49720-0_10

Statistical methods are only part of the answer. It often happens that a "stronger" statistical solution poses complexity that makes it impossible to implement in a business context while a slightly "weaker" solution illuminates the data with a clear story and fits the business context so well that it can have broad influence.

To maximize chances of finding such a model, we recommend that an analyst expects—and prepares management to understand—two things. First, a segmentation project is not a matter of "running a segmentation study" or "doing segmentation analysis on the data." Rather, and second, it is likely to take multiple rounds of data collection and analysis to determine the important data that should be collected in the first place, to refine and test the solutions, and to conduct rounds of interpretation with business stakeholders to ensure that the results are actionable.

10.1.2 Segmentation as Clustering and Classification

In this chapter and the next, we demonstrate several methods in Python that will get you started with segmentation analysis. We explore two distinct yet related areas of statistics: *clustering* or *cluster analysis* (in this chapter) and *classification* (in Chap. 11). These are the primary branches of what is sometimes called *statistical learning* or *machine learning*, i.e., learning from data through statistical model fitting.

A key distinction in statistical learning is whether the method is *supervised* or *unsupervised*. In *supervised learning*, a model is presented with observations whose outcome status (dependent variable) is known, with a goal to predict that outcome status from the independent variables of novel observations. For example, we might use data from previous direct marketing campaigns—with a known outcome of whether each target responded or not, plus other predictor variables—to fit a model that predicts likelihood of response in a new campaign. We refer to this process as *classification*, which we discuss in the next chapter, Chap. 11.

In *unsupervised learning* we do not know the outcome groupings but are attempting to discover them from structure in the data. For instance, we might explore a direct marketing campaign and ask, "Are there groups that differ in how and when they respond to offers? If so, what are the characteristics of those groups?" We use the term *clustering* for this approach.

Clustering and classification are both useful in segmentation projects. Stakeholders often view segmentation as discovering groups in the data in order to derive new insight about customers. This obviously suggests clustering approaches because the possible customer groups are unknown. Still, classification approaches are also useful in such projects for at least two reasons: there may be outcome variables of interest that are known (such as observed in-market response) that one wishes to predict from segment membership, and if you use clustering to discover groups you will probably want to predict (i.e., classify) future responses into those groups. Thus, we view clustering and classification as complementary approaches.

A topic we do not address is how to determine what data to use for clustering, the observed *basis variables* that go into the model. That is primarily a choice based on business need, strategy, and data availability. Still, you can use the methods here to evaluate different sets of basis variables. If you have a large number of measures available and need to determine which ones are most important, the *variable importance* assessment method we review in Sect. 11.1.3 might assist. Aside from that, we assume in this chapter that the basis variables have been determined (and we use the customer relationship data from Chap. 5).

There are hundreds of books, thousands of articles, and many Python packages for clustering and classification methods, all of which propose hundreds of approaches with—as we noted above—no single "best" method. This chapter cannot cover clustering or classification in a comprehensive way, but we can give an introduction that will get you started, teach you the basics, accelerate your learning, and help you avoid some traps. As you will see, in most cases the process of fitting such models in Python is extremely similar from model to model.

10.2 Segmentation Data

We use the segmentation data (object `seg_df`) from Chap. 5. If you saved that data in Sect. 5.1.2, you can reload it (see Sect. 2.6.2 for a review of importing data):

```
In [1]: from google.colab import files

        f = files.upload()
Saving segment_dataframe_Python_intro_Ch5.csv to segment_dataframe_Python_intro_Ch5.csv
```

```
In [2]: import pandas as pd

        seg_df = pd.read_csv('segment_dataframe_Python_intro_Ch5.csv',
                             index_col=0)
```

Otherwise, you could download the dataset from the book website:

```
In [3]: import pandas as pd
        seg_df = pd.read_csv('http://bit.ly/PMR-ch5')
```

As you may recall from Chap. 5, these are simulated data with four identified segments of customers for a subscription product, and contain a few variables that are similar to data from typical consumer surveys. Each observation has the simulated respondent's age, gender, household income, number of kids, home ownership, subscription status, and assigned segment membership. In Chap. 5, we saw how to simulate this data and how to examine group differences within it. Other data sources that are often used for segmentation are customer relationship management (CRM) records, attitudinal surveys, product purchase and usage, and most generally, any dataset with observations about customers.

We check the data after loading:

```
In [4]: seg_df.head()

Out[4]:      Segment        age  gender          income  kids  own_home  \
        0  travelers  60.794945    male   57014.537526     0      True
        1  travelers  61.764535  female   43796.941252     0     False
        2  travelers  47.493356    male   51095.344683     0      True
        3  travelers  60.963694    male   56457.722237     0      True
        4  travelers  60.594199  female  103020.070798     0      True

           subscribe
        0      False
        1      False
        2      False
        3       True
        4      False
```

We use the subscription segment data for two purposes: to examine clustering methods that find intrinsic groupings (unsupervised learning, in this chapter), and to show how classification methods learn to predict group membership from known cases (supervised learning, in Chap. 11).

10.3 Clustering

We examine three clustering procedures that are illustrative of the hundreds of available methods. You'll see that the general procedure for finding and evaluating clusters in Python is similar across the methods.

To begin, we review two *distance-based* clustering methods, *hierarchical* and *k-means*. Distance-based methods attempt to find groups that minimize the distance between members within the group, while maximizing the distance of members from other groups. Hierarchical clusters does this by modeling the data in a tree structure, while k-means uses group centroids (central points).

Then we examine a *model-based* clustering method, a *Gaussian mixture model*. Model-based methods view the data as a mixture of groups sampled from different distributions, but whose original distribution and group membership has been "lost" (i.e., is unknown). The Gaussian mixture model method attempts to model the data such that the observed variance can be best represented by a small number of Gaussian (normal) variables with specific distribution characteristics such as different means and standard deviations.

10.3.1 The Steps of Clustering

Clustering analysis requires two stages: finding a proposed cluster solution and evaluating that solution for one's business needs. For each method we go through the following steps:

- Transform the data if needed for a particular clustering method; for instance, some methods require all numeric data (e.g., k-means)
- Scale the data if there are large differences in the magnitudes between columns. Distance-based methods are sensitive to those and can be dominated by a single dimension if it is much larger than the others (e.g., income in our data).
- Check for outliers in the data, which can dominate the clustering. Consider removing outlier data points. Note that we skip this step in our analysis.
- Apply the clustering method and save its result to an object. For most methods this requires specifying the number (K) of groups desired.
- For some methods, further parse the object to obtain a solution with K groups (e.g., `fcluster()` for extracting clusters from a linkage matrix).
- Examine the solution in the model object with regards to the underlying data, and consider whether it answers a business question.

As we've already argued, the most difficult part of that process is the last step: establishing whether a proposed statistical solution answers a business need. Ultimately, a cluster solution is largely just a vector of purported group assignments for each observation, such as "1, 1, 4, 3, 2, 3, 2, 2, 4, 1, 4 ...." It is up to you to figure out whether that tells a meaningful story for your data.

Transforming and Scaling the Data

The original dataset `seg_df` contains "known" segment assignments that have been provided for the data from some other source (as might occur from some human coding process). Because our task here is to discover segments, we create a copy `seg_sub` that omits those assignments, so we don't accidentally include the known values when exploring segmentation methods. (Later, in the chapter on classification, we will use the correct assignments because they are needed to train the classification models.)

While some clustering models can make use of categorical variables, the approaches we describe in this chapter use numerical data. In this dataset, all categorical variables are binary. `subscribe` and `own_home` are already explicitly binary, coded as booleans. Python functions that expect numeric values generally treat booleans as numeric values, `0` or `1` for `False` or `True`, respectively. We will convert `gender` to a boolean variable `is_female` and drop the `gender` column when we drop the `Segment` columns:

```
In [5]: seg_df['is_female'] = seg.gender == 'female'
        seg_sub = seg.drop(['Segment', 'gender'], axis=1)
        seg_sub.head()
```

Additionally, distance-based clustering approaches are sensitive to the scale of the variables. A change of 10 is relatively insignificant in income, but quite significant in the number of children! However, the distance metric would treat those equally. To address this we can scale the data as we have in past chapters:

```
In [6]: from sklearn import preprocessing

        seg_sc = pd.DataFrame(preprocessing.scale(seg_sub),
                              columns=seg_sub.columns)
        seg_sc.head()
```

```
Out[6]:        age    income      kids  own_home  subscribe  is_female
        0  1.551729  0.328689 -0.902199  1.120553  -0.363422  -0.960769
        1  1.627442 -0.356010 -0.902199 -0.892416  -0.363422   1.040833
        2  0.513037  0.022062 -0.902199  1.120553  -0.363422  -0.960769
        3  1.564906  0.299844 -0.902199  1.120553   2.751623  -0.960769
        4  1.536053  2.711871 -0.902199  1.120553  -0.363422   1.040833
```

The scaled data look appropriate. We have created two new dataframes. One, seg_sub, has all the columns that we need for this analysis and no more. The other, seg_sc, has the same columns, but scaled such that columns are directly comparable.

A Quick Check Function

We recommend that you think hard about how you would know whether a solution—assignments of observations to groups—that is proposed by a clustering method is useful for your business problem. Just because some grouping is proposed by an algorithm does not mean that it will help your business. One way we often approach this is to write a simple function that summarizes the data and allows quick inspection of the high-level differences between groups.

A segment inspection function may be complex depending on the business need and might include plotting in addition to data summarization. Pandas provides the pivot_table() function that allows us to aggregate by any arbitrary index:

```
In [7]: pd.pivot_table(seg_sub, index=seg_df.Segment)

Out[7]:                   age        income  is_female      kids  own_home  \
        Segment
        moving up   36.216087  51763.552666      0.700  1.857143  0.357143
        suburb_mix  39.284730  55552.282925      0.530  1.950000  0.480000
        travelers   57.746500  62609.655328      0.325  0.000000  0.662500
        urban_hip   23.873716  20267.737317      0.320  1.140000  0.140000

                    subscribe
        Segment
        moving up    0.214286
        suburb_mix   0.070000
        travelers    0.025000
        urban_hip    0.220000
```

The pivot_table() function calculates the mean by default, but any suitable function or list of functions can be passed to the aggfunc argument:

```
In [8]: import numpy as np

        # Output not shown
        pd.pivot_table(seg_sub, index=seg_df.Segment,
                       aggfunc=[np.mean, np.std]).unstack()
In [9]: pd.pivot_table(seg_sub, index=seg_df.Segment,
                       aggfunc=lambda x: np.percentile(x, 95))
```

However, for purposes here we use a simple function that reports the mean by group. We use mean here instead of a more robust metric such as median because we have several binary variables and mean() easily shows the mixture proportion for them (i.e., 0.5 means a 50% mix of 0 and 1).

We create our own function to simplify calling it:

```
In [10]: def check_clusters(data, labels):
           return pd.pivot_table(data,
                                 index=labels)

         # Output not shown
         check_clusters(seg_sub, seg_df.Segment)
```

While making this a function isn't really necessary in this case, there are several reasons to do so. Writing our own function allows us to minimize typing by providing a short command. By providing a consistent and simple interface, it reduces risk of error. And it is extensible; as an analysis proceeds, we might decide to add to the function, expanding it to report variance metrics or to plot results, without needing to change how we invoke it (something we will do in this chapter, see Sect. 10.3.4).

With a summary function of this kind we are easily able to answer the following questions related to the business value of a proposed solution:

- Are there obvious differences in group means?
- Does the differentiation point to some underlying story to tell?
- Do we see immediately odd results such as a mean equal to the value of one data level?

This simple function will help us to inspect cluster solutions efficiently. It is not intended to be a substitute for detailed analysis—and it takes shortcuts such as treating categorical variables as numbers, which is inadvisable except for analysts who understand what they're doing—yet it provides a quick first check of whether there is something interesting (or uninteresting) occurring in a solution.

10.3.2 Hierarchical Clustering

Pairwise Distances

The primary information in hierarchical clustering is the *distance* between observations. There are many ways to compute distance, and we start by examining the best-known method, the *Euclidean distance*. For two observations (vectors) X and Y, the Euclidean distance d is:

$$d = \sqrt{\sum (A - B)^2} \tag{10.1}$$

For single pairs of observations, such as $A = \{1, 2, 3\}$ and $B = \{2, 3, 2\}$ we can compute the distance easily in Python using numpy:

```
In [11]: # Vector of differences
         np.array([1, 2, 3]) - np.array([2, 3, 2])

Out[11]: array([-1, -1,  1])

In [12]: # Sum of the squared distances
         np.sum((np.array([1, 2, 3]) - np.array([2, 3, 2]))**2)

Out[12]: 3

In [13]: # Root sum of the squared distances
         np.sqrt(np.sum((np.array([1, 2, 3]) - np.array([2, 3, 2]))**2))

Out[13]: 1.7320508075688772
```

When there are many pairs, this can be done with the `pdist()` function from the scipy `distance` module. Let's check it first for the simple X, Y example:

```
In [14]: from scipy.spatial import distance

         distance.pdist([np.array([1, 2, 3]), np.array([2, 3, 2])])

Out[14]: array([1.73205081])
```

A limitation is that Euclidean distance is only defined when observations are numeric. In our data `seg_df` it is impossible to compute the distance between `male` and `female`. This is why we transformed the gender column into a boolean column. If we did not care about the factor variables, then we could compute Euclidean distance using only the numeric columns.

The `scipy.spatial.distance` module also includes the `squareform` function which computes a distance matrix. We can observe how it works on the first three rows of the `seg_sc` dataframe:

```
In [15]: distance.squareform(distance.pdist(seg_sc.iloc[:3]))

Out[15]: array([[0.        , 2.92113022, 1.08300539],
                [2.92113022, 0.        , 3.07299428],
                [1.08300539, 3.07299428, 0.        ]])
```

To find the distance between the second and third variable, we can look at the second row, third column or the third row, second column and see that the distance is 3.07. The distance matrix is symmetric and, as expected, the distance of an observation from itself is 0. Notice that the square-form distance matrix has a lot of redundant information, which we can remove to make it more memory efficient. This can be important when dealing with datasets with a large number of columns.

To get a more compact form, we use `pdist()` which returns a *condensed*, *vector-form* distance matrix:

```
In [16]: distance.pdist(seg_sc.iloc[:3])

Out[16]: array([2.92113022, 1.08300539, 3.07299428])
```

The condensed distance matrix is more memory efficient, but is harder for a human to inspect.

If we inspect the pairwise distances in our dataset, the importance of scaling the data becomes clear.

```
In [17]: import matplotlib.pyplot as plt
         plt.style.use('seaborn-white')

         plt.figure(figsize=(5,10))
         plt.subplot(2,1,1)
         plt.hist(distance.pdist(seg_sub))
         plt.title('Pairwise distances from unscaled data')
         plt.xlabel('Distance')
         plt.ylabel('Count')
         plt.subplot(2,1,2)
         plt.hist(distance.pdist(seg_sc))
         plt.title('Pairwise distances from scaled data')
         plt.xlabel('Distance')
         plt.ylabel('Count')
```

As we can see in the left panel of Fig. 10.1, the unscaled data produces a highly skewed distribution of distances, which is dominated by differences in income between customers (rows in the data). Contrast that with the distribution of distances from the scaled data in the right panel.

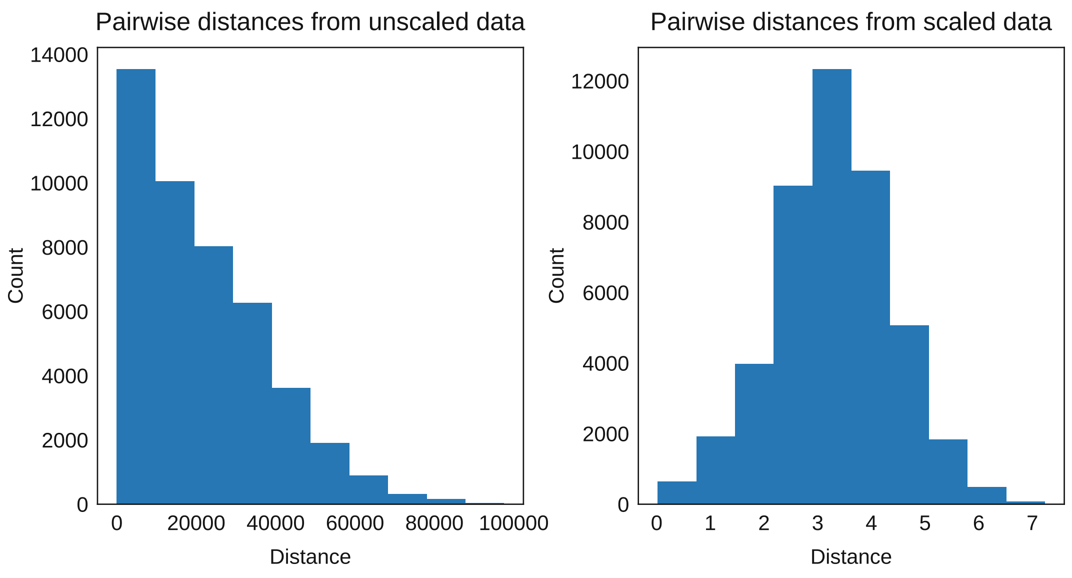

Fig. 10.1 The distribution of pairwise distances from the unscaled data is very skewed (left panel), in contrast to the approximately normal distribution from the scaled data (right panel)

Hierarchical Clustering in Python

Hierarchical clustering is a popular method that groups observations according to their similarity. The `linkage()` function from the scipy `hierarchy` module generates hierarchical clustering. The clusters are generated using a distance-based algorithm that operates on a *dissimilarity* matrix, an N-by-N matrix that reports a metric for the *distance* between each pair of observations.

The hierarchical clustering method begins with each observation in its own cluster. It then successively joins neighboring observations or clusters one at a time according to their distances from one another, and continues this until all observations are linked. This process of repeatedly joining observations and groups is known as an *agglomerative* method. Because it is both very popular and exemplary of other methods, we present hierarchical clustering in more detail than the other clustering algorithms.

We pass our data into the `linkage()` function, which calculates distances and runs the clustering algorithm, producing a linkage matrix:

```
In [18]: from scipy.cluster import hierarchy

         linkages = hierarchy.linkage(seg_sc, method='ward')
```

We use the *Ward* linkage method, which forms groups that minimize the total within-cluster variance.

Passing the linkage matrix to the `dendrogram()` function will plot a tree representing the linkage matrix:

```
In [19]: hierarchy.dendrogram(linkages)
         plt.show()
```

The resulting tree for all $N = 300$ observations of `seg_sub` is shown in Fig. 10.2.

A hierarchical dendrogram is interpreted primarily by height and where observations are joined. The height represents the dissimilarity between elements that are joined. At the lowest level of the tree in Fig. 10.2 we see that elements are combined into small groups of 2–10 that are relatively similar, and then those groups are successively combined with less similar groups moving up the tree. The horizontal ordering of branches is not important; branches could exchange places with no change in interpretation.

Figure 10.2 is difficult to read. There are two ways in which we can make it simpler to read.

One is with the `truncate_mode` and `p` arguments to `dendrogram()`. If `truncate_mode` is set to `'lastp'` or to `'level'`, a condensed tree will be shown. The degree of truncation is set with the argument to `p`. In `lastp` mode, only `p` branches will be shown. For example, if we set `p=20`, we can see in Fig. 10.3 that the leaves are condensed.

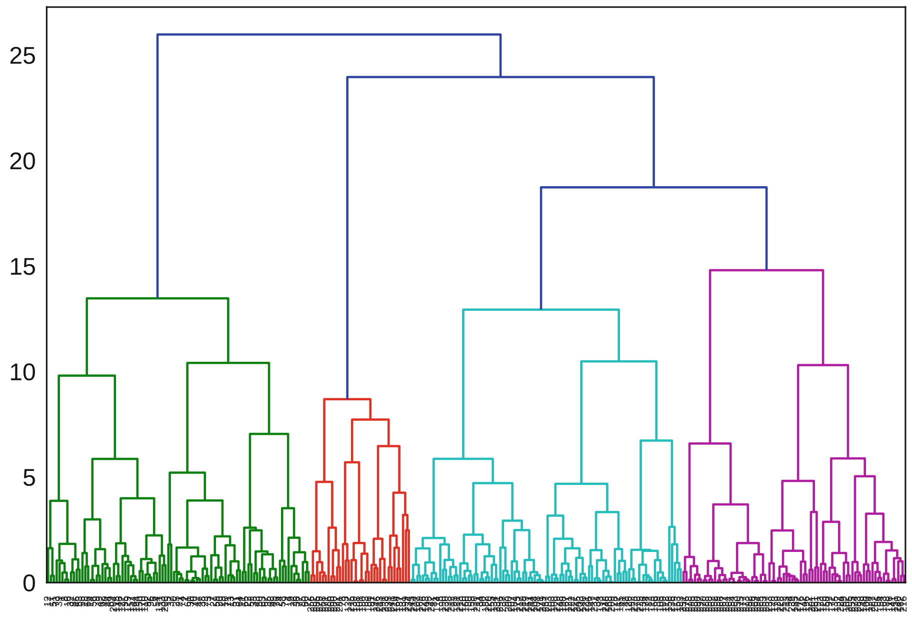

Fig. 10.2 Complete dendrogram for the scaled segmentation data, using `linkage()` and `dendrogram()` from the scipy `hierarchy` module

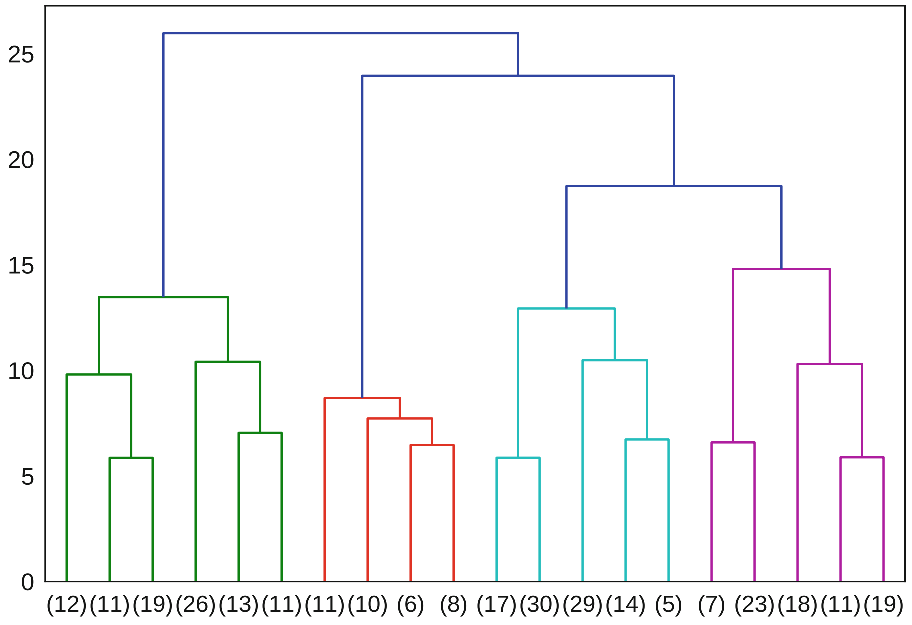

Fig. 10.3 A truncated tree, showing only 20 branches, each of which represents multiple leaves (the number of leaves are shown in parentheses)

```
In [20]: hierarchy.dendrogram(linkages, orientation='top',
                              truncate_mode='lastp', p=20)
         plt.show()
```

The number of leaves that each branch represents are indicated in parentheses. If an individual leaf is present, its label will not have parentheses.

It is also helpful to zoom in on one section of the chart. We can do so using the `xlim()` function:

```
In [21]: plt.subplot(1,2,1)
         hierarchy.dendrogram(linkages, leaf_rotation=0)
         plt.xlim((0,200))
         plt.subplot(1,2,2)
         hierarchy.dendrogram(linkages, leaf_rotation=0)
         plt.xlim((2800, 3000))
         plt.show()
```

The result is shown in Fig. 10.4, where we are now able to read the observation labels (which defaults to the row names—usually the row numbers—of observations in the dataframe). Each node at the bottom represents one customer, and the tree show how each has been grouped progressively with other customers. Here we can see the far left and right edges of the tree and the observations present in those parts of the tree.

We can check the similarity of observations by selecting a few rows listed in Fig. 10.4. Observations 17 and 51 are represented as being quite similar because they are linked at a very low height, as are observations 163 and 88. On the other hand, observations 17 and 163 are only joined at the highest level of the tree and thus should be relatively dissimilar. We can check those directly:

```
In [22]: # Similar
         seg_sub.loc[[17, 51]]

Out[22]:           age         income  kids  own_home  subscribe  is_female
         17  73.266707  70157.058678     0     False      False       True
         51  71.172291  75554.353842     0     False      False       True

In [23]: # Similar
         seg_sub.loc[[163, 88]]
```

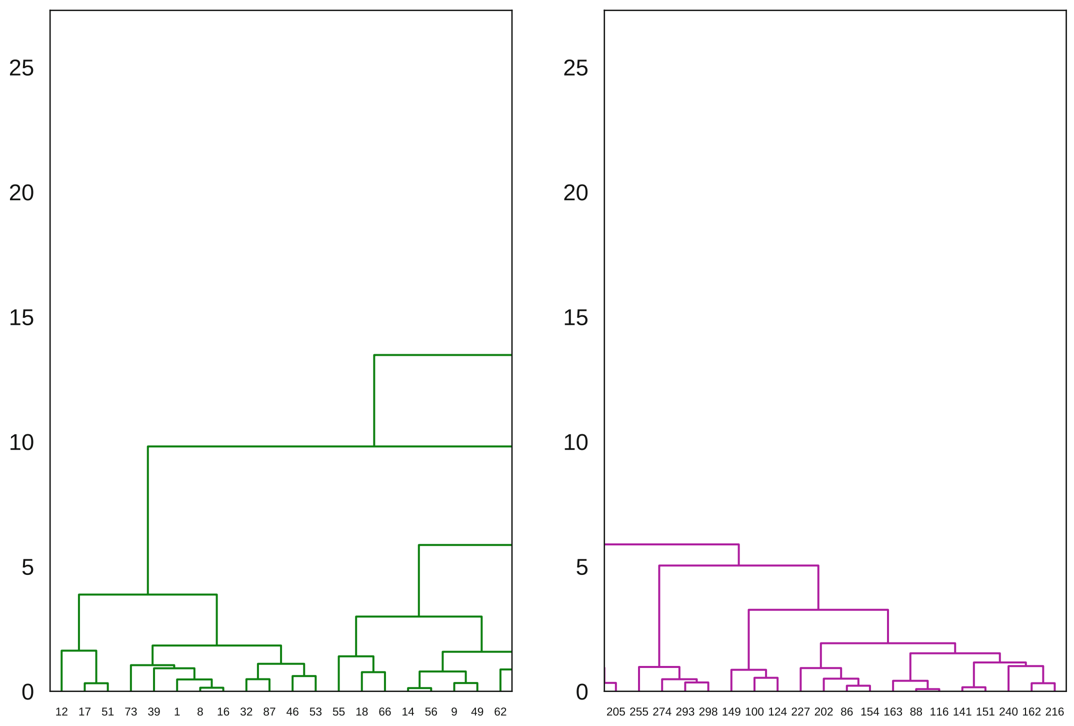

Fig. 10.4 A close up view of the left- and right-most branches from Fig. 10.2

```
Out[23]:            age         income  kids  own_home  subscribe  is_female
         163  39.653607   48996.400976     2      True      False      False
         88   40.106702   41744.977842     2      True      False      False
```

```
In [24]: # Dissimilar
         seg_sub.loc[[17,163]]
```

```
Out[24]:            age         income  kids  own_home  subscribe  is_female
         17   73.266707   70157.058678     0     False      False       True
         163  39.653607   48996.400976     2      True      False      False
```

The first two pairs—observations that are neighbors in the dendrogram—are similar on all variables (age, gender, income, etc.). The third pair—observations taken from widely separated branches—differ substantially.

We can check one of the goodness-of-fit metrics for a hierarchical cluster solution. One method is the *cophenetic correlation* coefficient (CPCC), which assesses how well the dendrogram matches the distance metric (Sokal and Rohlf 1962). We use `cophenet()` to compare the distances from the linkages with the `pdist()` metrics:

```
In [25]: hierarchy.cophenet(linkages, distance.pdist(seg_sc))[0]
```

```
Out[25]: 0.5985290160084774
```

CPCC is interpreted similarly to Pearson's r. In this case, CPCC is about 0.6, indicating a moderately strong fit, meaning that the hierarchical tree represents the distances between customers well.

10.3.3 Hierarchical Clustering Continued: Groups from `fcluster`

How do we get specific segment assignments? A dendrogram can be cut into clusters at any height desired, resulting in different numbers of groups. For instance, if Fig. 10.2 is cut at a height of 25 there are $K = 2$ groups (draw a horizontal line at 25 and count how many branches it intersects; each cluster below is a group), while cutting at height of 9 defines $K = 11$ groups.

Because a dendrogram might be cut at any point, an analyst must specify the number of groups desired. We can see the clusters based on where the dendrogram would be cut by passing the `color_thresh` argument, which will color only the branches below that threshold. In Fig. 10.2, the default value selected 4 clusters, but we can manually set it to a different value:

```
In [26]: # Not shown
         hierarchy.dendrogram(linkages, color_threshold=9)
         plt.show()
```

We obtain the assignment vector for observations using `fcluster()`. We specify a criterion type as `'maxclust'` and a threshold of 4 (`t=4`), which tells `fcluster()` that we expect 4 clusters. There are other clustering criteria that you can read about in the `fcluster()` documentation.

```
In [27]: labels = hierarchy.fcluster(linkages, t=4, criterion='maxclust')
         list(zip(*np.unique(labels, return_counts=True)))

Out[27]: [(1, 92), (2, 35), (3, 95), (4, 78)]
```

`fcluster()` returned the cluster label for each observation. We use `np.unique(return_counts=True)` to get the observation count for each cluster. We use `list(zip(...))` just as a formatting trick. Here we use the "`*`" operator to unpack the tuple of values returned by `np.unique()` and pass them to the `zip()` function. What does "unpacking" mean in this context? Since the values are returned in a tuple, they cannot be directly passed as arguments. We could have assigned the output to a variable, say `unique_label_counts` and then used the command `zip(unique_label_counts[0], unique_label_counts[1])` to achieve the same effect.

We see that groups 1, 3, and 4 are similar in size while group 2 is less than half the size of the others. Note that the class labels (1, 2, 3, 4) are in arbitrary order and are not meaningful in themselves.

We use our custom summary function `check_clusters()`, defined above, to inspect the variables in `seg_sub` with reference to the four clusters:

```
In [28]: check_clusters(seg_sub, labels)

Out[28]:          age         income  is_female      kids  own_home  subscribe
         1  54.474706  63219.658293   0.250000  0.152174  0.521739        0.0
         2  34.523881  41685.199147   0.542857  1.514286  0.314286        1.0
         3  38.204641  51578.802282   1.000000  1.873684  0.463158        0.0
         4  31.122503  38790.506683   0.089744  1.756410  0.384615        0.0
```

We see that group 2 contains all of the subscribers. Group 1 is the oldest and has the highest income and home ownership. Group 3 contains only women and has an intermediate level of income, age, and home ownership. Group 4 is predominately male and younger, with the lowest income.

For comparison, we can run the same analysis with the unscaled data and see what we find:

```
In [29]: linkages_unscaled = hierarchy.linkage(seg_sub, method='ward')
         hierarchy.dendrogram(linkages_unscaled)
         plt.show()
```

We see in Fig. 10.5 that 3 clusters are more appropriate than 4. Also, note that the distances (indicated on the y-axis) are much larger the before: these clusters are dominated by income.

Using our `check_clusters()` function, we see a slightly different pattern:

```
In [30]: labels_unscaled = hierarchy.fcluster(linkages_unscaled, t=3,
                                              criterion='maxclust')
         check_clusters(seg_sub, labels_unscaled)

Out[30]:          age         income  is_female      kids  own_home  subscribe
         1  26.238778  20026.508497   0.320755  1.113208  0.150943   0.207547
         2  48.102952  74464.263260   0.394737  1.052632  0.500000   0.052632
         3  42.283774  49591.504755   0.567251  1.421053  0.508772   0.116959
```

The three clusters are clearly very segregated by income, with group 1 being low, group 2 being high, and group 3 being intermediate. But we see potentially interesting patterns beyond that. Group 1 is low in income, young, two-thirds male, with

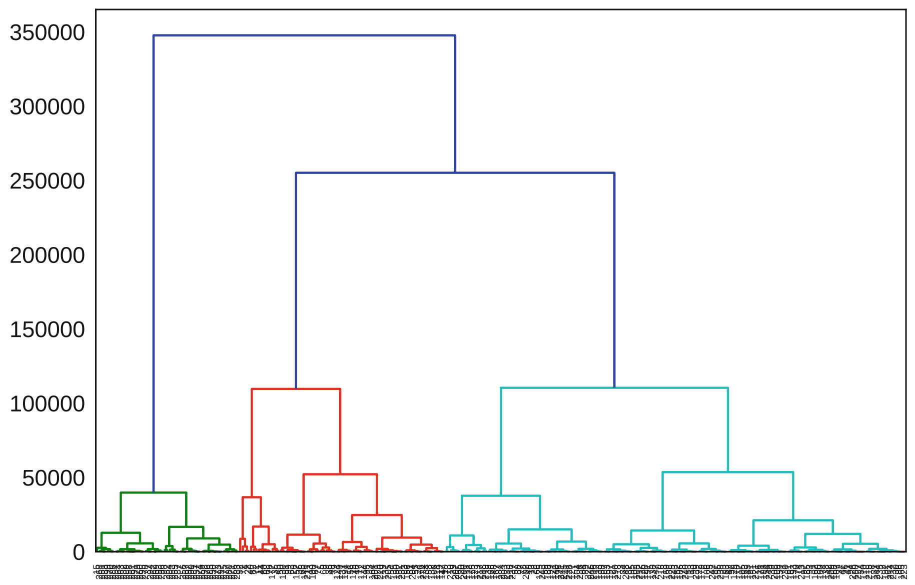

Fig. 10.5 Complete dendrogram for the unscaled segmentation data

few kids and low home-ownership, but a high subscription rate. Group 2 has the lowest subscription rate, but the highest income. Group 3 has a similar home-ownership rate as group 2, but twice the subscription rate, the most women, and more children.

It is interesting that, in this case, despite being defined by income, the clusters are well-differentiated in the other dimensions as well. We would argue that these clusters are probably more actionable from a business perspective. This goes to show that clustering is more about *interpretability* than about *accuracy*. In this case, the unscaled data outperformed the scaled data because *income is a good proxy for other demographics in our dataset*, something that will not surprise marketing researchers.

However, if the variable with the longest range had not been informative, the unscaled data would have likely done a better job revealing structure within the dataset.

We can visualize cluster membership based on values to get a better understanding. We create a helper function to produce a scatterplot of two of the columns and then color each point based on cluster membership. We can then investigate, for example, cluster membership based on age and income:

```
In [31]: def cluster_plot_raw(x, y, labels):
           for l in np.unique(labels):
             idx = labels == l
             plt.scatter(x[idx],
                         y[idx],
                         label=l)
           plt.legend()
           plt.xlabel(x.name)
           plt.ylabel(y.name)
```

```
In [32]: cluster_plot_raw(seg_sub.age, seg_sub.income, labels_unscaled)
```

This visualization (Fig. 10.6) confirms what we suspected: income dominates the clustering. Whether this is interesting or uninteresting depends on your objective. Overall, this demonstrates why you should expect to try several methods and iterate in order to find something useful.

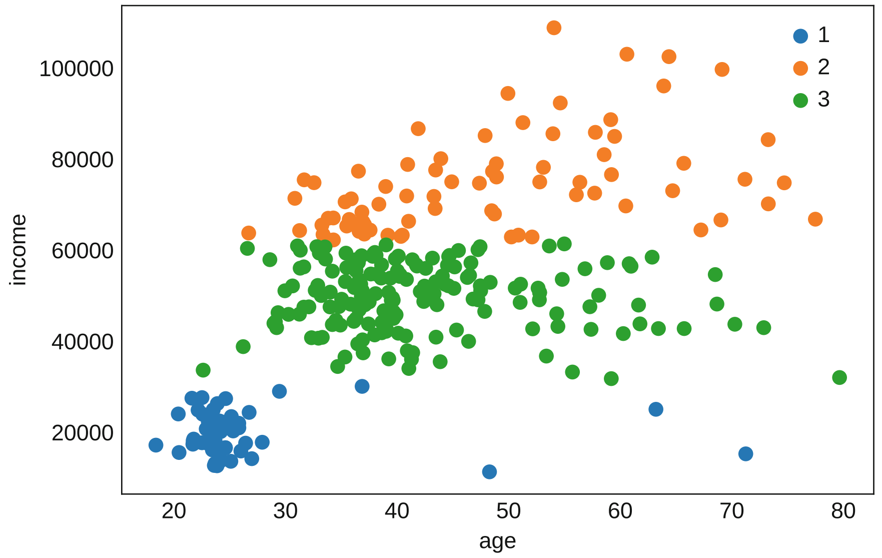

Fig. 10.6 Plotting the 3-segment hierarchical solution from the unscaled segmentation data by age and income, with color representing segment membership. We see that the clustering appears to be defined primarily by income

10.3.4 Mean-Based Clustering: `k_means()`

K-means clustering attempts to find groups that are most compact, in terms of the mean sum-of-squares deviation of each observation from the multivariate center (*centroid*) of its assigned group. Like hierarchical clustering, k-means is a very popular approach.

How does the basic algorithm work? It starts with an initial set of means, which may be random or nonrandom. The number of means is a parameter that must be specified. The algorithm then alternates between two steps:

- **Assign:** Each observation is assigned to the cluster represented by the nearest mean, nearness being defined by the least squared Euclidean distance
- **Update:** Calculate the centroid of each cluster, which collectively become the new means

This alteration repeats until the assignment of observations is stable.

Because it explicitly computes a mean deviation, k-means clustering relies on Euclidean distance. Thus it is only appropriate for numeric data or data that can be reasonably coerced to numeric, unlike hierarchical clustering which can utilize different distance metrics and thus be applied to categorical data as well.

We will extend our `check_clusters()` function by adding a line to determine cluster sizes:

```
In [33]: def check_clusters(data, labels):
           print(list(zip(*np.unique(labels, return_counts=True))))

           return pd.pivot_table(data,
                                 index=labels)
```

We can run the `k_means()` function from the `cluster` module of sci-kit learn. K-means requires specification of the number of clusters to find. We ask for four clusters with `n_clusters=4`. We will first run this on our scaled data:

```
In [34]: import numpy as np
         from sklearn import cluster

         np.random.seed(536)
         centroids, labels, inertia = cluster.k_means(seg_sc, n_clusters=4)
         check_clusters(seg_sub, labels)

[(0, 73), (1, 101), (2, 91), (3, 35)]
```

```
Out[34]:           age         income  is_female      kids  own_home  subscribe
          0  31.672851  39921.012710   0.000000  1.821918  0.315068        0.0
          1  37.043120  49285.905471   1.000000  1.811881  0.405941        0.0
          2  55.112042  64282.900228   0.263736  0.142857  0.637363        0.0
          3  34.523881  41685.199147   0.542857  1.514286  0.314286        1.0
```

Note that the line beginning `[(0, 73)`, is output from the function `check_clusters()` and will appear in the output block in a Colab notebook or in between the input and output blocks in a Jupyter notebook.

We also ran our check function `check_clusters()` to do a quick check of the data by proposed group, where cluster assignments are found in the `labels` vector.

These clusters superficially look a lot like the clusters we found in the hierarchical clustering of our scaled data. We have a group containing all of our subscribers (group 4). We now have a group that contains only women (group 1) and a group that contains only men (group 0). These are unlikely to prove useful. Telling stakeholders that there are two really important groups uncovered by our segmentation analysis, men and women, is unlikely to receive accolades!

What about if we use the unscaled data? We can check with four groups and also with three:

```
In [35]: centroids, k_labels_unscaled4, inertia = cluster.k_means(seg_sub,
                                                                n_clusters=4)
         check_clusters(seg_sub, k_labels_unscaled4)

[(0, 96), (1, 55), (2, 42), (3, 107)]

Out[35]:           age         income  is_female      kids  own_home  subscribe
          0  42.346106  60157.505981   0.541667  1.625000  0.447917   0.093750
          1  27.809087  20457.938690   0.327273  1.072727  0.163636   0.200000
          2  52.117381  81545.927332   0.309524  0.476190  0.571429   0.023810
          3  41.993915  45566.356272   0.570093  1.373832  0.532710   0.130841

In [36]: centroids, k_labels_unscaled3, inertia = cluster.k_means(seg_sub,
                                                                n_clusters=3)
         check_clusters(seg_sub, k_labels_unscaled3)

[(0, 64), (1, 65), (2, 171)]


Out[36]:           age         income  is_female      kids  own_home  subscribe
          0  29.635597  22520.530838   0.343750  1.109375  0.171875   0.187500
          1  49.494653  76393.497749   0.384615  0.923077  0.507692   0.046154
          2  41.889908  51426.578619   0.567251  1.467836  0.520468   0.116959
```

These groups, again, look rather like the hierarchical clusters we found with the unscaled data. Similar to those, they look potentially interesting. In each grouping, we have a young, lower-income group that is predominately male, with low home-ownership and a high subscription rate. We also have an older, higher-income group with a low subscription rate. In the four-segment model, both of those groups are better differentiated.

In the four-segment model, the intermediate group is split in two. This could be of interest or not. One might, for example, try to understand what is underlying the substantial difference in subscription rates between these otherwise undifferentiated groups.

Note that in clustering models, the group labels are in arbitrary order, so don't worry if your solution shows the same pattern with different labels.

We can visually check the differences in income between the groups using a boxplot:

```
In [37]: import matplotlib.pyplot as plt
         seg_sub.boxplot(column='income', by=k_labels_unscaled4)
         plt.xlabel('Cluster')
         plt.ylabel('Income')
         plt.suptitle('') # Remove cluster id subtitle
```

The result is Fig. 10.7, which shows substantial differences in income by segment.

We may also visualize the clusters by plotting them against a dimensional plot. We write a function to perform dimensional reduction with principal components and then plot the observations with cluster membership identified (see Chap. 9 to review principal component analysis and plotting):

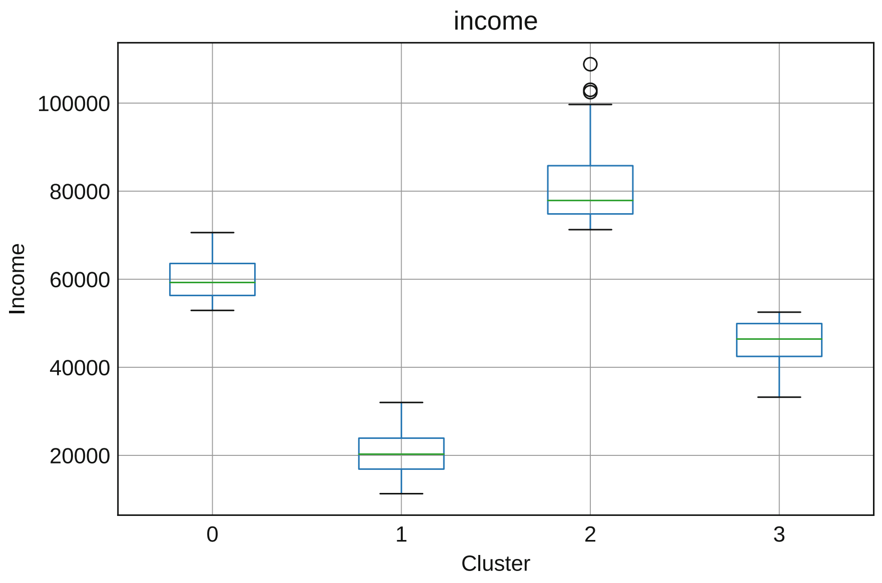

Fig. 10.7 Boxplot of income by cluster as found with `k_means()`

```
In [38]: from sklearn import decomposition
         from matplotlib import cm

         def cluster_plot(data_df, labels):
           p = decomposition.PCA(random_state=132, svd_solver='full')
           scaled_transformed = p.fit_transform(preprocessing.scale(data_df))
           for l in np.unique(labels):
             idx = np.where(labels == l)[0]
             plt.scatter(scaled_transformed[idx, 0],
                         scaled_transformed[idx, 1],
                         label=l)
           plt.legend()
           plt.title('First two components explain {}% of the variance'
                     .format(round(100*p.explained_variance_ratio_[:2].sum())))
           plt.xlabel('First principal component')
           plt.ylabel('Second principal component')


         cluster_plot(seg_sub, k_labels_unscaled4)
```

This produces Fig. 10.8, which plots cluster assignment by color against the first two principal components of the predictors (see Sect. 9.2.2). Groups 1 and 2 are largely overlapping (in this dimensional reduction) while groups 0 and 3 are more differentiated. This is consistent with what we observed using the output from `check_clusters()`.

Overall, this is an interesting cluster solution for our segmentation data. The groups here are clearly differentiated on key variables such as age and income. With this information, an analyst might cross-reference the group membership with key variables (as we did using our `check_clusters()`) function and then look at the relative differentiation of the groups (as in Fig. 10.8).

This may suggest a business strategy. In the present case, for instance, we see that group 3 is modestly well-differentiated, and has the highest average income. That may make it a good target for a potential campaign. Or we might focus on group 0 as our segment with the highest subscription rate and understand how to grow our market there. Many other strategies are possible, too; the key point is that the analysis provides interesting options to consider.

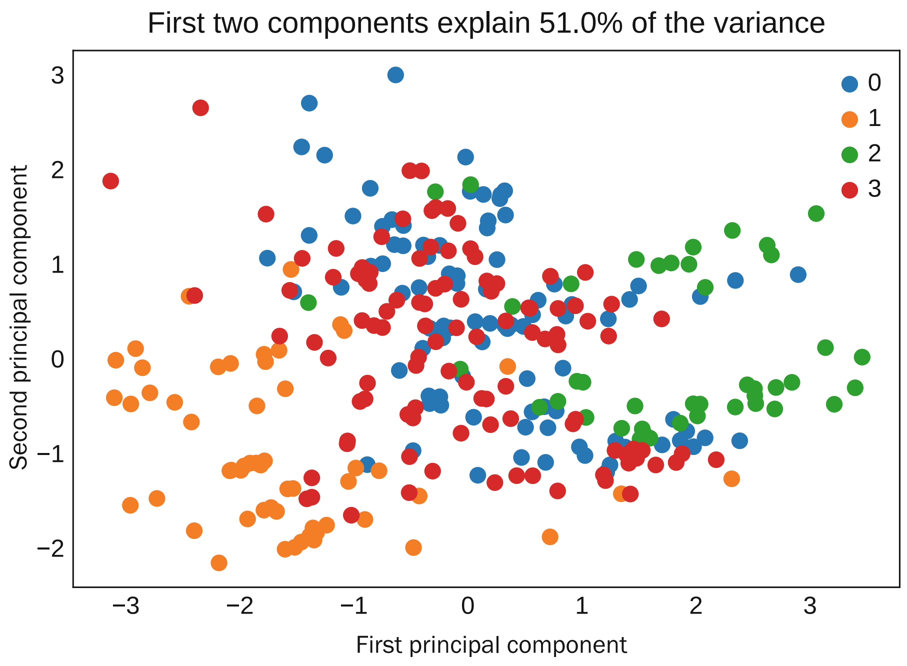

Fig. 10.8 Cluster plot created for the four group solution from `k_means()`. This shows the observations on a multidimensional scaling plot with group membership identified by the color

10.3.5 Model-Based Clustering: `GaussianMixture()`

The key idea for model-based clustering is that observations come from groups with different statistical distributions (such as different means and variances). The algorithms try to find the best set of such underlying distributions to explain the observed data. We use the `mixture` module from scikit-learn to demonstrate this.

Such models are also known as "mixture models" because it is assumed that the data reflect a mixture of observations drawn from different groups called components, although we don't know which component each observation was drawn from. `GaussianMixture()` models assume the observations are drawn from a mixture of normal (also known as *Gaussian*) distributions. We are trying to estimate the underlying component parameters and the mixture proportions.

How does this work? The most common approach is called the *expectation maximization (EM) algorithm*, which is analogous to that used for k-means, but rather than iteratively estimating centroids and using distances from those centroids to it estimates the probability of each point belonging to each Gaussian component (which is defined by its mean and variance). So, starting with random parameters for each model, the algorithm repeats these steps:

- **Assign:** Each observation is assigned to the component that it is most likely to belong to
- **Update:** The parameters for each component are updated given the points assigned to it

As you might guess, because `GaussianMixture()` models data with normal distributions, it uses only numeric data. The model is estimated with the `fit()` method on the `GaussianMixture()` and labels are generated with the `predict()` method:

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

         gmm4 = mixture.GaussianMixture(n_components=4,
                                        covariance_type='full',
                                        random_state=323).fit(seg_sub)
         gmm4_labels = gmm4.predict(seg_sub)
         gmm4.bic(seg_sub)

Out[39]: 7892.76042330893
```

`GaussianMixture()` requires as input the number of model components and the covariance structure within each component. Above, we fit the model with four components, i.e. four groups or clusters, which each have a full covariance

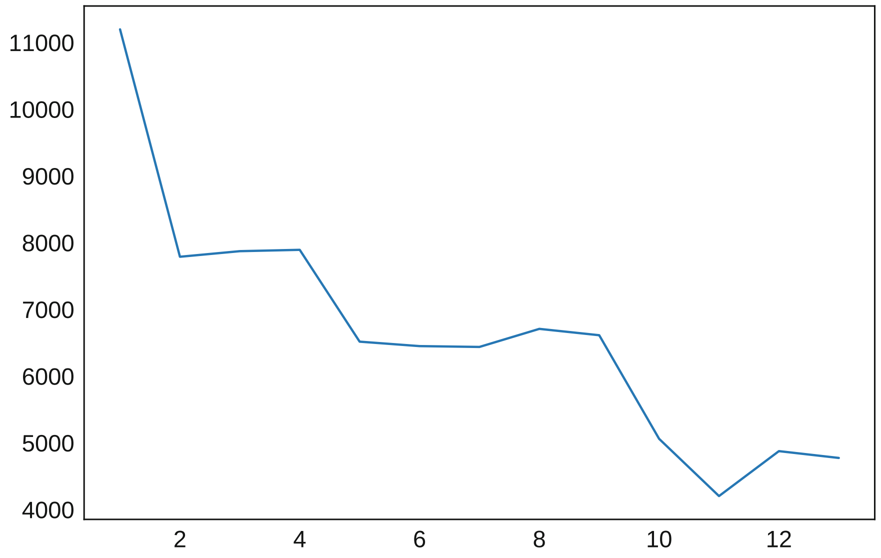

Fig. 10.9 BIC values of `GaussianMixture()` models as a function of the number of components. Model fit does not improve beyond five components until we get to ten components, which is likely too many to interpret effectively

structure. Note that since this is a probability model, we have available metrics on the model goodness of fit, such as the Bayesian Information Criterion (BIC) (Raftery 1995). This gives us an objective function on which to optimize the model, for instance, by optimizing the number of components. Additionally, since the cluster assignments are probabilistic, we can inspect the model confidence for each individual point.

First, let's determine what the best number of clusters is. We can do so by fitting models with different numbers of components and comparing the BIC values:

```
In [40]: gmm_n_test = [mixture.GaussianMixture(n_components=n,
                                               covariance_type='full',
                                               random_state=323)
                                 .fit(seg_sub) for n in range(1,14)]
         plt.plot(range(1, 14), [g.bic(seg_sub) for g in gmm_n_test])
```

Lower BIC values mean a better model fit. We can see in Fig. 10.9 that four components is not optimal, but beyond five components the BIC values do not continue to drop until we get to ten components. Note that BIC values for good fits would be expected to be *negative*, suggesting that these model fits are not very robust.

Another feature we must choose for the Gaussian mixture models is the covariance type. We can vary both the covariance type and the number of components to improve our model further. Note that here we are using *dictionary comprehension* as well as *list comprehension*. Dictionary comprehension is analogous to list comprehension, but generates a `dict` object rather than a `list`. See Sect. 2.4.10 for a refresher on list comprehension.

```
In [41]: gmm_n_v_test = {v: [mixture.GaussianMixture(n_components=n,
                                                     covariance_type=v,
                                                     random_state=323)
                                 .fit(seg_sub) for n in range(1,14)]
                             for v in ['full', 'tied', 'diag', 'spherical']}
         gmm_n_v_test_bic = {v: [g.bic(seg_sub) for g in m]
                                 for v, m in gmm_n_v_test.items()}
         pd.DataFrame(gmm_n_v_test_bic).plot()
```

We can see in Fig. 10.10 that the full covariance that we tried initially does a reasonable job, but diagonal covariance gives the best fit, and six components is an appropriate number.

Let's next inspect the clusters we find with six components:

```
In [42]: gmm5 = mixture.GaussianMixture(n_components=5,
                                        covariance_type='diag',
```

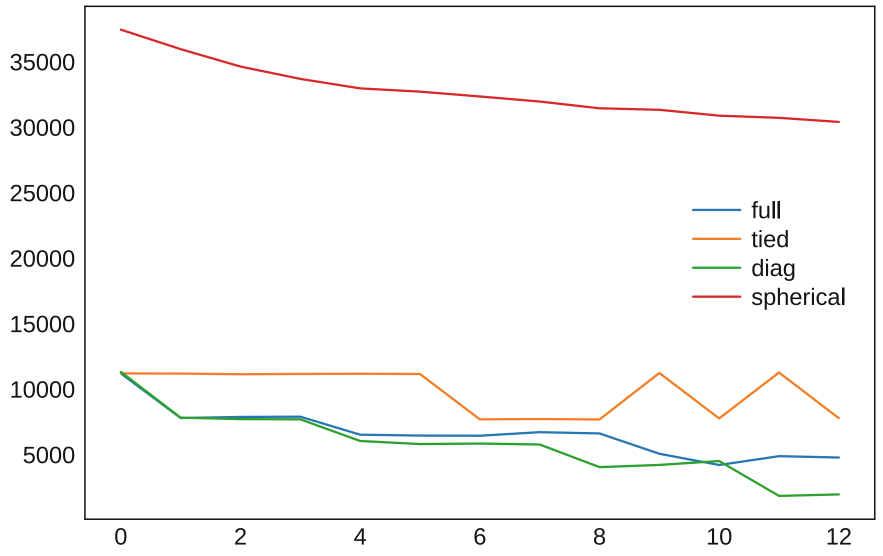

Fig. 10.10 BIC values of `GaussianMixture()` models as a function of the number of components and covariance type. Diagonal and full covariance provide the best fits

```
                                                random_state=323).fit(seg_sub)
         gmm5_labels = gmm5.predict(seg_sub)
```

```
In [43]: check_clusters(seg_sub, gmm5_labels)
```

```
[(0, 129), (1, 32), (2, 21), (3, 4), (4, 114)]
```

```
Out[43]:          age          income  is_female      kids  own_home  subscribe
         0  37.366073  52743.873543   0.581395  2.248062  0.465116     0.0000
         1  23.630276  20251.707688   0.375000  1.593750  0.000000     0.3125
         2  36.492245  51554.737478   0.619048  1.857143  0.333333     1.0000
         3  52.523755  44005.211404   0.000000  0.500000  1.000000     1.0000
         4  50.212110  56931.154434   0.385965  0.000000  0.543860     0.0000
```

These clusters do not look to be very actionable. Nearly all observations fall in either group 0 or group 4, which include all non-subscribers. Those two groups are differentiated on a few measures, most notably children: individuals in group 4 have no children whereas those in group 0 have, on average, 2.24 children. Again, the idea that individuals with and without children might be important segments very well may be accurate and business-relevant, but it wouldn't really take a sophisticated clustering analysis to find that!

10.3.6 Recap of Clustering

We've covered three methods to identify potential groups of observations in a dataset. In the next section we examine the problem of how to predict (classify) observations into groups after those groups have been defined. Before we move to that problem, there are two points that are crucial for success in segmentation projects:

- Different methods are likely to yield different solutions, and in general there is no absolute "right" answer. We recommend trying multiple clustering methods with different potential numbers of clusters.
- The results of segmentation are primarily about business value, and solutions should be evaluated in terms of both model fit (e.g., using BIC) *and* business utility. Although model fit is an important criterion and should not be overlooked, it is ultimately necessary that an answer can be communicated to and used by stakeholders.

10.4 Learning More*

We covered the basics of clustering . There are many places to learn more about those methods and related statistical models. A recommended introduction to the field of statistical learning is James et al., *An Introduction to Statistical Learning* (ISL) (James et al. 2013). A more advanced treatment of the topics in ISL is Hastie et al., *The Elements of Statistical Learning* (Hastie et al. 2016).

For cluster analysis, a readable text is Everitt et al., *Cluster Analysis* (Everitt et al. 2011). An introduction to latent class analysis is Collins and Lanza, *Latent Class and Latent Transition Analysis* (Collins and Lanza 2010).

Python has support for a vast number of clustering algorithms that we cannot cover here, but a few are worth mentioning. Exploring the scikit-learn `cluster` module (scikit-learn developers 2019a) and reading the associated references is a great place to learn more.

Marketing segmentation has developed approaches and nuances that differ from the typical description in statistics texts. For instance, in addition to the static, cross-sectional models considered in this chapter (where segmentation examines data at just one point in time), one might wish to consider dynamic models that take into account customer lifestyle changes over time. An overview of diverse approaches in marketing is Wedel and Kamakura, *Market Segmentation: Conceptual and Methodological Foundations* (Wedel and Kamakura 2000).

There are various ways to model changes in class membership over time. One approach is latent transition analysis (LTA), described in Collins and Lanza (2010). At the time of writing, LTA was not supported by a specific package in Python. Another approach is a finite state model such as Markov chain model (cf. Ross 2019). An alternative when change over time is metric (i.e., is conceptualized as change in a *dimension* rather than change between *groups*) is to use longitudinal structural equation modeling or latent growth curve models.

10.5 Key Points

We addressed segmentation through the lens of clustering. We examined several varieties of clustering methods and compared them. Once segments or groups are identified, classification methods can help to predict group membership status for new observations.

- The most crucial question in a segmentation project is the business aspect: will the results be useful for the purpose at hand? Will they inspire new strategies for marketing to customers? It is important to try multiple methods and evaluate the utility of their results (cf. Sect. 10.1.1).
- Distance-based clustering methods attempt to group similar observations. We examined `scipy.cluster.hierarchy()` for hierarchical clustering (Sect. 10.3.2) and `sklearn.cluster.k_means()` for k-means grouping (Sect. 10.3.4). Distance-based measures rely on having a way to express metric distance, which is a challenge for categorical data.
- Model-based clustering methods attempt to model an underlying distribution that the data express. We examined `sklearn.mixture.GaussianMixture` for model-based clustering of data assumed to be a mix of normal distributions (Sect. 10.3.5).
- Model-based methods enable calculation of Bayesian Information Criterion (BIC), which allows us to identify models with the best statistical fit (Sect. 10.3.5). We recommend that the ultimate decision to use a model's solution be made on the grounds of both statistics (i.e., excellent fit) and the business applicability of the solution (i.e., actionable implications).