Life is, of course, a series of coincidences, but we never cease to be surprised as each new one happens, and nothing can destroy their recurring freshness.

➤ *Robert Lynd, Irish writer*

CHAPTER 10

Constructing Semantic and Product Networks

An interesting, novel, and relatively understudied class of complex networks is networks based on co-occurrence, or coincidence—the property of items being in the same place (or close enough) at the same time. The edges in co-occurrence networks are implicit: they are not given (and often not even obvious); you have to deduce, extract, and calculate them from other data, and this is a significant departure from the relatively intuitive way you build social networks. Co-occurrence networks are living proof that you can connect anything to anything and make sense of the connections.

In this chapter, you will learn how to start with a seemingly odd collection of material or immaterial items, examine temporal and spatial connections between them, identify significant relationships, and convert your observations into a network graph. Just like social network graphs, these graphs have nodes and edges with respective attributes, but this time you will go one step further and explore their complex internal structure. You will be able to divide and conquer a complex network: decompose it into components, cores, coronas, communities, and similar structural elements; assign proper names to the extracted parts; understand their purpose and importance; and put them together again.

We will start by looking at two examples of co-occurrence networks: semantic networks and product networks. In the next chapters, we'll go over definition, extraction, naming, and use of complex network constituents.

Semantic Networks

A semantic network is a network of nodes that represent terms—words, word stems, word groups, or concepts—connected based on the similarity or dissimilarity of their usage or meanings. Link terms that:

- Are commonly used together in the same place in text: same sentence, paragraph, chapter, scene, act, list of keywords, list of interests in a social network, and so on ("semantic" ↔ "network")
- Describe the same property ("red" ↔ "blue")
- Occupy the same semantic niche (synonyms: "program" ↔ "application"; hypernyms: "pet" ↔ "cat"; antonyms: "erase" ↔ "restore")

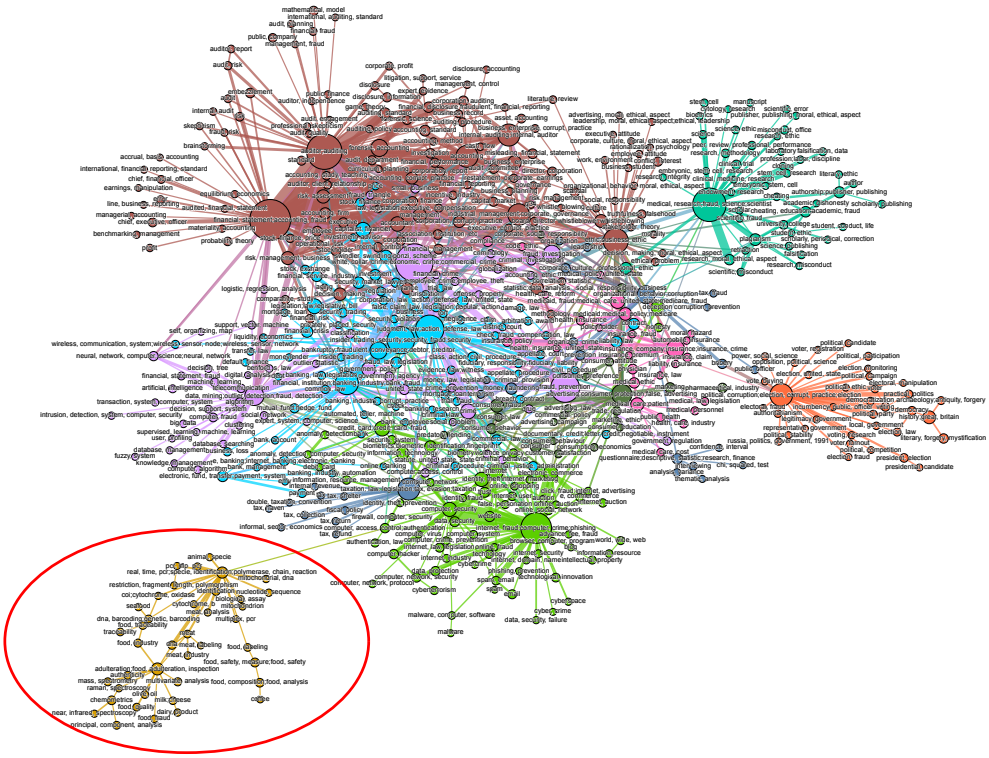
In the latter case, you may want to assign negative weights to some edges, which would make many network processing algorithms heartbroken. If your network has negatively weighted edges by construction, be prepared to remove them before analyzing the network.

Knowledge specialists use semantic networks for graphical (and machine-readable) knowledge representation, and social and behavioral researchers and anthropologists use semantic networks for semantic domain analysis. Let's have a look at two not-so-typical semantic networks: a network of keywords for fraud-related research papers and a network of characters from *Othello*.

Detect Food Fraud

Semantic networks often reveal surprising facts about texts and other term collections (corpora). Suppose you do research in accounting—namely, in fraud—and want to know everything about fraud types. You understand that nobody knows fraud better than other fraud researchers and fraudsters themselves. The latter are typically off limits, but the former are well represented in numerous databases of academic research papers. You could collect all research papers that mention "fraud," extract subject tags assigned to them by database editors, and create a semantic network of the tags, based on their co-occurrence. The subject tags (such as *DNA* and *meat industry*) are the nodes of the network. Two tag nodes are adjacent if the tags are frequently assigned together to the same paper. For example, the nodes *food fraud* and *food safety* are adjacent because many research papers focus on food fraud *and* food safety.

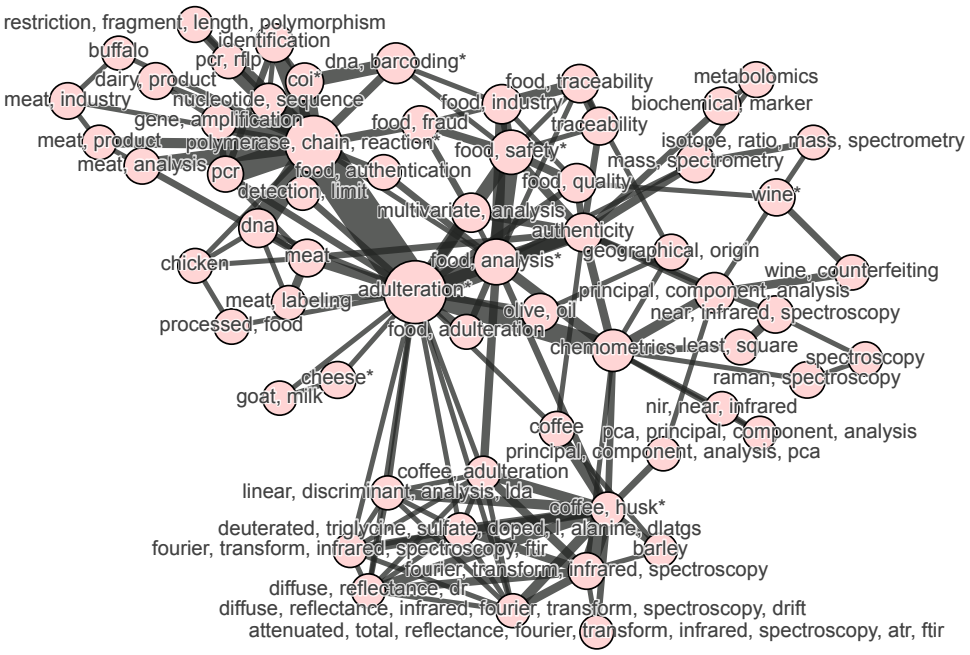
The original network (adapted from *Conceptual Structure of Fraud Research and Its Dynamics [GZ17]*), shown in the figure on page 119, is huge and could tell us many an exciting story. However, we will look only at the circled fragment in the bottom left corner.



The most striking conclusion from the figure is that the selected fragment is almost entirely removed from the rest of the network. It is connected to the bulk of the network with only one edge.

The figure on page 120 depicts the close-up view of the fragment. Remember that as a rule, node size represents node importance (in our case, the number of tags), and edge width represents edge weight. The nodes are colored based on their membership in network communities (*Outline Modularity-Based Communities*, on page 138), but this property is not relevant now.

Just by glancing through the node labels, you can see that the topic of the fragment is food fraud, also known as adulteration (no connection to adultery, adult stores, or any other "adult" business). Apparently, there is "fraud," and there is "food fraud!" Within the "food fraud" fragment, you can see tags related to fraud objects ("milk," "olive oil," "meat"); fraud detection methods ("spectroscopy," "DNA," "principal component analysis"); fraud prevention mechanisms ("food labeling," "identification"), and so on. If you are a PhD student or young postdoc looking for a future fraud-related research direction, you may be excited to have come across this semantic network fragment. Judging by its secluded position, few "hardcore" fraud analysts know or care about food! Why not become one of those who do?



By the way, we will show you how to construct a similar network step-by-step in Chapter 12, *Case Study: Performing Cultural Domain Analysis*, on page 143.

Expose a Protagonist

Who is the protagonist (or the main character, if you prefer less academic speak) of *Othello*? Be careful with what you answer because a trivial question like this must be tricky. Hint: No, Othello is not a protagonist. At least not by the standards of semantic networks.

The emerging field of digital humanities uses co-occurrence semantic networks to analyze texts: plays, scripts, and other forms of prose and poetry. The method allows us to identify the main and peripheral characters (see core-periphery analysis on page 131); group characters and places (see *Outline Modularity-Based Communities*, on page 138); and eventually break down the storyline into scenes suitable, say, for film or stage adaptation.

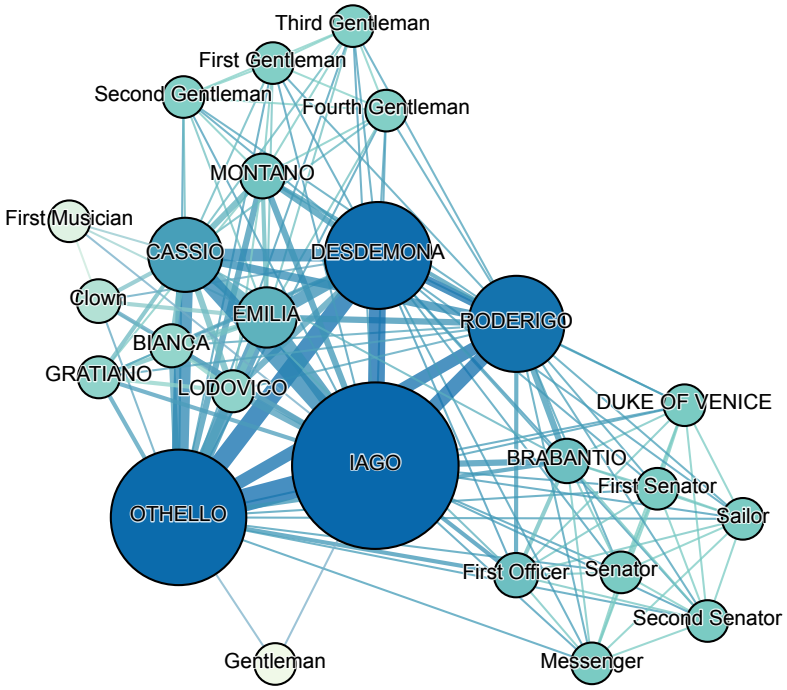
Let's outline a semantic network construction from the text of *Othello*. After you read the next chapter and the case studies, you will be able to implement the algorithm in Python. This exercise is inspired by *Measuring Tie Strength in Implicit Social Network [EG12]*.

1. You need a list of all characters. *Othello* is a short text; you can compose the list by hand. Alternatively, find all references to *Enter* and *Exit* remarks; or collect references to all characters as they speak if there is a property

in the text that identifies the characters. For example, a character may be marked with an HTML tag, as in <A NAME=speech1>RODERIGO.[1]

2. You need a definition of co-occurrence. Play scripts are perfect from this point of view: two characters co-occur if they occur in the same scene! In a general text, co-occurrence may be based on paragraphs, sections, chapters, pages, and so on.

3. Now that you have characters (nodes) and their co-occurrences (edges), you can build a network. Remarkably, once constructed, this network is a social network, of which you heard so much in Chapter 6, *Understanding Social Networks*, on page 55. The result is shown in the following figure.



4. Finally, you need a measure of importance. How do you know, indeed, who is the protagonist of the story? Luckily, you have the whole box of network centralities (*Choose the Right Centralities*, on page 94) that you can apply to each node. When you work with a social network, and the network in the figure *is* a social one, the best importance measures are betweenness and eigenvector centralities. The eigenvector centrality is proportional to the graph node sizes, and the betweenness centrality is reflected by the node color (the darker, the more central). Both centralities seem to be in good agreement: Iago is the protagonist. Not Othello.

1. shakespeare.mit.edu/othello/full.html

So, is Iago indeed the protagonist of *Othello*? Some researchers strongly believe that he is![2] Welcome to digital humanities!

Product Networks

A product network is a network of retail items. Network nodes in a product network represent items purchased by individuals and co-occurring in their shopping baskets or carts. You can connect two product nodes if customers often or always buy the respective products together. We call such products complements. Left and right shoes (if sold separately), nuts and bolts, nails and hammers, and one-way airline tickets from Boston to Seattle and from Seattle to Boston are good examples of complements: when you buy one, you almost always buy the other as well.

Product networks can (but do not have to) be weighted: you can define the weight of the edge as the frequency of co-purchasing. You can slice (*Slice Weighted Networks, on page 81*) the network later to remove low-weighted edges, if you want.

Sometimes product networks allow negatively weighted edges. If one of the products in a pair is a reasonable replacement for the other—in some sense!—we call them substitutes. If you live in Alaska and buy a husky to pull your sled, then you probably won't buy a reindeer for the same purpose, at least not at the same time. (You can still get a reindeer as a pet.) A husky and reindeer are substitutes; you can connect the respective nodes with a negatively weighted edge to represent their substitutive nature.

Here are two product networks for you, as a warm-up: a network of common cooked food ingredients and a network of tools and materials for a painting do-it-yourself project.

Explore Your Pantry

To find a product network, look no further than your pantry.

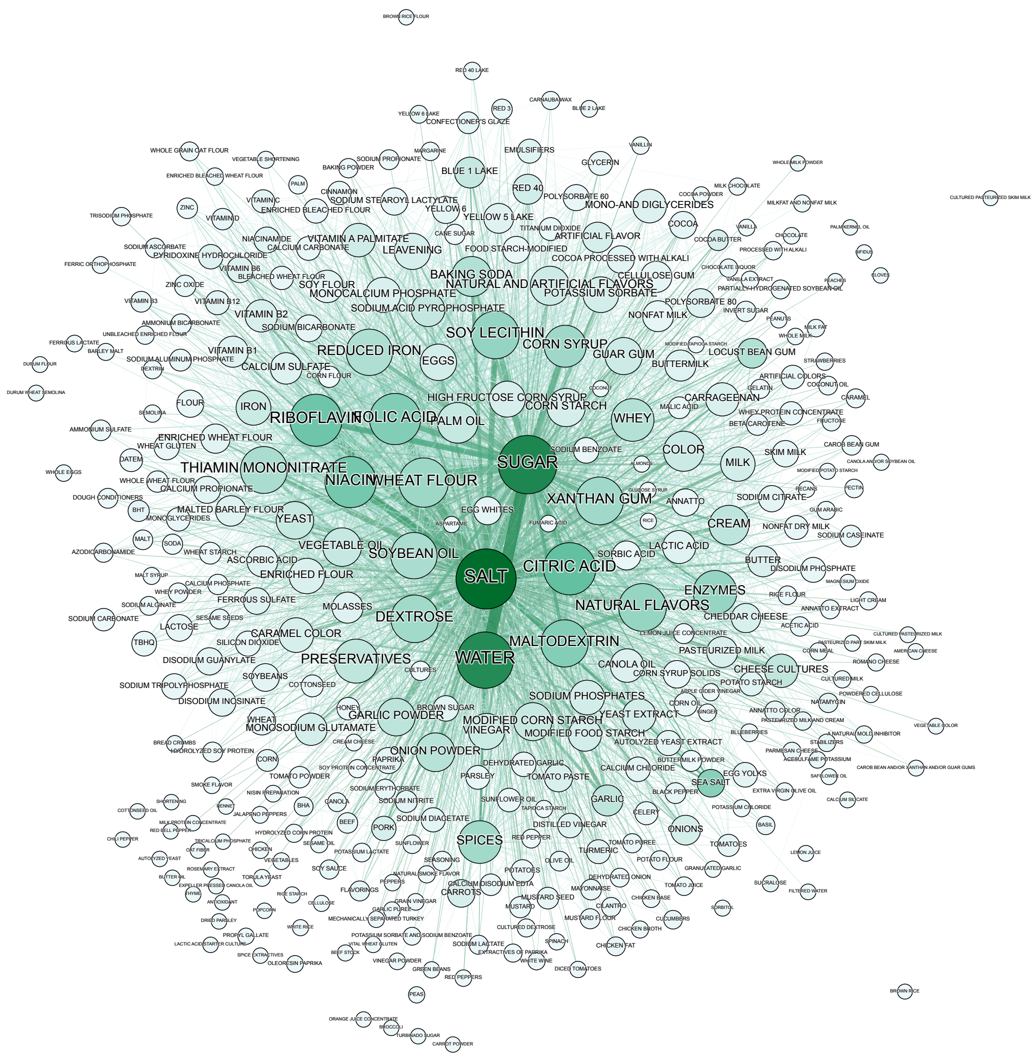
When you buy prepared food (say, a can of baked beans), you buy an elaborate concoction of ingredients: prepared beans, water, sugar, applewood smoked bacon, molasses, textured vegetable protein, and many others. You can think of the ingredients as separate products that happen to be packed together in the can. They occur in the same place at the same time—therefore, they are excellent candidates for becoming product network nodes. By constructing a product network, you can learn which ingredient combinations are most

2. http://www.shmoop.com/othello/antagonist.html

common, whether and how the ingredients group, and which ingredients are central to our food.

You can collect data for a network of ingredients from the website of the United States Department of Agriculture (USDA[3]). There is no need to download all several hundred thousand product descriptions. For starters, we suggest crawling a couple of thousand pages—for example, 925 products with 356 distinct ingredients.

In the following figure, two ingredient nodes are connected if they happen together in more than five food items (the threshold of five was chosen to keep the network connected but not too hairy).

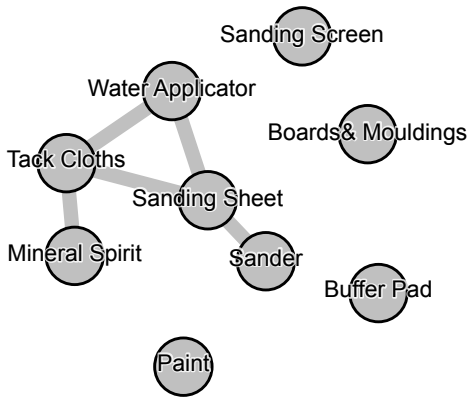


For each node, we calculate its betweenness (color) and eigenvector (size) centralities. The most central nodes represent the core ingredients. Not surprisingly, the top three ingredients are salt, sugar, and water: they go almost into any food item, and they almost always go side by side. The composition of the second ring is less predictable. The next ten most central ingredients are: citric acid (acidifier), maltodextrin (sweetener), xanthan gum (thickener), enzymes (catalysts), natural flavors, wheat flour, niacin (vitamin B), riboflavin (another kind of vitamin B), folic acid (yet another kind of vitamin B), and lecithin (emulsifier). Most of these ingredients are responsible for foundational taste and texture food properties, which explains their position in the network.

Design a Do-It-Yourself Store

Networks of products are common in marketing analysis. Marketing specialists construct product networks to reveal tightly coupled groups of products frequently purchased together. Retailers may compactly stock the products in a group in stores for the ease of shopping. If someone buys a product from a group, they may be reminded to buy other products from the same group. Finally, a group of products may be a stepping stone in a long-term customer project (for example, someone purchasing masonry products may be building a garage and would later need carpentry tools and materials, followed by brushes and paints).

The following figure shows a product network for a pre-painting project, derived from the sales data collected and provided by a Fortune 500 specialty retailer (adapted from *Building Mini-Categories in Product Networks [ZLZ15]*). The network has only nine nodes, four of which are isolated from the other five (we call them isolates—you will learn more about them in *Locate Isolates*, on page 127). Apparently, the isolated products were insignificantly connected to their neighbors, and the network analyst decided to drop the thin edges.



If you were to design an ideal do-it-yourself store, you would put water applicators, tack cloths, sanding sheets, sanders, and mineral spirits on the same shelf, and the other four products on the next shelf. If your customers bought a sanding sheet, but no sander, you (or your recommendation system) would remind them to purchase the sander and another seven items as well.

You learned about two uncommon types of complex networks—semantic networks of words and concepts and co-purchasing-based product networks. The latter can be found in marketing research; the former apply to text analysis and knowledge representation. You will see a complete example of a product network construction and analysis in Chapter 13, *Case Study: Going from Products to Projects*, on page 155. However, before you construct something big, you will learn how to deconstruct a network into compact blocks. In the world of large complex networks, dividing and conquering is the only way to manage complexity. The next chapter will show you how to unearth the network structure.

Divide et impera.

➤ *Attributed to Philip II, King of Macedon*

CHAPTER 11

Unearthing the Network Structure

You're probably not going to be surprised that complex networks have…a complex structure. From the ancient times to modern days, researchers and practitioners have confronted complexity by dividing a complex system into smaller parts—constituents—and then taking a closer look at the parts and making sense out of them. A part could be as small as a single network node or as large as a so-called giant connected component (GCC). (You will meet a GCC later on page 130; for now, it suffices to know that it is giant.) You need a "network-o-scope" to zoom in and out—and you're going to build it in this chapter.

You will learn how to dissect an original complex network into constituents of various sizes, shapes, and types: isolates, connected components, cliques, communities, and k-cores, to mention but a few. You will understand the function and place of each type of constituent in the network analysis workflow. At the end, you will get some suggestions about naming the extracted parts, because, as a rule of thumb, you cannot successfully use something that does not have a name. In other words, you will be ready to *divide and conquer* complex networks. (Which, sadly, did not save the author of these words, King Philip II of Macedon, from an assassination.)

Locate Isolates

The smallest distinct element of any network is an isolate: a node that is not connected to any other node (an isolate can still be connected to itself with a loop edge). Though isolates belong to a bigger network, their very existence is against the networking spirit, because the whole idea of networking is that of connectedness. An example of an isolate in a semantic network is a word that has no synonyms, no homonyms, no antonyms, and no other relationships to any other word (say, "sphygmomanometer" in a network of simple

synonyms—because it has none). An example of an isolate in a product network is an item that nobody ever buys together with any other item. The last meal comes to mind, but then, again, nobody pays for the last meal, so technically it is not even a purchase.

As a network analyst, you want to identify isolates and research the reasons for their isolation. Have you overlooked an edge while constructing the network? (Go over the network construction process one more time.) Have you replaced negative ties in a signed graph with zero-weight positive ties and later discarded them? (Check if there is a better way to preserve negative ties.) Have you sliced a weighted network too aggressively? (*Slice Weighted Networks*, on page 81. Select a lower slicing threshold, if possible.) If the reasons seem to be valid, there is no more need to keep the isolates in the network. Locate them with `nx.isolates(G)`, include their names or count into the final report, and chop the isolates off:

```
G = nx.Graph()
G.add_nodes_from("ABCD") # No edges -- all nodes are isolates
my_isolates = nx.isolates(G)
```

```
['D', 'C', 'B', 'A']
```

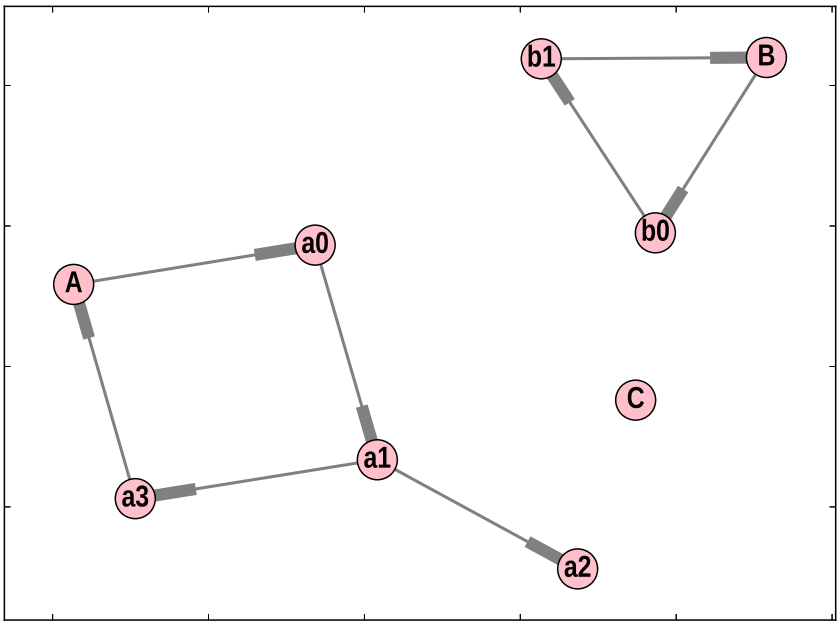
```
G.remove_nodes_from(my_isolates) # No more isolates!
my_isolates = nx.isolates(G)
```

```
[]
```

Split Networks into Connected Components

A connected component is a subset of network nodes such that there exists a path (*Think in Terms of Paths*, on page 90) from each node in the subset to any other node in the same subset. An isolate is a special case of a connected component: there is only one node in the subset, so no path is even needed! The figure on page 129 shows a network with three connected components: a larger component A, smaller component B, and isolate C. A fictitious network traveler can get from any node in A to any node in A, but not to any node in B.

If a network is directed, it may have weakly and strongly connected components. In a strongly connected component, there is always a directed path from any node of the component to any other node of the same component. In a weakly connected component, you are allowed to travel one-way streets in the wrong direction (drive responsibly!), if this is what it takes to get from the source to the destination. Both components A and B in the figure are weakly connected, but B is also strongly connected. (No nodes are reachable from *a2*!)



NetworkX provides two families of functions for component analysis. Let's experiment with the network from the previous figure:

make-figures.py

```
F = nx.DiGraph()
F.add_node("C")
F.add_edges_from([("B", "b0"), ("b0", "b1"), ("b1", "B")])
F.add_edges_from([("A", "a0"), ("a0", "a1"), ("a1", "a2"), ("a1", "a3"),
                  ("a3", "A")])
```

Functions nx.connected_components(G) (implemented only for undirected networks), nx.strongly_connected_components(F), and nx.weakly_connected_components(F) (both implemented only for directed networks) take a network of the appropriate type as the parameter and return a generator of sets of nodes that belong to the namesake components. You can coerce the generator to a list, if necessary:

```
list(nx.weakly_connected_components(F))
```

```
[{'A', 'a0', 'a2', 'a1', 'a3'}, {'B', 'b0', 'b1'}, {'C'}]
```

```
list(nx.strongly_connected_components(F))
```

```
[{'a2'}, {'A', 'a0', 'a1', 'a3'}, {'B', 'b0', 'b1'}, {'C'}]
```

```
G = nx.Graph(F)
list(nx.connected_components(G))
```

```
[{'A', 'a0', 'a2', 'a1', 'a3'}, {'B', 'b0', 'b1'}, {'C'}]
```

Note how we convert the directed graph F into an undirected graph G in the last expression. The connected components of the converted graphs are the same as the weakly connected components of the original graph. You can use the obtained node sets to extract the respective subgraphs from the original graph:

```
wcc = nx.subgraph(F, list(nx.weakly_connected_components(F))[1])
len(wcc)
```

```
2
```

Note that nx.subgraph() returns a view of F, not a new graph. Any future change in F is reflected in G. Make a copy of the view if you anticipate any changes in the underlying graph.

Alternatively, you can use the other family of functions:

- nx.connected_component_subgraphs(G) (implemented only for undirected networks)
- nx.strongly_connected_component_subgraphs(F)
- nx.weakly_connected_component_subgraphs(F) (the latter two functions are implemented only for directed networks)

These functions take a network as the parameter and return a generator of Graph of DiGraph objects, depending on the original network type. So, if you only want to know which nodes belong to what component, the functions without the _subgraphs suffix may save you some time. If you have further operations in mind, go with the second family.

Most co-occurrence networks, by construction, are undirected. Indeed, the fact that items A and B are in the same place at the same time implies that A is in the same place with B and the other way around. That's why for the rest of the chapter, let's assume that all networks are undirected and all connected components are just connected components, without any references to their strength or weakness.

One of the components in a complex network often dominates the others: not so much because it is strong, but because it is giant. The giant connected component (GCC) is simply the largest component by the node count. NetworkX does not provide a function for extracting the GCC, but you can still find it by calling one of the functions mentioned previously, reverse sorting the generated list by size, and taking the first element:

```
comp_gen = nx.connected_components(G)
gcc = sorted(comp_gen, key=len, reverse=True)[0]
```

The size of a GCC in complex networks typically ranges between 80 and 100 percent of the full network size. You may want to treat each smaller component

as an indivisible structural unit of the network and focus your attention on the GCC. As a bonus, you will spare yourself from remembering which algorithms and functions apply to disconnected networks and which do not.

What Makes Components Giant?

According to Albert-László Barabási, prominent complex network researcher, most complex networks evolve as a result of preferential attachment. Preferential attachment (also known as the "rich get richer," "80/20," or the Pareto principle) suggests that when a new node joins a network, it is likely to attach itself to a node with the highest degree. Thus, the degree of the node with the highest degree becomes even higher, and the connected component that contains that node grows faster than all other connected components, leading to the emergence of the GCC. The converse is also true: if a network has a GCC, it is likely a result of preferential attachment.

Separate Cores, Shells, Coronas, and Crusts

The only valuable property of a connected component is its connectedness. There is always a way to get from any node A in a component to any other node B in the same component. The property of connectedness is global and, while important for social and communication networks (where paths are responsible for information diffusion), may not be adequate for semantic, product, and other types of networks, where direct or short-haul connections are more essential. Consider a network of synonyms: "emerald" is a synonym of "green," and "green" is a synonym of "ecological," but "ecological" is hardly a synonym of "emerald."

Let's zoom into a connected component (say, in the GCC) and try to find more elements inside.

One of the fundamental tools in modern sociology is core-peripheral analysis. A social network, thereby, consists of two sets of nodes: the core (the nodes that are more or less tightly interconnected) and the periphery (the nodes that are tightly connected to the core, but only weakly, if at all, connected to the other peripheral nodes). The graphs of core-peripheral networks often have a "hairy" appearance: their dense "body" is adorned with "pendulums," multi-edge self-loops, and so on.

Social networks are not the only networks known to have the core-periphery structure. As another example, networks of journal citations reveal the same pattern, where the core consists of papers published in prominent journals in the field, and papers published in marginal journals populate the periphery.

Traditional social network analysis attacks the core-periphery decomposition via fuzzily defined blockmodeling. We, on the other hand, will start by introducing a more mathematically rigid classification of nodes into four categories: cores, shells, coronas, and crusts.

A core or, more accurately, a k-core (where k could be any non-negative integer number) is a subgraph of the original network graph such that each node in the subgraph has at least k neighbors. A 0-core is, naturally, the whole graph. A 1-core is a graph with no isolates. A 2-core is a graph where no node has fewer than two neighbors (no node is a part of a pendulum). Any graph usually has more than one core; the core with the largest possible k is called the main core. A k-core construction process is iterative:

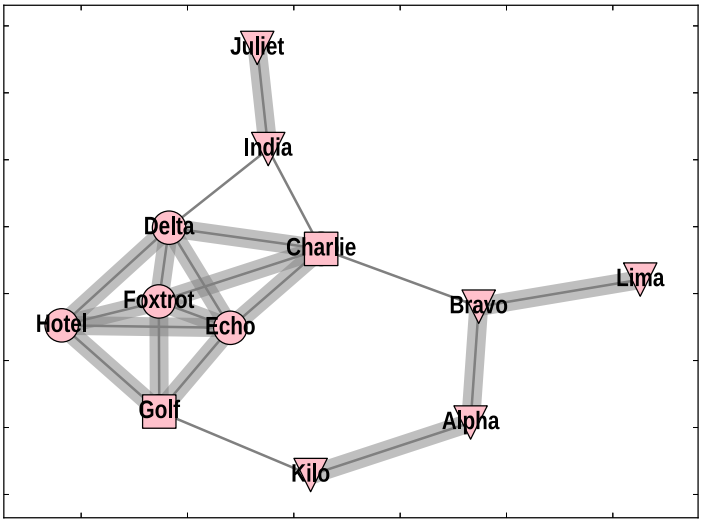
1. Start with the original graph and remove all nodes that have a degree smaller than k and all the incident edges; this will probably result in some of the remaining nodes losing their neighbors and their degree decreasing.
2. Some nodes that have k neighbors or more may have fewer than k neighbors after trimming; remove them, too, and iterate until no remaining node has fewer than k neighbors.
3. The remaining nodes form the k-core.

A k-crust is what is left of the original network when we remove the k-core. In other words, the crust is the periphery. A core has its internal structure. The subgraph of the k-core in which all nodes have exactly k neighbors in the core is called a k-corona. Unlike crusts, coronas are not necessarily connected and may consist of unconnected components—that is, unconnected within the corona.

Finally, a subset of nodes in k-core but not in (k+1)-core, is called a k-shell. Just like a corona, a shell may consist of components that are not connected within the shell. Let's experiment with the graph from the figure on the next page.

make-figures.py

```
G = nx.Graph(
    (("Alpha", "Bravo"), ("Bravo", "Charlie"), ("Charlie", "Delta"),
     ("Charlie", "Echo"), ("Charlie", "Foxtrot"), ("Delta", "Echo"),
     ("Delta", "Foxtrot"), ("Echo", "Foxtrot"), ("Echo", "Golf"),
     ("Echo", "Hotel"), ("Foxtrot", "Golf"), ("Foxtrot", "Hotel"),
     ("Delta", "Hotel"), ("Golf", "Hotel"), ("Delta", "India"),
     ("Charlie", "India"), ("India", "Juliet"), ("Golf", "Kilo"),
     ("Alpha", "Kilo"), ("Bravo", "Lima")))
```



NetworkX provides a useful collection of functions for calculating all k things. Each of them takes a graph and k as the parameters and returns the namesake core-periphery element (k is optional for nx.k_shell(), nx.k_crust(), and nx.k_core(): they return the main shell, crust, and core by default):

```
nx.k_core(G).nodes() # Round and square nodes and shaded edges
```

```
['Golf', 'Charlie', 'Delta', 'Hotel', 'Foxtrot', 'Echo']
```

```
nx.k_crust(G).nodes() # Triangular nodes and shaded edges
```

```
['Lima', 'Bravo', 'Kilo', 'Juliet', 'Alpha', 'India']
```

```
nx.k_shell(G).nodes() # Round and square nodes and shaded edges
```

```
['Golf', 'Charlie', 'Delta', 'Hotel', 'Foxtrot', 'Echo']
```

```
nx.k_corona(G, k=3).nodes() # Square nodes
```

```
['Golf', 'Charlie']
```

Extract Cliques

Unlike the smaller components, the GCC and the k-cores are usually too large to be considered a single structural element. Depending on your interpretation of the nodes and edges, you should zoom in even further in a search for smaller network building blocks, such as cliques.

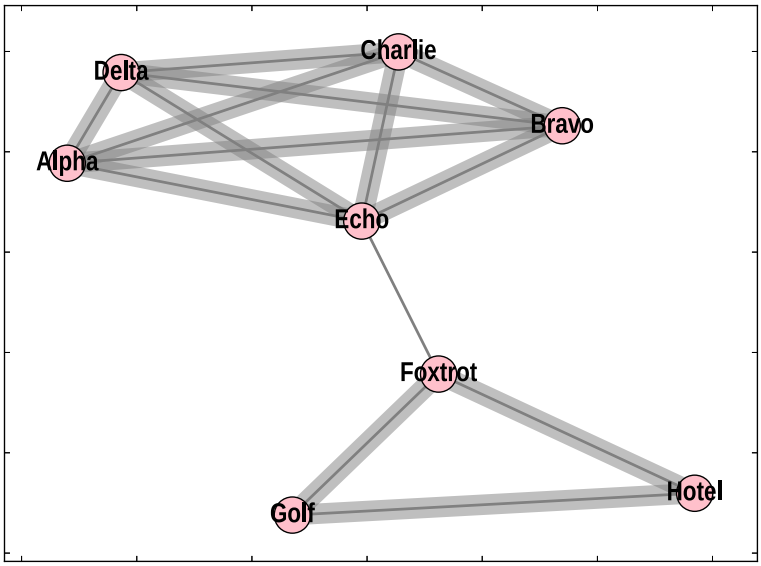
A clique, or, more accurately, a k-clique is a subset of k nodes such that each node is directly connected to each other node in the clique. (We distinguish weak and strong cliques in directed graphs.) Cliques are also known as complete subgraphs. The nodes in a clique may be connected to other nodes

as well, but they do not have to—that is, the degree of a node in a k-clique is at least k-1. The principal difference between cliques and connected components is that the path between any two nodes in a clique must have the length of 1, while in a component, the path length is limited only by the graph diameter (*Think in Terms of Paths*, on page 90).

Any single node is a 1-clique, a monad. Any two connected nodes form a 2-clique, a dyad. A triangle of nodes—the result of transitive closure—is a 3-clique, a triad (*Explore Neighborhoods*, on page 86). Monads, dyads, and triads are very common in complex networks.

A maximal clique is a k-clique that cannot be made a (k+1)-clique by adding another node to it. For example, clique (Alpha, Bravo, ..., Echo) in the following figure is a maximal clique, because including any other node (Foxtrot, Golf, or Hotel) into it invalidates the complete connectedness property. For example, if Foxtrot is included, then (Alpha, Bravo, ..., Foxtrot) is not a clique anymore. The largest maximal clique in a network graph is called the maximum clique. (No, I did not invent this terminology!)

Let's experiment with the graph from the following figure:



make-figures.py

```
# Generate a 5-clique
G = nx.complete_graph(5, nx.Graph())
nx.relabel_nodes(G,
        dict(enumerate(("Alpha", "Bravo", "Charlie", "Delta", "Echo"))),
                copy=False)
# Attach a pigtail to it
```

```
G.add_edges_from([
        ("Echo", "Foxtrot"), ("Foxtrot", "Golf"), ("Foxtrot", "Hotel"),
        ("Golf", "Hotel")])
```

NetworkX provides function nx.find_cliques() for finding all maximal cliques in a graph (the largest of which is the maximum clique). The function returns a list generator, and this time, the use of a generator is not a tribute to Pythonic programming style, but a dire necessity. As a matter of fact, larger cliques, especially maximal and maximum cliques, are rare and hard to find. Finding large cliques is a computationally very hard problem (known as an NP-complete problem) and listing all large cliques requires exponential time. Unfortunately, function nx.find_cliques() generates cliques in a random order, but if you want to get only some maximal cliques, not all of them, then you can stop the generator whenever you want. The following code finds all three maximal cliques from the figure (I highlighted the larger two in the figure):

```
list(nx.find_cliques(G))
```

```
[['Golf', 'Hotel', 'Foxtrot'], ['Echo', 'Alpha', 'Bravo',
  'Charlie', 'Delta'], ['Echo', 'Foxtrot']]
```

You have at least two good reasons to search a network for k-cliques: a theoretical and an empirical one. In the theoretical case, you may already have some prior knowledge about the network structure. For example, a marketing specialist may define a project basket in a product network as a collection of products such that they are always purchased together (and, therefore, form a clique when represented as a network). Recognizing k-cliques in a product network almost instantly leads you to the discovery of project baskets. Closely cooperating teams in social and organizational networks are k-cliques, and such are collections of complete synonyms in semantics networks.

In the empirical case, you use cliques as opaque network atoms. If you assume that an edge between two nodes is an indication of their significant similarity, then a complete connectedness within a clique implies overall significant similarity of the member nodes. Thus, you can replace all k nodes with one node that represents the entire clique, or with a newly minted "clique-node," potentially significantly simplifying the network topology. Function nx.make_max_clique_graph() generates a new graph by replacing each maximal clique with a new synthetic node:

```
synthetic = nx.make_max_clique_graph(G)
synthetic.edges()
```

```
[(1, 3), (2, 3)]
```

Naturally, you can replace cliques with synthetic nodes in the a priori case, too! Just be aware that this function first finds all maximal cliques, with all the NP-completeness implications of finding all maximal cliques.

Recognize Clique Communities

By definition, a clique is a very rigid and sensitive network structure. Removing an edge from a k-clique transforms it into two interwound, partially overlapping, adjacent (k-1)-cliques. In the figure on page 137, subgraphs Alpha–Delta and Alpha–Charlie, Echo are 4-cliques each, but the whole network is not a 5-clique, because the edge Delta–Echo is missing (I show it as a dashed line).

Intuitively, you may feel that all k nodes somehow belong together and the missing edge could have been a victim of a measurement or data entry error or improper slicing (*Slice Weighted Networks*, on page 81) or conversion from a directed graph. Nonetheless, nx.find_cliques() will not recognize your k nodes as a clique (because they are not). Instead, it will report two smaller cliques, leaving it up to you to notice that they actually share k-1 nodes and their separation may have been caused by a missing edge.

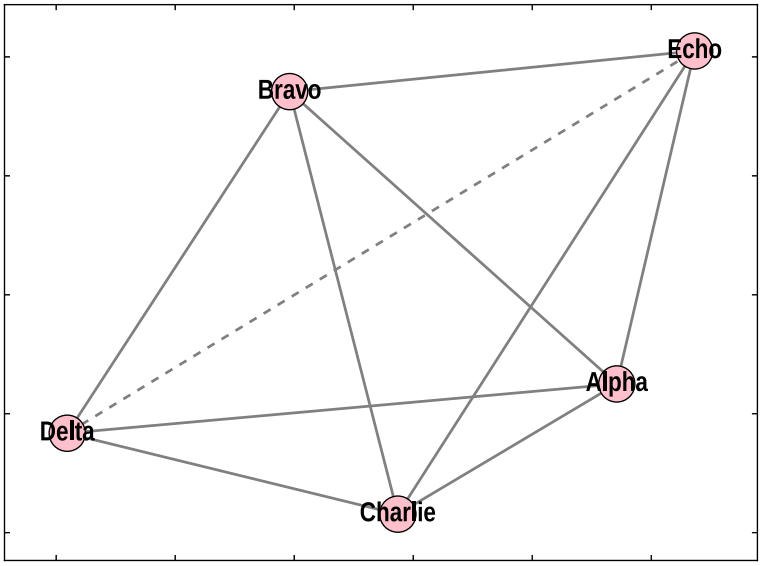
Luckily, NetworkX supports k-clique communities. A k-clique community is a union of all k-cliques that can be reached through adjacent k-cliques. The process of reaching all cliques in the union is called *clique percolation [PDFV05]*.

When a Community Is Not a Community and a Cluster Is Not a Cluster

Anthropologists and social scientists have a different idea of a community and may get easily confused at this point. For them, a community is a tightly knit group of people, not a group of abstract nodes. To avoid getting into pointless terminological fights, I will sometimes refer to communities as clusters. Unfortunately, data scientists who may be reading this book, too, have a different definition of a cluster. Apparently, when it comes to network analysis, terminological fights are unavoidable.

K-clique communities in complex networks are a curse and a blessing. Why are clique communities a blessing?

They provide a flexible substitute for inflexible proper cliques. True, the nodes in a community are not in general directly interconnected; however, if the relationship represented by the edges is actually transitive (if A is adjacent to B, and B is adjacent to C, then A is supposed to be adjacent to C—as it would be in the case of two products always purchased together) and the



missing edges result from network construction imperfection, then the length of the path between any two nodes in a clique community does not really matter.

Why are they a curse, then?

Just like strict cliques, clique communities do not necessarily partition the network and can overlap with other clique communities. In other words, the same node may belong to more than one clique or clique community. This may or may not be what you want.

What's worse, if the relationship represented by the edges is only approximately transient (if A is adjacent to B, and B is adjacent to C, then A is not necessarily adjacent to C—as it would be in the case of personal friendship), then two nodes separated by a multi-edge path may actually have little or nothing shared, and their membership in the same clique community would be questionable.

Only you can determine whether clique communities are appropriate for your network. But once you do, here's the function for doing the dirty job:

```
list(nx.k_clique_communities(G, k=3))
```

```
[frozenset({'Golf', 'Hotel', 'Foxtrot'}),
 frozenset({'Alpha', 'Charlie', 'Bravo', 'Echo', 'Delta'})]
```

Note that a k-clique community always has at least `k` nodes!

Frozen Sets

A frozenset is an immutable version of a Python set. Because of its immutability, it can be used as key in a dictionary, but it can be cast to a set, if any modifications are necessary.

Outline Modularity-Based Communities

This section uses community.

The fuzziest and most flexible form of node organization in a complex network is network communities based on modularity. They are sometimes also called clusters or groups, and are not to be confused with clique communities (*Recognize Clique Communities*, on page 136).

Let's start with modularity first, and assume that the network has been already partitioned into non-overlapping communities (later you'll figure out how). According to *Newman's definition [New06]*, modularity `m` is the fraction of the edges that fall within the given communities minus the expected fraction if edges were distributed at random, while conserving the nodes degrees. The value of `m` is in the range from -0.5 (inclusive) to 1 (exclusive). If most of the edges are incident to the nodes within the same community, the modularity is very high, close (but not equal) to 1, and the proposed partition describes a very good community structure. The modularity of -0.5 means that the nodes within the same community are not adjacent at all—the proposed community structure is worse than random; in fact, you are probably dealing with anti-communities that induce bi- and multi-partite networks (as in *Project Bipartite Networks*, on page 180).

Ideally, you want to partition a network in such a way that the modularity is as high as possible. The modularity of 0.6 and above corresponds to networks that have a clearly visible community structure. Unfortunately, getting the largest modularity is hard for at least three reasons:

- The problem of optimal partitioning is NP-complete with respect to the network size. To find the best partition, you should calculate modularity for every possible partition and then select the best one. The number of partitions is simply too large, and the problem does not have a feasible exact solution for any non-trivial graph.
- Approximate solutions (for example, the *most popular Louvain algorithm [BGLL08]*) do a pretty good job, but some of them are probabilistic, which means every time you run them, you may end up having slightly different partitions.

- The resolution of modularity-based methods scales poorly, and they overlook small communities in large networks. A plausible solution is to partition the network recursively into smaller and smaller communities.

Anaconda, the most popular Python distribution, does not currently include tools for modularity-based community detection. Fortunately, the tool exists and can be easily installed via pip. It is called python-louvain. The externally visible name of the module is community, and you import it under this name.

Module community uses the louvain algorithm that optimizes network modularity. The discovered communities are represented as a partition: a dictionary with node labels as keys and integer community identifiers as values. The module also calculates the modularity of the partition with respect to the original network.

```
part = community.best_partition(G)
```

```
{'Golf': 0, 'Bravo': 1, 'Delta': 1, 'Hotel': 0, 'Foxtrot': 0,
 'Charlie': 1, 'Alpha': 1, 'Echo': 1}
```

```
community.modularity(part, G)
```

```
0.3035714285714286
```

A Faster Way

There is a faster implementation of the louvain algorithm provided through the module louvain (must be installed separately, too). However, it works only with graphs constructed in iGraph, not in NetworkX.

Just as with cliques and clique communities, you may want to replace the smaller structural elements of a large network with synthetic nodes and build an induced graph:

```
induced = community.induced_graph(part, G)
induced.nodes()
```

```
NodeView((0, 1))
```

```
induced.edges(data=True)
```

```
EdgeView([(0, 0, {'weight': 10}), (0, 1, {'weight': 1}),
          (1, 1, {'weight': 3})])
```

Note that the induced graph is weighted and has loops. The weight of an induced edge incident to the synthetic community nodes is the number of edges in the original network that are incident to the nodes in the respective communities.

Explore Modularity-Based Communities with Pandas

If you're familiar with Pandas, you can convert a network partition part (a dictionary) into a Series for the ease of further processing. You can see the total number of communities, their sizes, and which nodes belong to which community:

```
part_as_series = pd.Series(part)
part_as_series.sort_values()
```

```
Foxtrot    0
Golf       0
Hotel      0
Alpha      1
Bravo      1
Charlie    1
Delta      1
Echo       1
dtype: int64
```

How big are the communities?

```
part_as_series.value_counts()
```

```
0    5
1    3
dtype: int64
```

Perform Blockmodeling

The construction of the graph of maximal cliques or communities is a special case of blockmodeling—grouping network nodes according to some meaningful definition of equivalence and replacing them with synthetic "supernodes." A more general function nx.quotient_graph(G,part,relabel=True) takes a graph G and its partition part as a list of node collections (lists or sets), and creates an induced graph. Unlike nx.make_max_clique_graph() and community.induced_graph(), nx.quotient_graph() requires the partition includes every node in the original graph at most once. You can manually remove the offending overlapping clique from a clique partition, if you want:

```
cliques = list(nx.find_cliques(G))
```

```
[['Golf', 'Hotel', 'Foxtrot'], ['Echo', 'Alpha', 'Bravo',
  'Charlie', 'Delta'], ['Echo', 'Foxtrot']]---not good!
```

```
synthetic = nx.quotient_graph(G, [cliques[0], cliques[1]], relabel=True)
synthetic.edges()
```

```
EdgeView([(0, 1)])
```

Not All Blockmodeling Leads to the Same Rome

For social scientists, "blockmodeling" often means a very different thing: separating a network into the core and periphery by way of rearranging rows and columns of the incidence matrix. Blockmodeling as understood by complex network analysts is a generalization of the core-periphery decomposition.

Name Extracted Blocks

From the data scientific point of view, network analysis at the macroscopic level (extraction of communities, cliques, and other structural blocks) is an example of unsupervised machine learning. The goal of unsupervised machine learning is to infer a network's hidden structure in the absence of "labels": node and edge attributes (except, perhaps, the edge weights).

The unearthed blocks suffer from two major interrelated problems:

- It is not clear what they mean.
- They are nameless.

In fact, if you knew the purpose or nature of a block, you would give it a name, and if you knew the name, you would guess what its purpose or nature is.

Selecting a good name for a block can be done in at least three ways.

- You can use your intelligence: look at the individual node labels and generalize. A block that has labels "car," "truck," "train," and "sled" probably deserves to be called "land transportation," and "hand," "arm," "leg," "head," and "chest" belong to the block "body parts." If unsure or confused, hire a subject matter expert (SME) whose job is to know why nodes X and Y ended up in the same block.
- Better yet, hire a lot of subject-matter experts—or sort-of-experts. *Amazon Mechanical Turk (AMT) [BKG11]* offers a way to put any question to literally thousands of people ("workers," in the AMT terminology) for a very modest price. Ask 10,000 AMT workers what "foos," "bars," and "foobars" have in common. If the terms have anything in common at all, you will most probably get an answer supported by a solid majority of workers.
- Finally, if you cannot hire an SME and would rather not mess with AMT, you can still generate block labels from its data. If the nodes in a block differ—for example, in size or weight—take the largest of them (say, "head") and use its label to synthesize the block label (for example, "the 'head' group"; see *Interpret the Results*, on page 152 for a better example). If all nodes have the same attributes or have no attributes at all, choose the first node in the alphabetic order ("the 'arm' group").

In this chapter, you learned how to dissect a complex network and explore its anatomy. You know how to locate isolated nodes and components; identify cores, shells, coronas, and crusts; and compute node cliques, clique communities, and modularity-based communities (sometimes referred to as clusters). Now, it is time to apply the freshly minted "network-o-scope" to a couple of real-world semantic and product networks.

…Culture opens the sense of beauty.

➤ *Ralph Waldo Emerson, American essayist, lecturer, and poet*

CHAPTER 12

Case Study: Performing Cultural Domain Analysis

This chapter uses NLTK, Pandas, NumPy.

Cultural domain analysis (CDA, *Analyzing Qualitative Data. Systematic Approaches [BWR17]*) is the study of how people in groups think about lists of terms that somehow go together and how this thinking differs between groups. Some people associate "candle" with "Christmas," others with "hurricane" and "blackout," and yet others with a "self-injury" (cutting/burning their skin) toolset. Anthropologists, ethnographers, psychologists, and sociologists use CDA to understand semantic mindscapes of social, ethnic, religious, professional, and other groups. Before personal computers and specialized CDA software became available, social scientists used to do CDA essentially by hand. But not anymore! Python comes to rescue.

You don't have to be an anthropologist, ethnographer, psychologist, or sociologist to read this chapter. Regardless of your background, you will learn how to harvest semantic data from a popular blogging website and cache it locally for further efficient access. You will see how to convert natural language units into terms and build, analyze, and interpret a semantic network reflecting the interests of the fans of *The Good Wife*, a CBS TV show. Hopefully, you will be able to extend the same approach to other shows, other websites, and other tag corpora.

The complete code for this case study is available in the file lj.py.[1]

1. pragprog.com/titles/dzcnapy/source_code

Get the Terms

Start by importing all necessary modules and defining the domain (LJ community) of interest. We suggest using Pandas and NumPy, the power tools of data science, and NLTK—the Natural Language Toolkit—in this project, as well as some other libraries, so you need to import them. (If you last used them a while ago, you might want to *blow the dust off your skill set [Zin16]*.)

lj.py
```
import urllib.request, os.path, pickle # Download and cache
import nltk # Convert text to terms
import networkx as nx, community # Build and analyze the network
import pandas as pd, numpy as np # Data science power tools
```

Your next step is to get and cache term lists. A term is a unit of CDA. It can be a word, a word group, a word stem, or even an emoticon (emoji). CDA looks into similarities between terms that are shared among a reasonably homogeneous group of people. So, you need to find a reasonably homogeneous group of individuals, a list of terms, and a way to assess their similarity.

A great source of semantic data is LiveJournal (LJ)—a collection of individual and communal blogs with elements of a massive online social network.[2] LiveJournal has an open, easily accessible API, and encourages the use of public data for research. Unfortunately, LJ membership and activity peaked in the early 2000s, but the site still hosts some lively blogging communities (such as the celebrity gossip blog "Oh No They Didn't!" [3]), and it serves rich layers of historical data.

LJ communities consist of individual members that have their private blogs, profiles, friend lists, and interests (online identity markers). In fact, LJ treats communities and users as same-class citizens: communities, like individuals, have their profiles, interests, and even "friends." The URLs of user/community profiles, interest lists, and friend lists have a regular structure. If *thegoodwife_cbs* is the name of a community (you will use it in the rest of the study), then thegoodwife-cbs.livejournal.com/profile/ is the URL of the community profile (note that the underscore was replaced by a dash), www.livejournal.com/misc/fdata.bml?user=thegoodwife_cbs&comm=1 is the friend list, and www.livejournal.com/misc/interestdata.bml?user=thegoodwife_cbs is the interest list.

For the purpose of this mini-study, let's define two terms to be similar from the perspective of an LJ community if they are consistently listed together on

2. www.livejournal.com
3. ohnotheydidnt.livejournal.com

different interest lists of the community members. Your first job is to obtain and process the community membership list. A typical list looks like this:

```
# Note: Polite data miners cache on their end. Impolite ones get banned.
# Note: thegoodwife_cbs is a community account
P> emploding
P> poocat
«…more members…»
P< brooketiffany
P< harperjohnson
```

The first line of the document makes a significant point: if you download something once, you should not download it again. Create the directory cache and store all downloaded data into it. If you run CDA on the same data again, it will hopefully still be in the cache, assuming that interest lists and community membership are reasonably stable.

lj.py

```
LJ_BASE = "http://www.livejournal.com/misc"
DOMAIN_NAME = "thegoodwife_cbs"

cache_d = "cache/" + DOMAIN_NAME + ".pickle"
if not os.path.isfile(cache_d):
    domain = download(LJ_BASE, DOMAIN_NAME)
    if not path.os.isdir("cache"):
        os.mkdir("cache")
    with open(cache_d, "wb") as ofile:
        pickle.dump(domain, ofile)
else:
    with open(cache_d, "rb") as ifile:
        domain = pickle.load(ifile)
```

This code fragment uses the module pickle—native Python data serializer (*Export and Import Networks*, on page 30). If the cache directory and file exist, function pickle.load() deserializes the data object. Otherwise, create the directory and file and call function pickle.dump() to serialize the data object domain and save it into the file. Note that you must open the file in the binary mode. The compressed cached pickle file is available as thegoodwife_cbs.pickle.zip.[4]

The rest of the community membership list on page 145 is a two-column table. The first column represents some subtle aspects of membership types (“P” for individual users, “C” for communities, “<” for “friends,” “>” for “friends-of”); the second column has usernames. To keep your code modular, write a function download(domain_name) that takes care of this and similar tables.

4. pragprog.com/titles/dzcnapy/source_code

lj.py

```
def download(base, domain_name):
    """
    Download interest data from the domain_name community on
    LiveJournal, convert interests to tags, create a domain DataFrame
    """
(1) members_url = "{}/fdata.bml?user={}&comm=1".format(base, domain_name)
    members = pd.read_table(members_url, sep=" ",
                            comment="#", names=("direction", "uid"))

(2) wnl = nltk.WordNetLemmatizer()
    stop = set(nltk.corpus.stopwords.words('english')) | set(('&'))
(3) term_vectors = []
(4) for user in members.uid.unique():
        print("Loading {}".format(user)) # Progress indicator
        user_url = "{}/interestdata.bml?user={}".format(base, user)

(5)     try:
            with urllib.request.urlopen(user_url) as source:
                raw_interests = [line.decode().lower().strip()
                                 for line in source.readlines()]
        except:
            print("Could not open {}".format(user_url)) # Error message
            continue

        if raw_interests[0] == '! invalid user, or no interests':
            continue

(6)     interests = [" ".join(wnl.lemmatize(w)
                              for w in nltk.wordpunct_tokenize(line)[2:]
                              if w not in stop)
                     for line in raw_interests
                     if line and line[0] != "#"]

(7)     interests = set(interest for interest in interests if interest)
(8)     term_vectors.append(pd.Series(index=interests, name=user).fillna(1))

    return pd.DataFrame().join(term_vectors, how="outer").fillna(0)\
(9)     .astype(int)
```

❶ Convert the interest table into a two-column DataFrame.

❷ Prepare a WordNet lemmatizer wnl—a tool for converting words into lemmas—and a list of stopwords stop, extended to include "&." You will need this list to eliminate too frequent words. Coerce the standard list of stopwords into a set for faster lookup, because the list lookups in Python are notoriously slow.

Your next step is to download all interest lists, convert interests into terms (they are not always the same!), and combine all term lists into a term matrix—a matrix whose columns are term lists. An interest list looks very similar to the membership list—even the call to caching is the same:

```
# Note: Polite data miners cache on their end. Impolite ones get banned.
# <intid> <intcount> <interest …>
18576742 1 +5 sexterity
624552 7 a beautiful mess
18576716 1 any more hot chicks?
44870 28 seriously?
1638195 94 shiny!
«…more interests…»
```

This is still a table (showing interest ID, the system-wide rank of the interest, and the actual interest), but the number of columns differs, depending on how many words are in the interest description. Pandas DataFrames are poor parsers for irregular texts; tackle the columns by hand, using low-level Python tools. Note that if the username is not found or the user has not declared any interests, the content is entirely different:

```
! invalid user, or no interests
```

So, let's continue the function code inspection:

❸ Set up an empty list accumulator term_vectors. At the end of the loop, it becomes a list of term vectors—raw material for the term matrix.

❹ Loop through all unique usernames in the community, because you need the URLs of their interest lists.

❺ Obtain an interest list raw_interests as a Python list of strings for each distinct community member in the try-except block. If the URL fails to open (for any reason beyond your control), the script does not crash but politely informs the programmer of the failure and proceeds to the next user. The same thing happens when the user has no interests or does not exist at all. If everything goes fine, decode and strip each string of trailing whitespaces. LJ interest lists are always in lowercase, but if they are not, a call to lower() ensures that in the rest of the script you compare apples to apples.

❻ Split each non-empty, non-commented interest into individual words with nltk.wordpunct_tokenize(). There may be more than one word in an interest description, and some or all of these words may be forms of other words (as in "chicks"—"chick"). Text analysis practitioners preach different ways of handling word forms. Some suggest to leave the forms alone and treat them as words on their own. Some advocate lemmatizing or stemming: reducing a form to its lemma (the standard representation of the word) or to the stem (the smallest meaningful part of the word to which affixes can be attached). A lemmatizer reduces "programmers" to one "programmer," and a stemmer, depending on its zeal, yields a "programm" or even a "program." Let's follow the lemmatizing crowd. Furthermore, almost

everyone agrees that certain words (such as “a,” “the,” and “and”) should never be counted at all. Remove them.

❼ As a result of lemmatization, stemming, and stopword elimination, a term list may end up having duplicates (for example, “the chicks” and “a chick” may both become “chick”). Convert each term list into a set. Surely, sets have no duplicates.

❽ Your goal is to produce a term vector model (TVM)—a table where rows are terms and columns are community members.[5] In the Pandas language, this table is known as DataFrame, and its columns are known as Series. Transform a term list to a Series, where the individual terms become the Series index, and all values are set to 1s, like this:

```
print(term_vectors)
```

```
shiny!           1
+5 sexterity     1
big damn hero    1
«…more terms…»
Name: twentyplanes, dtype: float64
```

❾ Finally, join all Series into a DataFrame. This operation involves the mystery of data alignment, whereby all participating Series are stretched vertically to have their row labels (terms!) aligned. Such stretching almost inevitably produces empty cells in the frame. Fill them up with zeros: an empty cell in row A and column B signals that the term A was not on the list B. The resulting variable DataFrame has 12,437 rows and adequately represents the cultural domain of LiveJournal users interested in the CBS show *The Good Wife*.

Build the Term Network

The next CDA step requires that you build a network of terms: a graph where nodes represent terms, and (weighted) edges represent their similarities.

You could have included all 12,437 discovered terms in the graph, but some of them are mentioned only once or twice (which is expected, given Zipf’s law[6]). Rather than wondering why the less frequently used words are in fact less frequently used, remove all rows with fewer than ten occurrences, but provide an option of changing the cut-off value MIN_SUPPORT in the future. At this point, you might wish that Python had first-class constants, but

5. en.wikipedia.org/wiki/Vector_space_model
6. mathworld.wolfram.com/ZipfsLaw.html

nonetheless spell MIN_SUPPORT in all capital letters. DataFrame limited is a truncated version of domain: it has only 319 rows.

Zipf's Law

Zipf's law states that given some corpus of terms drawn from a natural language, the frequency of any word is inversely proportional to its rank by frequency. In other words, if the frequency of the most frequent term is f_0, then the frequency of the next most frequent term is $f_0/2$, and so on, and the frequency of the nth term is $f_0/n\alpha$. The same law (with the slightly different exponent α) applies to population ranks of cities, corporation sizes, income rankings, and more. The continuous form of the discrete Zipf law is known as Pareto distribution, and Zipf's law is a special case of the power law (mentioned on page 108).

lj.py
```
MIN_SUPPORT = 10
sums = domain.sum(axis=1)
limited = domain[sums >= MIN_SUPPORT]
```

Since you want to build a network based on co-occurrences, you can consider two terms as similar if different community members frequently use them together. Calculate the matrix of co-occurrence by matrix-multiplying the limited DataFrame by itself.

The resulting square DataFrame cooc contains the total counts of all terms on the main diagonal (suppress them by multiplying the matrix by an inverted identity matrix) and the counts of co-occurrences elsewhere (they will eventually become weighted network edges).

lj.py
```
cooc = limited.dot(limited.T) * (1 - np.eye(limited.shape[0]))
```

Slice the Network

Now, you must make another painful decision: which matrix elements become edges and which get discarded? *Slice Weighted Networks*, on page 81, explains the slicing philosophy. Choose the slicing threshold, SLICING, to be six. Higher SLICING results in many small communities. Lower SLICING results in few large communities. Six seems to be a good compromise between count and size.

The resulting matrix is very sparse (every cell represent an edge, but we agreed to have as few edges as possible!). Stack and normalize it—essentially convert into a sparse matrix, where each row represents a significant edge and its weight. Since NetworkX prefers to deal with Python (rather than Pandas) data structures, convert the weights to a dictionary:

lj.py
```
SLICING = 6
weights = cooc[cooc >= SLICING]
weights = weights.stack()
weights = weights / weights.max()
cd_network = weights.to_dict()
cd_network = {key:float(value) for key,value in cd_network.items()}
```

You are just one step away from having an amazingly structured network. Let's create a new empty graph, populate it with the edges from the dictionary, and update the "weight" edge attributes:

lj.py
```
tag_network = nx.Graph()
tag_network.add_edges_from(cd_network)
nx.set_edge_attributes(tag_network, cd_network, "weight")
```

The constructed network with the added attributes is your first fascinating result; save it without hesitation into a GraphML file in a specially created directory results (*Share and Preserve Networks*, on page 30).

lj.py
```
if not os.path.isdir("results"):
    os.mkdir("results")

with open("results/" + DOMAIN_NAME + ".graphml", "wb") as ofile:
    nx.write_graphml(tag_network, ofile)
```

If you had no access to non-Anaconda modules, you would abandon Python at this point and switch to interactive software like Pajek,[7] UCINET[8] (which are outside the scope of this book), or Gephi (Chapter 4, *Introducing Gephi*, on page 33) for further network analysis. Fortunately, Python has the community module (*Outline Modularity-Based Communities*, on page 138) that will let you stay in the same program for the entire analysis cycle.

Extract and Name Term Communities

The modularity of the new network is quite poor (we suggested on page 138 that a network is definitely modular only when the modularity is 0.6 or above):

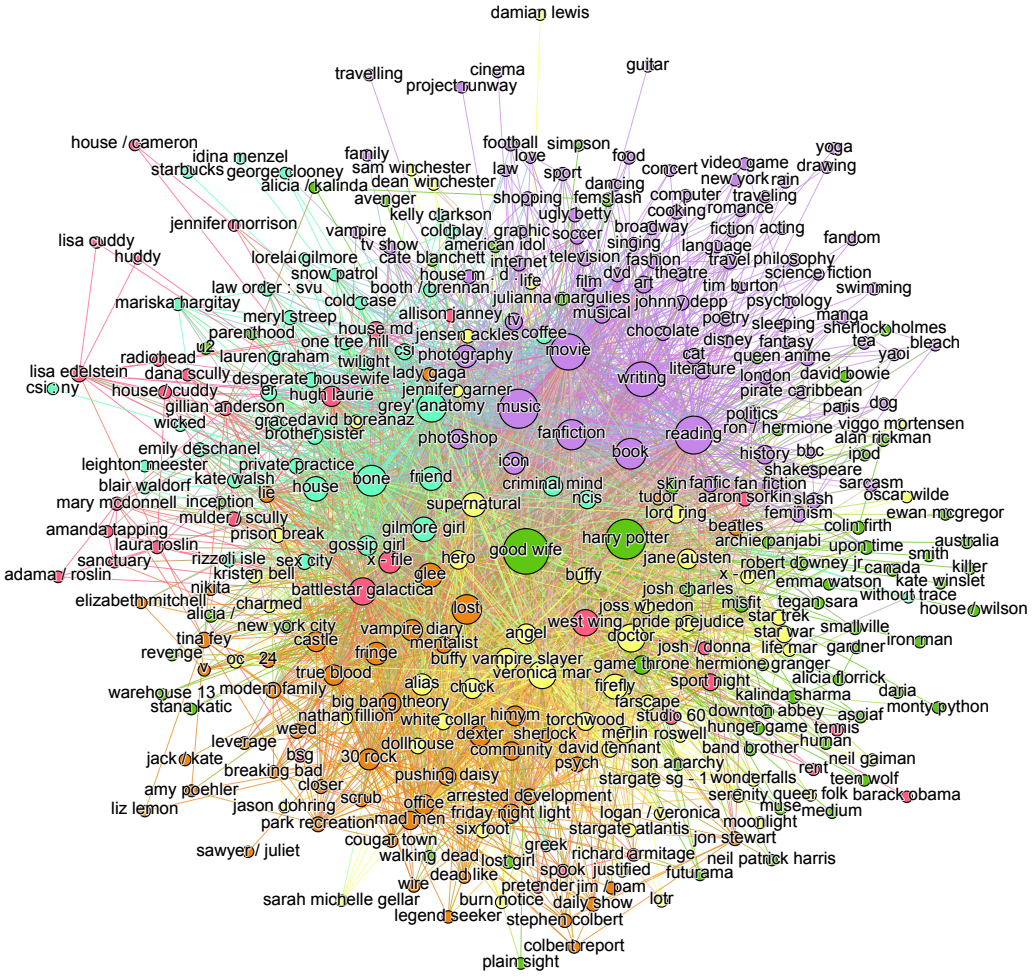
lj.py
```
partition = community.best_partition(tag_network)
print("Modularity: {}".format(community.modularity(partition,
                                  tag_network)))
nx.set_node_attributes(tag_network, partition, "part")
```

```
Modularity: 0.15815567681142356
```

7. vlado.fmf.uni-lj.si/pub/networks/pajek/
8. sites.google.com/site/ucinetsoftware/home

Apparently, counting raw co-occurrences is not the best way to describe similarities—indeed, correlation-based networks are much more flexible, and you will learn about them later (Chapter 14, *Similarity-Based Networks*, on page 165). However, even the coarse network that you have allows some meaningful interpretation.

The partition that you extracted precisely defines the term communities. (Add them as an attribute *part* to the network nodes.) The following figure shows the whole network of tags. The node diameter represents the number of times the corresponding tag was mentioned in the corpus, and the node colors match the term communities.



So, you extracted the communities, but they are nameless. You could have explored them with your bare eyes and come up with some proper names, but then you would not be a hardcore Python programmer after that, would you? Instead, add a cherry on the cake and write another seven lines of code

that locate the five most frequently used terms per cluster. Hopefully, they indeed describe the content.

You need a little helper function describe_cluster(terms_df). The function takes a DataFrame of terms in one community, extracts the namesake rows from the original domain, calculates their use frequency, and returns the top HOW_MANY performers.

lj.py
```
HOW_MANY = 5
def describe_cluster(terms_df):
    # terms_df is a DataFrame; select the matching rows from "domain"
    rows = domain.join(terms_df, how="inner")
    # Calculate row sums, sort them, get the last HOW_MANY
    top_N = rows.sum(axis=1).sort_values(ascending=False)[:HOW_MANY]
    # What labels do they have?
    return top_N.index.values
```

Finally, convert the partition into a DataFrame, group the rows by their partition ID, and beg the helper to come up with a name for each community.

lj.py
```
tag_clusters = pd.DataFrame({"part_id" : pd.Series(partition)})
results = tag_clusters.groupby("part_id").apply(describe_cluster)
for r in results:
  print("-- {}".format("; ".join(r.tolist())))
```

Surprisingly, it works!

```
-- good wife; harry potter; game throne; misfit; skin
-- music; reading; movie; writing; book
-- bone; grey ' anatomy; gilmore girl; house; friend
-- battlestar galactica; west wing; x - file; hugh laurie; house md
-- lost; glee; fringe; vampire diary; 30 rock
-- doctor; veronica mar; firefly; supernatural; buffy vampire slayer
```

Each line shows up to five most frequently mentioned terms per cluster (the terms are separated by semicolons). Some terms look strange and barely recognizable (“x - file”)—but remember all those rigorous transformations that they had to go through, such as lemmatizing! Some terms are duplicates (“house md”—“house”), but this simply means that the transformations were not rigorous enough.

Interpret the Results

There are two levels of interpretation of the CDA results.

At the lower level, you can conclude that all 319 terms selected for analysis are important for *The Good Wife* viewers—otherwise, the viewers would not

have selected them. Moreover, some of the terms go together more often than the others. You can see that “bone[s],” “grey ’[s] anatomy,” and “gilmore girl[s]” have something to do with each other (hint: Sean Gunn starred in all three series); “veronica mar[s]” and “firefly” are TV shows that were both prematurely canceled; “music,” ”reading,” and “writing” are popular activities… You can make these mechanistic conclusions without having the slightest idea about the cultural domain.

At the higher level, you might be an ethnographer, anthropologist, psychologist, or sociologist interested in the mindscape of *The Good Wife* fans. In particular, you might want to compare their mindscape to the mindscapes of, say, *Harry Potter* or *House, M.D.* fans. However, since you are a humble computer programmer or data scientist, you shall entrust the higher level interpretation to the SMEs.

This case study guided you from selecting an online blogging (or, rather, gossiping) community to constructing and partially interpreting a cultural domain—a semantic network of terms partitioned into six term collections. The method is fairly extensible and can be applied to other topical communities.

You have seen a network of words, but you have not seen it all. In the next chapter, we will show you a network of cosmetics!

In Constantinople there are some persons, particularly Armenians, who devote themselves to the preparation of cosmetics, and obtain large sums of money from those desirous of learning this art.

➤ *G. W. Septimus Piesse, British perfumer*

CHAPTER 13

Case Study: Going from Products to Projects

This chapter uses Matplotlib, community.

One of the goals of product network analysis is to identify nontrivial collections of co-purchased or co-recommended products. We can treat such collections as "customer projects" or "toolsets." You can find these networks of products frequently purchased together or recommended to be used together in marketing, advertising, and similar business disciplines.

As an example of product network analysis, let's have a look at cosmetics sold by Sephora®. In this case study, you will learn how to convert a CSV data file with cosmetics co-purchasing data into a complex network with the help of `csv`, `itertools`, and `collections` libraries. You will calculate attribute assortativity of the complex network and blockmodel it—construct its higher-level representation as an induced graph. Finally, you will use `graphviz_layout()` to produce a picture of the network without invoking any non-Python software.

Read Data

Most products on Sephora's website have *"Use With"* recommendations: one or more other products that the Sephora staff recommends customers purchase in conjunction with the original product.[1] For each product, the website contains plenty of characterizing information, such as brand, category, price, volume, and star rating. Each product is uniquely identified by the store SKU number and an alphanumeric ID. We will build a network of *"Use With"*

1. www.sephora.com

products created from previously acquired data and explore its structure concerning the product categories.

Crawling Sephora

If you want to download information about a specific product, you can program a crawling procedure using modules urllib.request for the actual download, and BeautifulSoup (bs4) for parsing the HTML response. (Both modules are outside of the scope of this book.) Sephora's website has a straightforward organization. Once you extract the IDs of the recommended products, you can repeat the download/parse cycle until your script finds no more new products. Since the *"Use With"* network is disconnected, you will need to run the crawling procedure more than once, starting from randomly selected products, to harvest all or at least the largest components.

For your convenience, we provide the raw data for the network construction in two CSV files. File use-with.csv has 3,943 rows: the first row is the header; the remaining rows contain network edges as edge ID (not needed in this case study), start product node, and end product node. We assume that the network is undirected (in reality it is not). File product.csv has 2,976 rows: the first row is the header; the remaining rows describe product nodes (one node per row) and contain product attributes as product ID, brand, star rating, and category. The latter two attributes may be empty.

As always, we start by importing all the necessary modules (the Aside titled "Where to Import?" on page 104 explains why):

products.py
```
import csv
from collections import Counter
from operator import itemgetter
from itertools import chain, groupby
import networkx as nx
from networkx.drawing.nx_agraph import graphviz_layout
import community
import matplotlib.pyplot as plt
import dzcnapy_plotlib as dzcnapy
```

Our next step is to read the edges and product attributes from the files, construct a vanilla network, and decorate its nodes with attributes (*Add Attributes*, on page 23):

products.py
```
with open("use-with.csv") as usewith_file:
    reader = csv.reader(usewith_file)
➤   next(reader)
    G = nx.from_edgelist((n1, n2) for _, n1, n2 in reader)
```

```
with open("products.csv") as product_file:
    reader = csv.reader(product_file)
➤   next(reader)

    brands = {}
    cats = {}
    star_ratings = {}

    for ppid, brand, star_rating, category in reader:
        brands[ppid] = brand
        cats[ppid] = category
➤       star_ratings[ppid] = float(star_rating if star_rating else 0)

# Set node attributes, based on product attributes
attributes = {"brand" : brands, "category" : cats, "star" : star_ratings}
for att_name, att_value in attributes.items():
    nx.set_node_attributes(G, att_value, att_name)
```

Note how we use next(reader) to skip the header rows in the first two highlighted lines, and how we impute zero star rating for the rows that do not have the star rating field in the last highlighted line.

Analyze the Networks

The resulting graph G has 2,975 nodes and 3,162 edges. It is very sparse:

```
print(nx.density(G))
```

```
❮ 0.0007147660678259198
```

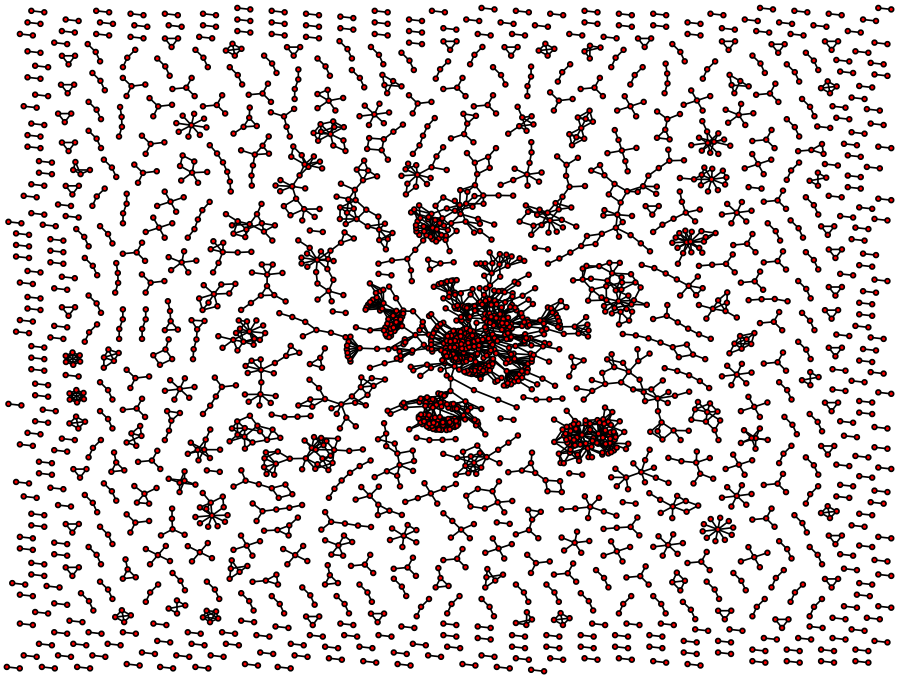
It also has a lot of small connected components with two to four nodes, as shown in the figure on page 158. (You can call nx.connected_components(G) and measure the size and count of them on your own.)

To keep the case simple, we consider only the largest component (the GCC). We sort all components of G by size, select the last one (the largest!), join the respective label lists into one with chain.from_iterable(), and extract the subgraph induced by these nodes. We store the resulting subgraph in the variable called gccs:

products.py

```
TOP_HOWMANY = 1
gccs_nodes = chain.from_iterable(sorted(nx.connected_components(G),
                                        key=len)[-TOP_HOWMANY:])
gccs = nx.subgraph(G, gccs_nodes)
```

The subgraph contains 25 percent of nodes and 36 percent of edges from the original graph, and it also has the most interesting structural elements. If you don't fancy these numbers, simply change TOP_HOWMANY.



So, what can we say about the distribution of the graph attributes? Do neighbors tend to be assortative or disassortative (*Estimate Network Uniformity Through Assortativity*, on page 99)? Let's find out:

products.py
```
for att_name in attributes:
    print("Assortativity by {}: {}"\
              .format(att_name,
                      nx.attribute_assortativity_coefficient(gccs, att_name)))
```

```
Assortativity by category: 0.03577904976206569
Assortativity by brand: 0.8687551723142831
Assortativity by star: -0.0058012311220827645
```

The news is mixed. On the one hand, connected products are very likely sold under the same brand—because cosmetic brands provide comprehensive toolkits! On the other hand, connected products belong to different categories—indeed, why would one buy two tools from the same category together? On the third hand (yes, computer scientists can have as many hands as it takes to describe the problem, as long as all hands, except for the first two, are virtual), connected products have unrelated star ratings. The last result is confusing, and you can leave the question open until you can afford to hire a subject-matter expert.

Because of the poor node assortativity by category, we expect a weird mixture of categories within any structural element—for example, within modularity-defined communities. Let's partition the network into communities and see how they are connected and named.

products.py
```
part = community.best_partition(gccs)
print("Modularity: {}".format(community.modularity(part, gccs)))
```

```
Modularity: 0.8241527716500038
```

The following statements create a list of lists of nodes in each community by collecting the nodes with the same partition ID. Function itertools.groupby() demands that the sequence is already sorted by the same key as would be used for grouping. In our case, the key is the partition ID, the second element of each tuple on the list returned by parts.items(), thus itemgetter(1). We will need the list later to auto-generate community labels.

products.py
```
groups = groupby(sorted(part.items(), key=itemgetter(1)), itemgetter(1))
community_labels = [list(map(itemgetter(0), group)) for _, group in groups]
subgraphs = [nx.subgraph(gccs, labels) for labels in community_labels]
```

We could use the previously constructed list of lists as a partition in nx.quotient_graph(), but instead, we will utilize community.induced_graph(partition, graph), the blockmodeling tool from the community library:

products.py
```
induced = community.induced_graph(part, gccs)
induced.remove_edges_from(induced.selfloop_edges())
```

The induced graph usually has many self-loops because of copious connections between the nodes in the original network that belong to the same community. We remove the loops (on the highlighted line) to avoid clutter in the future network printout.

Name the Components

The new induced graph nicely reflects the macroscopic structure of the original product network. It has only eighteen nodes and twenty-nine edges. The nodes are nameless so far, and we need to give them names. Having no better source of labels than the product categories, we select the most popular category within each induced node as the node label. We need an auxiliary function to obtain the name of the dominant category in a community. The Sephora website reports category names as colon-separated hierarchical paths. To save space in the future printout, we keep only the last path component:

products.py
```
def top_cat_label(community_subgraph):
    items = [atts["category"] for _, atts
             in community_subgraph.nodes(data=True)]
    top_category = Counter(items).most_common(1)[0]
    top_label_path = top_category[0]
    return top_label_path.split(":")[-1]
```

Function collections.Counter(sequence) is an indispensable tool for counting occurrences of unique items in a sequence. It returns a dictionary-style Counter object with the method Counter.most_common(n) that reports a list of the n most popular items as (label,count) tuples (we only need the item label).

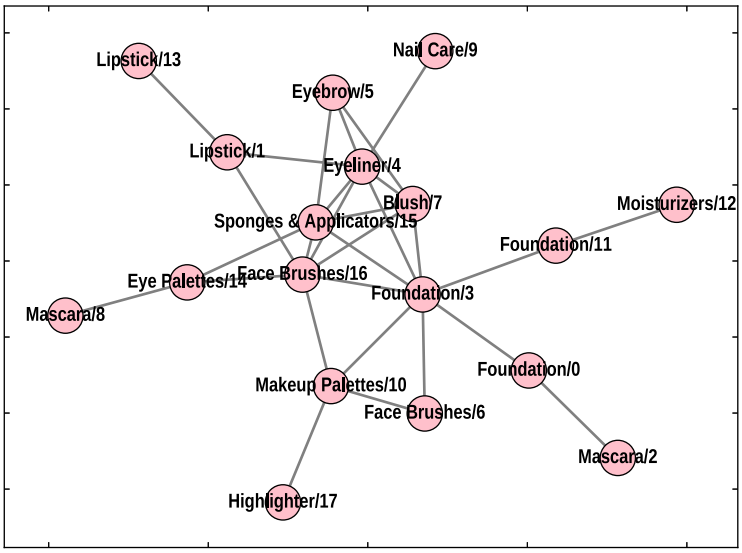
There may be several communities with the same dominant category in the network. If we blindly relabel them, their respective induced nodes will have the same label, and NetworkX will combine them into one node. To avoid unnecessary node merging, let's append the community ID to each label. The new labels look somewhat odd, but at least they are unique:

products.py
```
mapping = {comm_id: "{}/{}".format(top_cat_label(subgraph), comm_id)
           for comm_id, subgraph in enumerate(subgraphs)}
induced = nx.relabel_nodes(induced, mapping, copy=True)
```

At this point, our analysis is complete, but the data sponsor (the person or organization who ordered us the study) would rather see a nice picture than read a thousand barely decipherable labels. It's time to produce a picture. Function graphviz_layout() (*Harness Graphviz, on page 29*) attempts to find appropriate positions for the graph nodes, and nx.draw_networkx() draws the graph. The last function takes tons of parameters: you can customize edge and node colors, sizes, labels, and so on. You can save the resulting picture into a file, or display it on the screen, or both.

products.py
```
attrs = {"edge_color" : "gray", "font_size" : 12, "font_weight" : "bold",
         "node_size" : 700, "node_color" : "pink", "width" : 2,
         "font_family" : "Liberation Sans Narrow"}

# Calculate best node positions
pos = graphviz_layout(induced)

# Draw the network
nx.draw_networkx(induced, pos, **dzcnapy.attrs)

# Adjust the extents
dzcnapy.set_extent(pos, plt)

# Save and show
dzcnapy.plot("ProductNetwork")
```

The output of the script is in the following figure. Since graphviz_layout() uses random numbers to calculate the best network layout, you will probably see a different picture when you execute the same code.



To understand the figure better, let's explore the degrees of the induced nodes with nx.degree(induced). The results are shown in the following table.

Node(s)	Degree
Foundation/14, Face Brushes/3	8
Makeup & Travel Cases/8	7
Blush/9	6
Eyeliner/13	5
Makeup Palettes/11	4
Eye Palettes/10, Eyebrow/16	3
Contour/17, Face Brushes/0, Foundation/15, Foundation/4	2
Moisturizers/7, Highlighter/5, Lipstick/1, Mascara/2, Mascara/6, Nail Care/12	1

At the top of the table, you can see the cosmetics essentials that are required for makeup but are not visible—namely, foundations and tools (brushes, cases, palettes). The most eye-catching tools are at the bottom of the table: mascaras, lipsticks, nail care tools, and highlighters. We can hypothesize that if a node is connected to (recommended to be "used-with") fewer neighbors, it is more specialized. The specialized nodes are at the periphery of the product network and depend on the more general nodes in the core.

Just like some other case studies presented in the book, the "products to projects" case is not limited only to the Sephora products. Given sufficient co-purchasing data, you can build a network of products, identify dense product communities, name them, and argue about possible reasons for their existence.

In the Next Part

The complex networks you have seen so far had a reasonably crisp structure. For any two nodes, you could say, with a fair degree of confidence, whether there was an edge incident to them or not. That is not how things work in real life.

In real life, there is almost always a degree of uncertainty involved in binary relationships. Alice, Bob, and Chuck may be friends, but Alice and Bob may be better friends than Alice and Chuck. A husky and a reindeer may be less likely to be bought together than a husky and a penguin. The uncertainty is a fact, and we need to know how to deal with it. In the next part, you will learn how to connect nodes in a fuzzy way based on potential similarity.