13

Network Data

In this final chapter on advanced data types, we discuss another kind of data that is frequently used in the social sciences: networks. Until now, we focused on data collections that cover separate entities: people, countries, elections, or conflict events. Now, we extend this perspective to examine *relationships* between them, in addition to the entities themselves. We represent these relationships as network structures.

13.1 WHAT IS NETWORK DATA?

A network is a structure consisting of entities (or nodes) and the relations between them. In more formal language, networks are often referred to as "graphs," and the nodes as "vertices" with "edges" connecting them. For example, to represent an airline network, airports constitute the entities of the network, and the direct flight connections between these airports are the relations linking these entities. Graphs can also be used to represent social networks, where individuals are the nodes of the network, and edges exist between those individuals that know each other personally. Figure 13.1 (left panel) shows a simple network consisting of four nodes (A–D), and a total of four edges.

In the simple network above, any pair of nodes can either be connected with an edge or not. This is what we call an "undirected" graph, since the edges do not point one way or the other – they only connect a pair of nodes as in Figure 13.1 (left panel) above. Undirected graphs have many applications; for social network analysis, they can be used to connect individuals that have a symmetric relationship, for example, those that have coauthored a scientific publication (Newman, 2004) or have

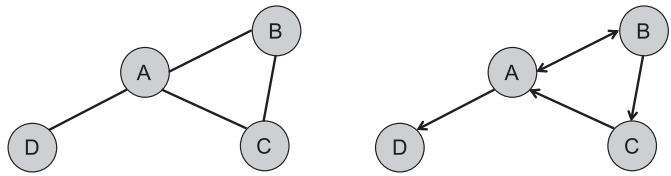

FIGURE 13.1. An undirected (left) and a directed graph (right).

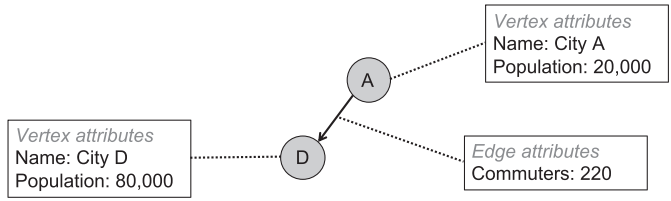

FIGURE 13.2. A graph with vertex and edge attributes.

co-starred together in a movie (Albert and Barabási, 2002). For many other applications, however, the links between the nodes in our network must be directional, and call this a "directed" graph. This simply means that we give each of our edges a direction. In Figure 13.1 (right panel), for example, you can see that there is an edge running from A to D, but not vice versa. Between A and B, however, we have edges in both directions (visualized as a two-directional arrow). Directed networks can be used to represent flows of some kind, for example, trade flows between countries (Barbieri and Keshk, 2017), or foreign direct investment of one country in another (Lee and Mitchell, 2012).

In many cases, simply having a network with vertices and edges is not enough, and we need to store more information about both. For this, we can amend the simple graph model, such that additional information is attached to the vertices and edges in our graph. Figure 13.2 illustrates this. We have a simple network with two nodes, A and D, each of which corresponds to a city. A directed edge from A to D represents the commuters from A who go to work in D every day. This network contains additional data, so-called "attributes," about cities and commuting links (shown as boxes in the figure). We have information about the name and the population of each city in the vertex attributes, and the number of commuters in the edge attributes.

So far, a graph with edges and attributes was a conceptual data model – an idea of what our data looks like. How do we work with this data model in practice? In other words, how do we physically store network

data in files? There are two ways for doing this: as an adjacency matrix or an adjacency list. In both cases, we map our network to something that looks like a table. Once we have done that, we can use these tables exactly as we did in the previous chapters of this book: store them in files, or put them into a database for tabular data. But let us first look at how adjacency matrices and lists work.

The idea of an adjacency matrix is very simple: We create a quadratic table such that there is one row and one column for each vertex that exists in the graph. The entries in this table are then used to store information about the edges in the network, such that the rows correspond to the nodes where edges start, and the columns to those where they end. Let us illustrate this for the directed network from Figure 13.1 (right). Our adjacency matrix has four rows and four columns. To indicate that there is an edge running from A to D, we put a value of 1 in the first row, fourth column (and correspondingly for the other edges in our graph):

	A	B	C	D
A	0	1	0	1
B	1	0	1	0
C	1	0	0	0
D	0	0	0	0

While the adjacency matrix format is easy to understand, there are two major downsides to it. First, remember the advice I gave in Chapter 3: Tables should grow down, not sideways. The adjacency matrix violates this rule, since adding vertices to a graph means adding columns to the matrix. Second, adding attributes is very difficult when working with adjacency matrices. As soon as we want to store different edge attributes, this becomes impossible with a single matrix, since it holds exactly one value for each connection. Therefore, adjacency lists are preferable for most applications.

The adjacency list format follows a different approach. Rather than mapping out the entire set of possible pairs of edges and then indicating which ones are connected, an adjacency list simply contains only those pairs of edges that are connected in the graph. For the directed network in Figure 13.1, the adjacency list looks as follows:

From	To
A	B
A	D
B	A
B	C
C	A

The graph has five directed edges, each of which corresponds to a single line in our table. This format is very flexible: If we want to add more vertices or edges, we can do so by inserting more rows into the table. Also, if we want to add edge attributes to our graph, we can do so by storing them in separate columns, in addition to the `from` and `to` columns we have in our table. While edge attributes can easily be accommodated in this way, vertex attributes are typically stored in a separate vertex list, which is what we will do below.

We are now equipped with sufficient knowledge about the concept of networks and how we can store them in tables. Let us now take a quick look at the applied example we work on in this chapter.

13.2 APPLICATION: TRADE AND DEMOCRACY

International trade patterns constitute a central question in international political economy. When studying if and how much countries trade with each other, our main interest is not in single countries and their characteristics, but rather in the interactions between them. Therefore, the data that we need to study this is best represented as a network, where states constitute the nodes and the trade links between them are the edges. In our example, we study an important question: How does the level of democracy of states determine the volume of trade between them (Bliss and Russett, 1998)? The data for our analysis comes from two different sources. The first one is a dataset on bilateral trade, initially presented by Barbieri et al. (2009) and later updated until 2014 by Barbieri and Keshk (2017). The dataset draws on different sources and records (among other variables) the annual trade volume between pairs of states for the period 1870–2014. Therefore, rather than a single, static trade network, the dataset captures the temporal evolution of international trade, with annual observations.

The file `trade.csv` in the data repository contains a simplified version of the trade data. The format corresponds to the adjacency list we discussed above, so each trade link between two states constitutes an observation. States are coded according to the Correlates of War (COW) coding system (Correlates of War Project, 2008), which assigns independent states a

unique identifier, the COW code. For each link in our trade network, the states it connects are stored in the ccode1 and ccode2 variables. Due to the fact that we have annual observations, each line in the data also has a year variable. The last three variables provide information about the trade volume between the two states in the given year: smoothflow1 is the total volume of the first country's imports from the second country (in millions of US dollars), and smoothflow2 is the trade flow in the opposite direction. Both values are smoothed over time (see Barbieri and Keshk, 2017). Finally, smoothtotrade is the smoothed total volume of trade in the given year between the two states, independent of the direction.

Our second dataset for this chapter provides us with information about the states themselves, which we later use to explain the volume of trade between them. We rely on a subset of the large *Varieties of Democracy* (V-Dem) database, a project at the University of Gothenburg (Coppedge et al., 2019). Most importantly for our purpose, V-Dem provides aggregated expert assessments of many aspects of a country's political system, which we can use to examine how the level of democracy affects trade between two states. You can find a simplified version of the V-Dem data in the repository. It contains annual observations of states, each of which is coded with a cowcode and a year. Note that there are many missing values for the COW code, since V-Dem also tracks political units that are not considered to be independent states by COW and therefore have no COW identifier. The variable that we will be using to measure a state's level of democracy is v2x_polyarchy, which codes "electoral democracy" on a range from 0 to 1 (for more details about this and the other variables in the dataset, see the V-Dem codebook). In the file, we also have the world region the country belongs to (e_regiongeo) as well as the GDP per capita (e_migdppc).

Together, the trade and V-Dem datasets constitute the (longitudinal) network we will be analyzing in this chapter. The trade dataset is an adjacency list with additional edge attributes, while the V-Dem data is a vertex list with vertex attributes. Vertices are identified by COW codes, so that we can link both datasets easily.

13.3 Exploring Network Data in R with igraph

When dealing with networks, we need a toolkit that is able to handle graph structures and allows us to conduct network analysis with them. R's base functions do not allow us to do that, as they are designed to mainly work with tabular data. However, several of R's extension libraries provide functionality to process network data. One of the most powerful

ones is the igraph package that we use in this chapter. igraph can read and export different data formats for network data, and allows us to manipulate and analyze networks in R. As always, we first need to load the package:

```
library(igraph)
```

There are different ways in which you can load a network into igraph. We use a function where we provide an edge list (the trade data) and a vertex list, both as simple R data frames. Before we can do this, we first need to import both datasets into R. Let us start with the trade network. Here, we need to keep in mind that the trade dataset stores missing values as −9, which is why we need to explicitly define this during the import. Also, we remove missing values and entries with a bilateral trade volume of 0, since they indicate that no trade is taking place and the states are therefore not connected:

```
trade <- read.csv(file.path("ch13", "trade.csv.gz"), na.strings = "-9")
trade <- subset(trade, !is.na(smoothtotrade) & smoothtotrade > 0)
```

If we take a look at the summary of the trade dataset, you will notice several things. Not surprisingly, even with the missing links removed, the dataset is large and contains around half a million observations. This is due to the fact that we observe pairs of states with annual estimates. To get started, we only use a subset of the trade data to simplify our exercise. We restrict coverage to 2014, and only keep those links with a total trade volume of 100 million dollars or more:

```
trade <- subset(trade, year == 2014 & smoothtotrade >= 100)
```

Next, we turn to the vertex list, the V-Dem data. We load it as a data frame, restrict it to 2014, and remove observations with missing COW codes since we do not need them in this exercise:

```
vdem <- read.csv(file.path("ch13", "vdem.csv"))
vdem <- subset(vdem, year == 2014 & !is.na(cowcode))
```

Before we can use the V-Dem data in igraph, we need to arrange the columns in the data frame. For the edge list, igraph expects the first two columns to contain the node identifiers – this is what we already have in the trade data frame. For the vertex data, igraph requires the first column to be the node identifier, which is why we need to make the COW code the first column in vdem:

```
vdem <- subset(vdem,
   select = c("cowcode", "country_name", "year",
              "v2x_polyarchy", "e_regiongeo"))
```

Now, both data frames should be in the right format so that we can create a network from them in igraph. Let us see how this works. We use the graph_from_data_frame() function, which is one among many different functions in igraph to construct a network. It takes an edge list as its main argument, and (optionally) a vertex data frame with additional data on the vertices. We also define the network to be undirected by setting directed to FALSE:

```
tradenetwork <- graph_from_data_frame(trade, directed = F, vertices = vdem)
```

This does not seem to work: igraph is complaining about some some vertices in the trade data not being listed in vdem. The reason is that we do not have V-Dem codings for some states in the trade data – many of them are micro-states and are not covered by V-Dem. Therefore, we restrict our trade network to those pairs of states where V-Dem data is available for both of them:

```
trade <- subset(trade, ccode1 %in% vdem$cowcode & ccode2 %in% vdem$cowcode)
```

Now, let us try to construct the network again:

```
n <- graph_from_data_frame(trade, directed = F, vertices = vdem)
```

We now have an igraph network n, and can apply network-specific functions to it. First, we take a look at the summary:

```
summary(n)

IGRAPH 5ad0bc2 UN-- 174 3456 --
+ attr: name (v/c), country_name (v/c), year (v/n), v2x_polyarchy
| (v/n), e_regiongeo (v/n), year (e/n), smoothflow1 (e/n), smoothflow2
| (e/n), smoothtotrade (e/n)
```

Our network is undirected (UN) and has 174 vertices and 3,456 edges – if you want to check, you can compute these numbers with vcount(n) and ecount(n). The summary also displays the attributes we have defined for our network. For example, country_name is a vertex attribute (v) of type character (c), while smoothflow1 is an edge attribute (e) of type numeric (n).

While igraph defines its own methods for accessing and modifying a network, many of them work in ways that are similar to R. For example,

`V(n)` gives you access to the entire list of nodes in the graph, and you can retrieve any one of them simply by indexing. The following statement returns the first vertex in the graph:

```
V(n)[1]
```

```
+ 1/174 vertex, named, from 5ad0bc2:
[1] 700
```

Similar to a data frame, we can use the `$` operator to access a single attribute:

```
V(n)[1]$country_name
```

```
[1] "Afghanistan"
```

We can also use the bracket operator to filter a subset of nodes, for example, those with democracy scores higher than 0.9:

```
V(n)[v2x_polyarchy > 0.9]
```

```
+ 7/174 vertices, named, from 5ad0bc2:
[1] 225 94  390 220 385 380 2
```

For the edges, the `E(n)` function works in the same way. For example, it allows us to find out which edge has the maximum total amount of trade in 2014 with:

```
E(n)[which.max(E(n)$smoothtotrade)]
```

```
+ 1/3456 edge from 5ad0bc2 (vertex names):
[1] 710--2
```

Not surprisingly, this is the edge between China (COW code 700) and the US (COW code 2). So far, we have only used `igraph` to retrieve information that we could have also extracted from the original tables. The real added value of the library, however, is its ability to perform network-specific calculations. One of these is the "centrality" of nodes in the network, which is a key concept in network analysis. More central nodes are those that are better connected to others in the network. Centrality can be computed in different ways. "Degree centrality" is one of the simplest centrality measures, and it is defined as the number of links (the "degree") a node has to others. In `igraph`, we can calculate degree centrality with the `degree()` function, as in:

```
degree(n)[1:3]
```

```
700 540 339
 13  29  13
```

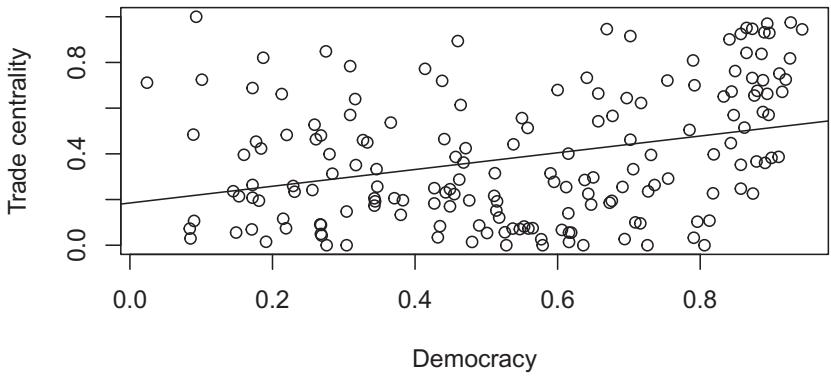

FIGURE 13.3. Level of democracy and centrality in the trade network.

Degree centrality, however, only considers the number of links that a country has, but not the other country it is connected to. Another centrality measure is eigenvector centrality, which gives higher centrality scores to those countries that are connected to other highly central countries. In other words, it measures centrality by identifying those countries that are connected to several other influential players in the trade network. Using the `vector` field, we can extract the centrality scores after running the corresponding function from `igraph`:

```
ec <- eigen_centrality(n)$vector
```

We can use these centrality scores to carry out a first analysis of whether democracy is related to a country's position in the trade network. To do so, we create a bivariate plot of the democracy scores for the vertices in our network, and the centrality measures we have just computed (see Figure 13.3).

While the plot does not show a clear pattern, the linear fit is positive. Still, this relationship could be confounded, so we will conduct further analyses below. To conclude the discussion of `igraph`, let us examine the trade network graphically. The entire network is large and densely connected, which is why a plot of the entire network would not be useful. Therefore, we extract a subset of the network (a "subgraph") containing only those countries located in South America according to V-Dem's region coding (region 18):

```
sa <- induced_subgraph(n, V(n)[!is.na(e_regiongeo) & e_regiongeo == 18])
```

This results in a much smaller graph with only 12 nodes and 43 edges. Before we plot it, we define two properties of the network that are later

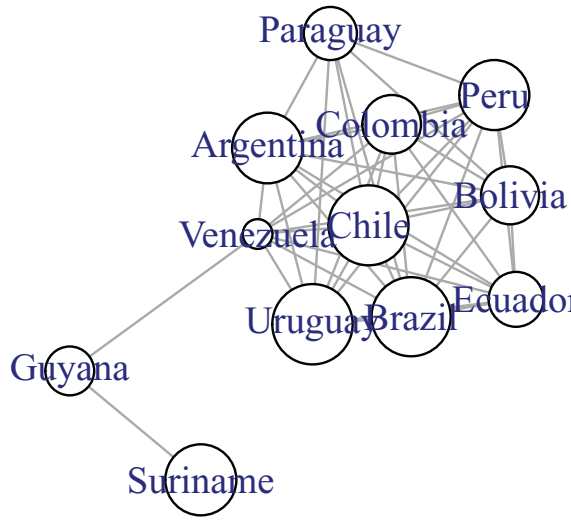

FIGURE 13.4. Trade network for South America.

used in the plot. First, we set a label for the nodes, which is simply the country name. Second, we define a weight for the edges, such that it is possible to distinguish trade relations with high volume from those with a low volume:

```
vertex_attr(sa, "label") <- V(sa)$country_name
edge_attr(sa, "weight") <- E(sa)$smoothtotrade
```

You can apply the generic `plot()` function to an `igraph` network and define a few network-specific parameters. In particular, we use a predefined layout function for the graph, which causes those nodes connected with edges of high weight (that is, with a high volume of trade between them) to be located close to each other. We also define the size of the nodes to be proportional to their democracy score, so that we can examine visually whether more democratic nodes are more central in the network:

```
plot(sa,
   layout = layout_with_gem,
   vertex.size = 40*V(sa)$v2x_polyarchy,
   vertex.color = "white")
```

The plot in Figure 13.4 shows that some countries such as Chile are more central actors when it comes to trade in South America. At first

glance, there seems to be a weak relationship with the level of democracy, such that less democratic countries (Venezuela, Ecuador, Bolivia) are located more at the periphery of the network. Still, this relationship remains to be explored more systematically, which is what we do in the next section.

13.4 Network Data in a Relational Database

In the above example, we used a file-based workflow for processing and analyzing network data in R and the `igraph` package. For larger and more complex networks, it is often useful to store them in a database that can handle large amounts of data and make them available to different users and in different formats. The trade network that we study in this chapter is an example for a more complex network dataset, due to its longitudinal structure with annual observations. In our example above, we simply avoided this difficulty by using a snapshot of the network for the year 2014. Now that we are moving towards a database-backed setup, we want to be able to deal with the entire dataset, without taking shortcuts.

In the previous section, we have seen that network data can be stored as tabular data, and more precisely, as a combination of two tables: an adjacency (edge) list, and a vertex list. This makes it easy to transfer this setup to a relational database, similar to what we did in the previous chapters. Assuming that your PostgresSQL server is running and that you created the `networkdata` database for this chapter, we connect to the PostgreSQL server as before:

```
library(RPostgres)
db <- dbConnect(Postgres(),
   dbname = "networkdata",
   user = "postgres",
   password = "pgpasswd")
```

First, we import the trade network data, using the `dbWriteTable()` function in R. Note that we need to be careful with the coding of missing values (−9) in the data, to make sure they are correctly recognized as `NA` values during the import.

```
trade <- read.csv(file.path("ch13", "trade.csv.gz"), na.strings = "-9")
dbWriteTable(db, "trade", trade)
```

We repeat the same procedure for the V-Dem dataset, which contains additional attributes for the nodes in our dataset (the countries), again with annual values:

```
vdem <- read.csv(file.path("ch13", "vdem.csv"))
dbWriteTable(db, "vdem", vdem)
```

Is the trade network an undirected or a directed graph? In our exercises with `igraph` above, we treated it as an undirected network with symmetric links between countries, each link representing a trade relation with a given total volume. In reality, however, the dataset contains directed information: The `smoothflow1` and `smoothflow2` variables for two countries A and B tell us the amount of imports into A from B, and vice versa. The trade dataset lists each of these connections only once; for example, the edge between the US (COW code 2) and Canada (COW code 20) is present only once for each year, as an edge $2 \rightarrow 20$, while $20 \rightarrow 2$ is not listed as a separate entry in the data.

This data structure may be useful if we want to minimize the size of a data file, but it is not convenient when working with the data. For that reason, it is useful to turn the trade dataset into a proper directed network, where each edge has a start and an end point and only one attribute, trade volume. How can we do this? If we assume that `smoothflow1` is our main edge attribute, we could keep the existing entries, but would have to add all of them again, but with reversed direction and `smoothflow2` as the edge attribute. This is exactly what the following line does:

```
dbExecute(db,
   "INSERT INTO trade (ccode1, ccode2, year, smoothflow1)
   SELECT ccode2, ccode1, year, smoothflow2 FROM trade")
```

Let us go through the different parts of this statement. Overall, it is an `INSERT` statement, so it takes some data and adds it to the `trade` table. Importantly, it specifies four columns of the table that the new data should be inserted in: `ccode`, `ccode2`, `year` and `smoothflow1`. These are the variables we would like to retain for our directed network with annual observations. But what is the data we want to insert? It is simply the entire `trade` table, but with reversed column order: `ccode2` should be inserted as `ccode1`, `ccode1` as `ccode2`, and `smoothflow2` as `smoothflow1`, while `year` remains the same. This is what we do in the second part of the `INSERT` statement, where we define the data to be inserted with a simple `SELECT` statement *on the same table the data should be inserted in*, but with the columns reordered such that they match the ones in the existing table.

Before we can proceed, however, let us clean up the table by removing the unnecessary columns `smoothflow2` and `smoothtotrade`:

```
dbExecute(db,
   "ALTER TABLE trade
   DROP COLUMN smoothflow2, DROP COLUMN smoothtotrade")
```

Also, the trade data contains entries with missing values in the trade volume column, or where the trade volume is zero. The latter is equivalent to there being *no* trade link between the two countries, so we remove these entries:

```
dbExecute(db,
   "DELETE FROM trade WHERE smoothflow1 IS NULL OR smoothflow1 = 0")
```

Finally, we strongly recommend that you create indexes on those columns that are frequently used to join tables, or to retrieve and aggregate data:

```
dbExecute(db, "CREATE INDEX ON vdem (cowcode)")
dbExecute(db, "CREATE INDEX ON vdem (year)")
dbExecute(db, "CREATE INDEX ON trade (ccode1)")
dbExecute(db, "CREATE INDEX ON trade (ccode2)")
dbExecute(db, "CREATE INDEX ON trade (year)")
```

Now, let us compute some simple network statistics directly in the database. Keep in mind that a relational database such as PostgreSQL has no notion of a network, so we cannot simply use existing functions to derive network statistics such as the length of shortest paths or centrality measures. However, we can use aggregate functions to compute the degree centrality of the different nodes. Recall that degree centrality is the number of incoming or outgoing links in a network. The following statement calculates this measure for the year 2014 and for all trade links with a volume of at least 100 million dollars, in decreasing order:

```
deg <- dbGetQuery(db,
   "SELECT ccode1, count(*) AS indegree
   FROM trade
   WHERE year = 2014 AND smoothflow1 >= 100
   GROUP BY ccode1
   ORDER BY indegree DESC")
deg[1:3,]
```

```
  ccode1 indegree
1    710      128
2      2      115
3    750      102
```

The statement selects trade links for 2014, and for each country (ccode1) counts the number of links that exist. Recall that the links in our trade network denote imports, which means that the three countries above are those that have the largest number of trade partners they import from. China, for example, imported from 128 other countries, with a volume of at least 100 million dollars from each. In a similar way, we can compute the centrality in terms of total import volume, simply by replacing the `count()` function with `sum()`:

```
deg <- dbGetQuery(db,
   "SELECT ccode1, sum(smoothflow1) AS totalimports
   FROM trade
   WHERE year = 2014
   GROUP BY ccode1
   ORDER BY totalimports DESC")
deg[1:3,]
```

```
  ccode1 totalimports
1      2      2344005
2    710      2075854
3    255      1201135
```

Note that unlike working with `igraph`, we do not hard-wire the network such that it consists, for example, only of trade links with a volume of 100 million dollars. Rather, we can dynamically extract different parts of the network, depending on what we need. This is a convenient way to deal with more complex network data, where the complete dataset resides in a relational database, and snapshots are dynamically extracted to be analyzed in `igraph` or another network analysis software. Our database also allows us to deal with the time series nature of our data. For example, the code below computes the degree centrality of the US over time (again restricted to those links with at least 100 million dollars):

```
deg <- dbGetQuery(db,
   "SELECT year, count(*) AS indegree
   FROM trade
   WHERE ccode1 = 2 AND smoothflow1 >= 100
   GROUP BY year
   ORDER BY year DESC")
deg[1:7,]
```

```
  year indegree
1 2014      115
2 2013      116
3 2012      117
4 2011      117
5 2010      118
6 2009      116
7 2008      122
```

As a final step in our analysis, we test the link between democracy and trade more systematically. To this end, we extract our network data in a way that makes it possible for regression analysis to be applied. Here, we use a simple *dyadic* setup, where we treat each trade link as a single observation. In this analysis, we model the trade volume from one state to another as a function of the economic performance of the two states as well as their level of democracy. This is one simple way to analyze network data; one problem is that it ignores all of the network structure beyond the individual dyads. For example, in this dyadic analysis we treat the imports from State A to State B as independent of other trade flows (e.g., from State C to State B). More complex network models can accommodate these higher-order dependencies, although this makes the estimation much more difficult (Ward et al., 2013).

For the purpose of our simple dyadic model, we have to export the data as a single data frame such that R can fit a regression model. This data frame contains data about edges as well as vertices – essentially, it is a combination of the vertex and edge lists we used in this chapter. In the following statement, we merge the `trade` and `vdem` tables in our database into a single data frame by means of a `SELECT` statement. The join operation we use here is based on the entries in the `trade` table, and appends V-Dem variables both for the first and the second country in each dyad. Note that we are joining the V-Dem table *twice* to the trade links: for the first country `ccode1` in the dyad, and then again for the second country `ccode2`. This is why we have to use the two alias names for the V-Dem table: `vdem1` and `vdem2`. Also note that in order to make the estimation of the regression model faster, we restrict the data again to one year with `trade.year = 2014`. However, you can remove this part of the statement to obtain the data for the entire time period:

```
tradedyads <- dbGetQuery(db,
   "SELECT
      ccode1, ccode2, trade.year, smoothflow1,
      vdem1.v2x_polyarchy AS polyarchy1,
      vdem1.e_migdppc AS gdppc1,
      vdem2.v2x_polyarchy AS polyarchy2,
      vdem2.e_migdppc AS gdppc2
   FROM
      trade,
      vdem vdem1,
      vdem vdem2
   WHERE
      ccode1 = vdem1.cowcode AND
      trade.year = vdem1.year AND
      ccode2 = vdem2.cowcode AND
      trade.year = vdem2.year AND
      trade.year = 2014")
```

This design omits those pairs of countries that do not trade with each other. Does democracy affect not just the volume of trade between pairs of trading countries, but also whether a trade link exists between them? To test this, we need to construct our dataset such that it contains all possible dyads, in other words, all pairs of countries *regardless of whether they trade or not*. For all these possible dyads, we add data from V-Dem about economic performance and level of democracy, and then use these data in a regression model to explain whether they had a trade link or not.

Before you take a closer look at the following statement, let us first think about how we generate such a data structure. Our strategy consists of two steps: First, we create a list of all possible dyads, irrespective of whether trade occurs between them. Second, we add the data on trade links, such that we can identify those cases in our complete list of dyads that actually have a trade link. Process for the first step: Our `vdem` table (the node list) contains all the countries in our sample, observed once per year. To create a list with all possible dyads, we simply join the table with itself. As you can see in the next statement, the `vdem` table is used twice, once as `vdem1`, and once as `vdem2`. Since we have time series data with annual observations, we need to make sure, however, that we only join observations from the same year (`vdem1.year = vdem2.year`) and also exclude links from one country to itself (`vdem1.cowcode != vdem2.cowcode`), since we cannot have dyads linking a country to itself. This is what the complete statement looks like:

```
alldyads <- dbGetQuery(db,
   "SELECT
      vdem1.cowcode AS ccode1,
      vdem2.cowcode AS ccode2,
      vdem1.year,
      vdem1.v2x_polyarchy AS polyarchy1,
      vdem1.e_migdppc AS gdppc1,
      vdem2.v2x_polyarchy AS polyarchy2,
      vdem2.e_migdppc AS gdppc2
   FROM
      vdem vdem1,
      vdem vdem2
   WHERE
      vdem1.year = vdem2.year AND
      vdem1.cowcode != vdem2.cowcode AND
      vdem1.year = 2014")
```

For performance reasons, we again restrict this example to observations from the year 2014. As you can see, this gives us a large dataset with 30,102 observations, much more than those in our original trade dataset.

This is not surprising, since the latter contains only pairs of countries where some trade has been registered, whereas our dataset lists all possible pairs of countries.

We can now continue to the second step: Join the information in the `trade` table to the complete list of dyads. The following statement takes our code above to generate a *temporary, virtual* table as part of a `SELECT` statement. This is done using the `WITH` keyword in SQL. Our virtual table is called `dyads` and can be used in the main statement as if it were a real table:

```
alldyads <- dbGetQuery(db,
   "WITH dyads AS
      (SELECT
         vdem1.cowcode AS ccode1,
         vdem2.cowcode AS ccode2,
         vdem1.year,
         vdem1.v2x_polyarchy AS polyarchy1,
         vdem1.e_migdppc AS gdppc1,
         vdem2.v2x_polyarchy AS polyarchy2,
         vdem2.e_migdppc AS gdppc2
      FROM vdem vdem1, vdem vdem2
      WHERE
         vdem1.year = vdem2.year AND
         vdem1.cowcode != vdem2.cowcode)
   SELECT
      dyads.ccode1,
      dyads.ccode2,
      dyads.year,
      polyarchy1,
      gdppc1,
      polyarchy2,
      gdppc2,
      smoothflow1
   FROM dyads LEFT JOIN trade ON
      dyads.ccode1 = trade.ccode1 AND
      dyads.ccode2 = trade.ccode2 AND
      dyads.year = trade.year
   WHERE dyads.year = 2014")
dbDisconnect(db)
```

The most important part of the statement is the `LEFT JOIN` of `dyads` to `trade` – as you may recall, a left join preserves all data from the first table, and joins those entries from the second table where the join condition (on `ccode1`, `ccode2`, and `year`) is met. Fields from the second table (such as `smoothflow1`) will be filled with `NULL` values for those rows from the first table without a match in the second table. This example again restricts the data to the year 2014 – you can get the entire dataset by removing the `LIMIT` clause of the statement.

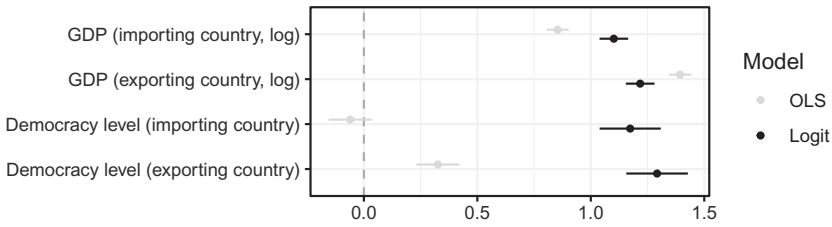

FIGURE 13.5. Coefficient plots for the regression models on trade and democracy.

The dataset we generate here can be used in an analysis where the dependent variable is the existence (0/1) of a trade link between a pair of countries. We can dynamically generate this binary dependent variable by testing whether the volume of imports between two countries is `NA` (which corresponds to the `NULL` values generated by the SQL left join, and indicates that the trade dataset does not contain a link between them).

13.5 RESULTS: TRADE AND DEMOCRACY

We have now extracted two datasets from our relational database. The first one (`tradedyads`) contains only those pairs of countries that trade with each other, and it allows us to study how the level of democracy affects the volume of trade between them. The `alldyads` dataset is a list of all possible dyads, and we will use it to analyze the impact of democracy on whether two countries trade at all. For the first analysis with volume of trade as the dependent variable, we simply fit a linear regression model, using the log10-transformed `smoothflow1` variable as the dependent variable. For the second analysis, we use a logit model with a binary dependent variable, which takes the value 1 if `smoothflow1` is not NA, and 0 otherwise. Each model includes the democracy levels of the importing country (`polyarchy1`) and the exporting country (`polyarchy2`), as well as their GDP per capita values (log-transformed).

Rather than showing the regression tables, I present the coefficients from the models graphically in Figure 13.5. Not surprisingly, richer countries import and export more, as the positive coefficients for GDP variables show. Beyond that, democracy affects trade: The more democratic both countries are in a given dyad, the more likely it is that they trade with each other (see the coefficients for the logit model). The effect of democracy on the volume of trade is not as clear-cut, as the results from the OLS model show: While more democratic countries export more, there is no evidence that democracy affects the amount of imports into a country.

13.6 SUMMARY AND OUTLOOK

Much research in the social sciences is about relationships between different kinds of entities, and network data are designed to capture this. Networks consist of nodes and the links between them, and are usually stored in adjacency matrices or adjacency lists. We saw that the latter format is much more versatile, and corresponds to well-designed tabular data. More complex network data, where nodes and edges have additional attributes attached to them, can be stored with separate tables for the nodes and edges, which are linked by unique node identifiers.

In this chapter, we used the `igraph` package for R, which is designed to process and analyze network data, as well as create flexible visualizations. Due to its special focus, it is able to generate network-specific measures, such as different types of centrality for the network nodes. We also discussed how network data can be processed in a relational database. A strength of this approach is its ability to process large network datasets using the built-in performance improvements such as indexes. Our examples above demonstrate how you can generate different types of network datasets for your analysis, while keeping the data in a relational database. PostgreSQL does not have graph-specific functions for network data, but it can be extremely helpful in managing your network data and shaping it in different ways. For your work with network data, here is a set of recommendations:

- *Always prefer the adjacency list format:* As the above examples showed, it is much easier to work with network data that come in the form of adjacency *lists* rather than matrices. Adjacency lists conform to the "long" table format, which has a number of advantages. Most importantly, simple networks queries such as the number of neighbors per node become simple data aggregation operations, which can easily be done in R or PostgreSQL.
- *Use tabular data formats for network data:* Unlike for spatial data, there are few established data formats specifically for network data. In most cases, other, more generic ones are used to store information about graphs, for example, XML, JSON, or the CSV format. These formats are usually a good choice, even though they do not incorporate network-specific features. For example, as in our example above, with a tabular data format we need to distribute the dataset in different files.
- *Keep the different datasets (nodes and edges) consistent:* Since network data is often spread out across different files (nodes and edges), there can be potential inconsistencies between them. For example, a node

referenced in the edge list can be missing in the node list. Tools such as `igraph` can detect these issues, and in PostgreSQL you can use the referential integrity checks with primary and foreign keys for this.

- *For large networks, graph databases can be useful:* It is possible to leverage a relational database such as PostgreSQL for large network datasets, allowing us to use referential integrity checks and indexing. However, PostgreSQL is restricted to tabular data, and does not have any functionality to deal with graph operations – for example, it is difficult to find the neighbors of the neighbors of a given node. For this purpose, it is possible to use a specialized type of database designed for graphs, such as Neo4j.