Clean Data

Save time by discovering effortless strategies for cleaning, organizing, and manipulating your data

Megan Squire
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Speaking the Lingua Franca – Data Conversions'
- A synopsis of the book’s content
- More information on Clean Data

About the Author

Megan Squire is a professor of computing sciences at Elon University. She has been collecting and cleaning dirty data for two decades. She is also the leader of FLOSSmole.org, a research project to collect data and analyze it in order to learn how free, libre, and open source software is made.
Clean Data

"Pray, Mr. Babbage, if you put into the machine the wrong figures, will the right answer come out?"

– Charles Babbage (1864)

"Garbage in, garbage out"

– The United States Internal Revenue Service (1963)

"There are no clean datasets."

– Josh Sullivan, Booz Allen VP in Fortune (2015)

In his 1864 collection of essays, Charles Babbage, the inventor of the first calculating machine, recollects being dumbfounded at the "confusion of ideas" that would prompt someone to assume that a computer could calculate the correct answer despite being given the wrong input. Fast-forward another 100 years, and the tax bureaucracy started patiently explaining "garbage in, garbage out" to express the idea that even for the all-powerful tax collector, computer processing is still dependent on the quality of its input. Fast-forward another 50 years to 2015: a seemingly magical age of machine learning, autocorrect, anticipatory interfaces, and recommendation systems that know me better than I know myself. Yet, all of these helpful algorithms still require high-quality data in order to learn properly in the first place, and we lament "there are no clean datasets".

This book is for anyone who works with data on a regular basis, whether as a data scientist, data journalist, software developer, or something else. The goal is to teach practical strategies to quickly and easily bridge the gap between the data we want and the data we have. We want high-quality, perfect data, but the reality is that most often, our data falls far short. Whether we are plagued with missing data, data in the wrong format, data in the wrong location, or anomalies in the data, the result is often, to paraphrase rapper Notorious B.I.G., "more data, more problems".

Throughout the book, we will envision data cleaning as an important, worthwhile step in the data science process: easily improved, never ignored. Our goal is to reframe data cleaning away from being a dreaded, tedious task that we must slog through in order to get to the real work. Instead, armed with a few tried-and-true procedures and tools, we will learn that just like in a kitchen, if you wash your vegetables first, your food will look better, taste better, and be better for you. If you learn a few proper knife skills, your meat will be more succulent and your vegetables will be cooked more evenly. The same way that a great chef will have their favorite knives and culinary traditions, a great data scientist will want to work with the very best data possible and under the very best conditions.
What This Book Covers

Chapter 1, Why Do You Need Clean Data? motivates our quest for clean data by showing the central role of data cleaning in the overall data science process. We follow with a simple example showing some dirty data from a real-world dataset. We weigh the pros and cons of each potential cleaning process, and then we describe how to communicate our cleaning changes to others.

Chapter 2, Fundamentals – Formats, Types, and Encodings, sets up some foundational knowledge about file formats, compression, and data types, including missing and empty data and character encodings. Each section has its own examples taken from real-world datasets. This chapter is important because we will rely on knowledge of these basic concepts for the rest of the book.

Chapter 3, Workhorses of Clean Data – Spreadsheets and Text Editors, describes how to get the most data cleaning utility out of two common tools: the text editor and the spreadsheet. We will cover simple solutions to common problems, including how to use functions, search and replace, and regular expressions to correct and transform data. At the end of the chapter, we will put our skills to test using both of these tools to clean some real-world data regarding universities.

Chapter 4, Speaking the Lingua Franca – Data Conversions, focuses on converting data from one format to another. This is one of the most important data cleaning tasks, and it is useful to have a variety of tools at hand to easily complete this task. We first proceed through each of the different conversions step by step, including back and forth between common formats such as comma-separated values (CSV), JSON, and SQL. To put our new data conversion skills into practice, we complete a project where we download a Facebook friend network and convert it into a few different formats so that we can visualize its shape.

Chapter 5, Collecting and Cleaning Data from the Web, describes three different ways to clean data found inside HTML pages. This chapter presents three popular tools to pull data elements from within marked-up text, and it also provides the conceptual foundation to understand other methods besides the specific tools shown here. As our project for this chapter, we build a set of cleaning procedures to pull data from web-based discussion forums.

Chapter 6, Cleaning Data in PDF Files, introduces several ways to meet this most stubborn and all-too-common challenge for data cleaners: extracting data that has been stored in Adobe's Portable Document Format (PDF) files. We first examine low-cost tools to accomplish this task, then we try a few low-barrier-to-entry tools, and finally, we experiment with the Adobe non-free software itself. As always, we use real-world data for our experiments, and this provides a wealth of experience as we learn to work through problems as they arise.
Chapter 7, *RDBMS Cleaning Techniques*, uses a publicly available dataset of tweets to demonstrate numerous strategies to clean data stored in a relational database. The database shown is MySQL, but many of the concepts, including regular-expression based text extraction and anomaly detection, are readily applicable to other storage systems as well.

Chapter 8, *Best Practices for Sharing Your Clean Data*, describes some strategies to make your hard work as easy for others to use as possible. Even if you never plan to share your data with anyone else, the strategies in this chapter will help you stay organized in your own work, saving you time in the future. This chapter describes how to create the ideal data package in a variety of formats, how to document your data, how to choose and attach a license to your data, and also how to publicize your data so that it can live on if you choose.

Chapter 9, *Stack Overflow Project*, guides you through a full-length project using a real-world dataset. We start by posing a set of authentic questions that we can answer about that dataset. In answering this set of questions, we will complete the entire data science process introduced in *Chapter 1, Why Do You Need Clean Data?* And we will put into practice many of the cleaning processes we learned in the previous chapters. In addition, because this dataset is so large, we will introduce a few new techniques to deal with the creation of test datasets.

Chapter 10, *Twitter Project*, is a full-length project that shows how to perform one of the hottest and fastest-changing data collection and cleaning tasks out there right now: mining Twitter. We will show how to find and collect an existing archive of publicly available tweets on a real-world current event while adhering to legal restrictions on the usage of the Twitter service. We will answer a simple question about the dataset while learning how to clean and extract data from JSON, the most popular format in use right now with API-accessible web data. Finally, we will design a simple data model for long-term storage of the extracted and cleaned data and show how to generate some simple visualizations.
Last summer, I took a cheese-making class at a local cooking school. One of the first things we made was ricotta cheese. I was thrilled to learn that ricotta can be made in about an hour using just milk and buttermilk, and that buttermilk itself can be made from milk and lemon juice. In a kitchen, ingredients are constantly transformed into other ingredients, which will in turn be transformed into delicious meals. In our data science kitchen, we will routinely perform conversions from one data format to another. We might need to do this in order to perform various analyses, when we want to merge datasets together, or if we need to store a dataset in a new way.

A lingua franca is a language that is adopted as a common standard in a conversation between speakers of different languages. In converting data, there are several data formats that can serve as a common standard. We covered some of these in Chapter 2, Fundamentals – Formats, Types, and Encodings. JSON and CSV are two of the most common. In this chapter, we will spend some time learning:

- How to perform some quick conversions into JSON and CSV from software tools and languages (Excel, Google Spreadsheets, and phpMyAdmin).
- How to write Python and PHP programs to generate different text formats and convert between them.
- How to implement data conversions in order to accomplish a real-world task. In this project, we will download a friend network from Facebook using the netvizz software, and we will clean the data and convert it into the JSON format needed to build a visualization of your social network in D3. Then, we will clean the data in a different way, converting it into the Pajek format needed by the social network package called networkx.
Quick tool-based conversions
One of the quickest and easiest ways to convert a small to medium amount of data is just to ask whatever software tool you are using to do it for you. Sometimes, the application you are using will already have the option to convert the data into the format you want. Just as with the tips and tricks in Chapter 3, Workhorses of Clean Data – Spreadsheets and Text Editors, we want to take advantage of these hidden features in our tools, if at all possible. If you have too much data for an application-based conversion, or if the particular conversion you want is not available, we will cover programmatic solutions in the upcoming sections, Converting with PHP and Converting with Python.

Spreadsheet to CSV
Saving a spreadsheet as a delimited file is quite straightforward. Both Excel and Google spreadsheets have File menu options for Save As; in this option, select CSV (MS DOS). Additionally, Google Spreadsheets has the options to save as an Excel file and save as a tab-delimited file. There are a few limitations with saving something as CSV:

- In both Excel and Google Spreadsheets, when you use the Save As feature, only the current sheet will be saved. This is because, by nature, a CSV file describes only one set of data; therefore, it cannot have multiple sheets in it. If you have a multiple-sheet spreadsheet, you will need to save each sheet as a separate CSV file.
- In both these tools, there are relatively few options for how to customize the CSV file, for example, Excel saves the data with commas as the separator (which makes sense as it is a CSV file) and gives no options to enclose data values in quotation marks or for different line terminators.

Spreadsheet to JSON
JSON is a little trickier to contend with than CSV. Excel does not have an easy JSON converter, though there are several converter tools online that purport to convert CSV files for you into JSON.

Google Spreadsheets, however, has a JSON converter available via a URL. There are a few downsides to this method, the first of which is that you have to publish your document to the Web (at least temporarily) in order to access the JSON version of it. You will also have to customize the URL with some very long numbers that identify your spreadsheet. It also produces a lot of information in the JSON dump—probably more than you will want or need. Nonetheless, here are some step-by-step instructions to convert a Google Spreadsheet into its JSON representation.
Step one – publish Google spreadsheet to the Web

After your Google spreadsheet is created and saved, select Publish to the Web from the File menu. Click through the subsequent dialogue boxes (I took all the default selections for mine). At this point, you will be ready to access the JSON for this file via a URL.

Step two – create the correct URL

The URL pattern to create JSON from a published Google spreadsheet looks like this:

http://spreadsheets.google.com/feeds/list/key/sheet/public/basic?alt=json

There are three parts of this URL that you will need to alter to match your specific spreadsheet file:

- **list**: (optional) You can change list to, say, cells if you would prefer to see each cell listed separately with its reference (A1, A2, and so on) in the JSON file. If you want each row as an entity, leave list in the URL.
- **key**: Change key in this URL to match the long, unique number that Google internally uses to represent your file. In the URL of your spreadsheet, as you are looking at it in the browser, this key is shown as a long identifier between two slashes, just after the /spreadsheets/d portion of the URL, shown as follows:

  ![Google Spreadsheet URL](image)

- **sheet**: Change the word sheet in the sample URL to od6 to indicate that you are interested in converting the first sheet.

  What does od6 mean? Google uses a code to represent each of the sheets. However, the codes are not strictly in numeric order. There is a lengthy discussion about the numbering scheme on the question on this Stack Overflow post and its answers: [http://stackoverflow.com/questions/11290337/](http://stackoverflow.com/questions/11290337/)
To test this procedure, we can create a Google spreadsheet for the universities and the counts that we generated from the exercise at the end of the example project in *Chapter 3, Workhorses of Clean Data – Spreadsheets and Text Editors*. The first three rows of this spreadsheet look like this:

<table>
<thead>
<tr>
<th>University</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yale University</td>
<td>26</td>
</tr>
<tr>
<td>Princeton University</td>
<td>25</td>
</tr>
<tr>
<td>Cornell University</td>
<td>24</td>
</tr>
</tbody>
</table>

My URL to access this file via JSON looks like this:

http://spreadsheets.google.com/feeds/list/1mWIAk_5KNoQHr4vFgPHdm7GX8Vh22WjgAUYHYuXSNM/od6/public/basic?alt=json

Pasting this URL into the browser yields a JSON representation of the data. It has 231 entries in it, each of which looks like the following snippet. I have formatted this entry with added line breaks for easier reading:

```json
{
    "id":{
        "$t":"https://spreadsheets.google.com/feeds/list/1mWIAk_5KNoQHr4vFgPHdm7GX8Vh22WjgAUYHYuXSNM/od6/public/basic/cokwr"
    },
    "updated":{"$t":"2014-12-17T20:02:57.196Z"},
    "category":{
        "term":"http://schemas.google.com/spreadsheets/2006#list"
    },
    "title":{
        "type":"text",
        "$t":"Yale University"
    },
    "content":{
        "type":"text",
        "$t":"_cokwr: 26"
    },
    "link": [{
        "rel":"self",
        "type":"application/atom+xml",
        "href":"https://spreadsheets.google.com/feeds/list/1mWIAk_5KNoQHr4vFgPHdm7GX8Vh22WjgAUYHYuXSNM/od6/public/basic/cokwr"
    }]
}
```
Even with my reformatting, this JSON is not very pretty, and many of these name-value pairs will be uninteresting to us. Nonetheless, we have successfully generated a functional JSON. If we are using a program to consume this JSON, we will ignore all the extraneous information about the spreadsheet itself and just go after the title and content entities and the $t values (Yale University and _cokwr: 26, in this case). These values are highlighted in the JSON shown in the preceding example. If you are wondering whether there is a way to go from a spreadsheet to CSV to JSON, the answer is yes. We will cover how to do exactly that in the Converting with PHP and Converting with Python sections later in this chapter.

**SQL to CSV or JSON using phpMyAdmin**

In this section, we'll discuss two options for writing JSON and CSV directly from a database, MySQL in our case, without using any programming.

First, phpMyAdmin is a very common web-based frontend for MySQL databases. If you are using a modern version of this tool, you will be able to export an entire table or the results of a query as a CSV or JSON file. Using the same enron database we first visited in Chapter 1, Why Do You Need Clean Data?, consider the following screenshot of the Export tab, with JSON selected as the target format for the entire employeelist table (CSV is also available in this select box):
The process to export the results of a query is very similar, except that instead of using the Export tab on the top of the screen, run the SQL query and then use the Export option under Query results operations at the bottom of the page, shown as follows:

```
SELECT concat(firstName,  " ", lastName) as name, email_id
FROM employeelist
ORDER BY lastName;
```

When we export the results as JSON, phpMyAdmin shows us 151 values formatted like this:

```
{
  "name": "Lysa Akin",
  "email_id": "lysa.akin@enron.com"
}
```

The phpMyAdmin tool is a good one, and it is effective for converting moderate amounts of data stored in MySQL, especially as the results of a query. If you are using a different RDBMS, your SQL interface will likely have a few formatting options of its own that you should explore.

Another strategy is to bypass phpMyAdmin entirely and just use your MySQL command line to write out a CSV file that is formatted the way you want:

```
SELECT concat(firstName,  " ", lastName) as name, email_id
INTO OUTFILE 'enronEmployees.csv'
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n'
FROM employeelist;
```

This will write a comma-delimited file with the name specified (employees.csv). It will be written into the current directory.
What about JSON? There is no very clean way to output JSON with this strategy, so you should either use the phpMyAdmin solution shown previously, or use a more robust solution written in PHP or Python. These programmatic solutions are covered in further sections, so keep reading.

**Converting with PHP**

In our Chapter 2, *Fundamentals – Formats, Types, and Encodings*, in a discussion on JSON numeric formatting, we briefly showed how to use PHP to connect to a database, run a query, build a PHP array from the results, and then print the JSON results to the screen. Here, we will first extend this example to write a file rather than print to the screen and also to write a CSV file. Next, we will show how to use PHP to read in JSON files and convert to CSV files, and vice versa.

**SQL to JSON using PHP**

In this section, we will write a PHP script to connect to the *enron* database, run a SQL query, and export it as a JSON-formatted file. Why write a PHP script for this instead of using phpMyAdmin? Well, this strategy will be useful in cases where we need to perform additional processing on the data before exporting it or where we suspect that we have more data than what a web-based application (such as phpMyAdmin) can run:

```php
<?php
// connect to db, set up query, run the query
$dbc = mysqli_connect('localhost','username','password','enron')
or die('Error connecting to database!' . mysqli_error());

$select_query = "SELECT concat(firstName, " \", lastName) as
    name, email_id FROM employeelist ORDER BY lastName";

$select_result = mysqli_query($dbc, $select_query);
if (!$select_result)
die ("SELECT failed! [$select_query]" . mysqli_error());

// ----JSON output----
// build a new array, suitable for json
$counts = array();
while($row = mysqli_fetch_array($select_result))
{
```
Speaking the Lingua Franca – Data Conversions

```php
// add onto the json array
array_push($counts, array('name' => $row['name'], 'email_id' => $row['email_id']));

// encode query results array as json
$json_formatted = json_encode($counts);

// write out the json file
file_put_contents("enronEmail.json", $json_formatted);

?>
```

This code writes a JSON-formatted output file to the location you specify in the `file_put_contents()` line.

### SQL to CSV using PHP

The following code snippet shows how to use the PHP file output stream to create a CSV-formatted file of the results of a SQL query. Save this code as a .php file in the script-capable directory on your web server, and then request the file in the browser. It will automatically download a CSV file with the correct values in it:

```php
<?php
// connect to db, set up query, run the query
$dbc = mysqli_connect('localhost','username','password','enron')
or die('Error connecting to database!' . mysqli_error());

$select_query = "SELECT concat(firstName, " ", lastName) as name, email_id FROM employeelist ORDER BY lastName";

$select_result = mysqli_query($dbc, $select_query);
if (!$select_result)
die("SELECT failed! [$select_query]" . mysqli_error());

// ----CSV output----
// set up a file stream
$file = fopen('php://output', 'w');
if ($file && $select_result)
{
    header('Content-Type: text/csv');
    header('Content-Disposition: attachment;
filename="enronEmail.csv"');
    // write each result row to the file in csv format
```
while($row = mysqli_fetch_assoc($select_result))
{
    fputcsv($file, array_values($row));
}
?>

The results are formatted as follows (these are the first three lines only):

"Lysa Akin",lysa.akin@enron.com
"Phillip Allen",k..allen@enron.com
"Harry Arora",harry.arora@enron.com

If you are wondering whether Phillip’s e-mail is really supposed to have two dots in it, we can run a quick query to find out how many of Enron’s e-mails are formatted like that:

SELECT CONCAT(firstName,  " ", lastName) AS name, email_id
FROM employeelists
WHERE email_id LIKE "%..%"
ORDER BY name ASC;

It turns out that 24 of the e-mail addresses have double dots like that.

**JSON to CSV using PHP**

Here, we will use PHP to read in a JSON file and convert it to CSV and output a file:

```php
<?php
// read in the file
$json = file_get_contents("outfile.json");
// convert JSON to an associative array
$array = json_decode ($json, true);
// open the file stream
$file = fopen('php://output', 'w');
header('Content-Type: text/csv');
header('Content-Disposition: attachment;
filename="enronEmail.csv"');
// loop through the array and write each item to the file
foreach ($array as $line)
{
    fputcsv($file, $line);
}
?>
```
This code will create a CSV with each line in it, just like the previous example. We should be aware that the `file_get_contents()` function reads the file into the memory as a string, so you may find that for extremely large files, you will need to use a combination of the `fread()`, `fgets()`, and `fclose()` PHP functions instead.

CSV to JSON using PHP

Another common task is to read in a CSV file and write it out as a JSON file. Most of the time, we have a CSV in which the first row is a header row. The header row lists the column name for each column in the file, and we would like each item in the header row to become the keys for the JSON-formatted version of the file:

```php
<?php
    $file = fopen('enronEmail.csv', 'r');
    $headers = fgetcsv($file, 0, ',');
    $complete = array();

    while ($row = fgetcsv($file, 0, ',')) {
        $complete[] = array_combine($headers, $row);
    }
    fclose($file);
    $json_formatted = json_encode($complete);
    file_put_contents('enronEmail.json', $json_formatted);
?>
```

The result of this code on the `enronEmail.csv` file created earlier, with a header row, is as follows:

```json
[{
  "name": "Lysa Akin",
  "email_id": "lysa.akin@enron.com"
},
{
  "name": "Phillip Allen",
  "email_id": "k..allen@enron.com"
},
{
  "name": "Harry Arora",
  "email_id": "harry.arora@enron.com"
}...]
```

For this example, of the 151 results in the actual CSV file, only the first three rows are shown.

Converting with Python

In this section, we describe a variety of ways to manipulate CSV into JSON, and vice versa, using Python. In these examples, we will explore different ways to accomplish this goal, both using specially installed libraries and using more plain-vanilla Python code.
CSV to JSON using Python

We have found several ways to convert CSV files to JSON using Python. The first of these uses the built-in csv and json libraries. Suppose we have a CSV file that has rows like this (only the first three rows shown):

<table>
<thead>
<tr>
<th>name</th>
<th>email_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Lysa Akin&quot;</td>
<td>&quot;<a href="mailto:lysa.akin@enron.com">lysa.akin@enron.com</a>&quot;</td>
</tr>
<tr>
<td>&quot;Phillip Allen&quot;</td>
<td>&quot;<a href="mailto:k..allen@enron.com">k..allen@enron.com</a>&quot;</td>
</tr>
<tr>
<td>&quot;Harry Arora&quot;</td>
<td>&quot;<a href="mailto:harry.arora@enron.com">harry.arora@enron.com</a>&quot;</td>
</tr>
</tbody>
</table>

We can write a Python program to read these rows and convert them to JSON:

```python
import json
import csv

# read in the CSV file
with open('enronEmail.csv') as file:
    file_csv = csv.DictReader(file)
    output = '['
    # process each dictionary row
    for row in file_csv:
        # put a comma between the entities
        output += json.dumps(row) + ',
    output = output.rstrip(',') + ']

# write out a new file to disk
f = open('enronEmailPy.json', 'w')
f.write(output)
f.close()
```

The resulting JSON will look like this (only the first two rows are shown):

```json
["email_id": "lysa.akin@enron.com", "name": "Lysa Akin"],
"email_id": "k..allen@enron.com", "name": "Phillip Allen"],...
```

One nice thing about using this method is that it does not require any special installations of libraries or any command-line access, apart from getting and putting the files you are reading (CSV) and writing (JSON).
CSV to JSON using csvkit

The second method of changing CSV into JSON relies on a very interesting Python toolkit called csvkit. To install csvkit using Canopy, simply launch the Canopy terminal window (you can find it inside Canopy by navigating to Tools | Canopy Terminal) and then run the pip install csvkit command. All the dependencies for using csvkit will be installed for you. At this point, you have the option of accessing csvkit via a Python program as a library using import csvkit or via the command line, as we will do in the following snippet:

```
csvjson enronEmail.csv > enronEmail.json
```

This command takes a enronEmail.csv CSV file and transforms it to a JSON enronEmail.csvkit.json file quickly and painlessly.

There are several other extremely useful command-line programs that come with the csvkit package, including csvcut, which can extract an arbitrary list of columns from a CSV file, and csvformat, which can perform delimiter exchanges on CSV files or alter line endings or similar cleaning procedures. The csvcut program is particularly helpful if you want to extract just a few columns for processing. For any of these command-line tools, you can redirect its output to a new file. The following command line takes a file called bigFile.csv, cuts out the first and third column, and saves the result as a new CSV file:

```
csvcut bigFile.csv -c 1,3 > firstThirdCols.csv
```

Additional information about csvkit, including full documentation, downloads, and examples, is available at http://csvkit.rtfd.org/.

Python JSON to CSV

It is quite straightforward to use Python to read in a JSON file and convert it to CSV for processing:

```
import json
import csv

with open('enronEmailPy.json', 'r') as f:
    dicts = json.load(f)
out = open('enronEmailPy.csv', 'w')
writer = csv.DictWriter(out, dicts[0].keys())
writer.writeheader()
writer.writerows(dicts)
out.close()
```
This program takes a JSON file called enronEmailPy.json and exports a CSV-formatted version of this file using the keys for the JSON as the header row new file, called enronEmailPy.csv.

The example project

In this chapter, we have focused on converting data from one format to another, which is a common data cleaning task that will need to be done time and again before the rest of the data analysis project can be completed. We focused on some very common text formats (CSV and JSON) and common locations for data (files and SQL databases). Now, we are ready to extend our basic knowledge of data conversions with a sample project that will ask us to make conversions between some less standardized— but still text-based— data formats.

In this project, we want to investigate our Facebook social network. We will:

1. Download our Facebook social network (friends and relationships between them) using netvizz into a text-based file format called **Graph Description Format** (GDF).
2. Build a graphical representation of a Facebook social network showing the people in our network as nodes and their friendships as connecting lines (called edges) between these nodes. To do this, we will use the D3 JavaScript graphing library. This library expects a JSON representation of the data in the network.
3. Calculate some metrics about the social network, such as the size of the network (known as the **degree** of the network) and the shortest path between two people our network. To do this, we will use the networkx package in Python. This package expects data in a text-based format, called the **Pajek** format.

The primary goal of this project will be to show how to reconcile all these different expected formats (GDF, Pajek, and JSON) and perform conversions from one format to another. Our secondary goal will be to actually provide enough sample code and guidance to perform a small analysis of our social network.

Step one – download Facebook data as GDF

For this step, you will need to be logged into your Facebook account. Use Facebook’s search box to find the netvizz app, or use this URL to directly link to the netvizz app: https://apps.facebook.com/netvizz/.
Once on the netvizz page, click on personal network. The page that follows explains that clicking on the start button will provide a downloadable file with two items in it: a GDF format file that lists all your friends and the connections between them and a tab-delimited Tab Separated Values (TSV) stats file. We are primarily interested in the GDF file for this project. Click on the start button, and on the subsequent page, right-click on the GDF file to save it to your local disk, as shown in the following screenshot:

The netvizz Facebook app allows us to download our social network as a GDF file. It may be helpful to also give the file a shorter name at this point. (I called my file personal.gdf and saved it in a directory created just for this project.)

**Step two – look at the GDF file format in a text editor**

Open the file in your text editor (I am using Text Wrangler for this), and note a few things about the format of this file:

1. The file is divided into two parts: nodes and edges.
2. The nodes are found in the first part of the file, preceded by the word nodedef. The list of nodes is a list of all my friends and some basic facts about them (their gender and their internal Facebook identification number). The nodes are listed in the order of the date when the person joined Facebook.
3. The second part of the file shows the edges or connections between my friends. Sometimes, these are also called links. This section of the file is preceded by the word edgedef. The edges describe which of my friends are linked to which other friends.
Here is an excerpt of what a nodes section looks like:

```
nodedef>name VARCHAR,label VARCHAR,sex VARCHAR,locale VARCHAR,agerank INT
1234,Bugs Bunny,male,en_US,296
2345,Daffy Duck,male,en_US,295
3456,Minnie Mouse,female,en_US,294
```

Here is an excerpt of what an edges section looks like. It shows that Bugs (1234) and Daffy (2345) are friends, and Bugs is also friends with Minnie (3456):

```
edgedef>node1 VARCHAR,node2 VARCHAR
1234,2345
1234,3456
3456,9876
```

**Step three – convert the GDF file into JSON**

The task we want to perform is to build a representation of this data as a social network in D3. First, we need to look at the dozens of available examples of D3 to build a social network, such as those available in the D3 galleries of examples, https://github.com/mbostock/d3/wiki/Gallery and http://christopheviau.com/d3List/.

These examples of social network diagrams rely on JSON files. Each JSON file shows nodes and the edges between them. Here is an example of what one of these JSON files should look like:

```
{"nodes": [
    {"name":"Bugs Bunny"},
    {"name":"Daffy Duck"},
    {"name":"Minnie Mouse"}],
"edges": [
    {"source": 0,"target": 2},
    {"source": 1,"target": 3},
    {"source": 2,"target": 3}]
```

The most important thing about this JSON code is to note that it has the same two main chunks as the GDF file did: nodes and edges. The nodes are simply the person's name. The edges are a list of number pairs representing friendship relations. Instead of using the Facebook identification number, though, these pairs use an index for each item in the nodes list, starting with 0.
We do not have a JSON file at this point. We only have a GDF file. How will we build this JSON file? When we look closely at the GDF file, we can see that it looks a lot like two CSV files stacked on top of one another. From earlier in this chapter, we know we have several different strategies to convert from CSV to JSON.

Therefore, we decide to convert GDF to CSV and then CSV to JSON.

Wait; what if that JSON example doesn't look like the JSON files I found online to perform a social network diagram in D3?

Some of the examples of D3 social network visualizations that you may find online will show many additional values for each node or link, for example, they may include extra attributes that can be used to signify a difference in size, a hover feature, or a color change, as shown in this sample: http://bl.ocks.org/christophermanning/1625629. This visualization shows relationships between paid political lobbyists in Chicago. In this example, the code takes into account information in the JSON file to determine the size of the circles for the nodes and the text that is displayed when you hover over the nodes. It makes a really nice diagram, but it is complicated. As our primary goal is to learn how to clean the data, we will work with a pared down, simple example here that does not have many of these extras. Do not worry, though; our example will still build a nifty D3 diagram!

To convert the GDF file to JSON in the format we want, we can follow these steps:

1. Use a text editor to split the personal.gdf file into two files, nodes.gdf and links.gdf.
2. Alter the header row in each file to match the column names we eventually want in the JSON file:
   
   ```
   id,name,gender,lang,num
   1234,Bugs Bunny,male,en_US,296
   2345,Daffy Duck,male,en_US,295
   9876,Minnie Mouse,female,en_US,294
   
   source,target
   1234,2345
   1234,9876
   2345,9876
   ```
3. Use the csvcut utility (part of csvkit discussed previously) to extract the first and second columns from the nodes.gdf file and redirect the output to a new file called nodesCut.gdf:
   ```
   csvcut -c 1,2 nodes.gdf > nodesCut.gdf
   ```
4. Now, we need to give each edge pair an indexed value rather than their full Facebook ID value. The index just identifies this node by its position in the node list. We need to perform this transformation so that the data will easily feed into the D3 force network code examples that we have, with as little refactoring as possible. We need to convert this:

```
source, target
1234, 2345
1234, 9876
2345, 9876
```

into this:

```
source, target
0, 1
0, 2
1, 2
```

Here is a small Python script that will create these index values automatically:

```python
import csv

# read in the nodes
with open('nodesCut.gdf', 'r') as nodefile:
    nodereader = csv.reader(nodefile)
    nodeid, name = zip(*nodereader)

# read in the source and target of the edges
with open('edges.gdf', 'r') as edgefile:
    edgereader = csv.reader(edgefile)
    sourcearray, targetarray = zip(*edgereader)
slist = list(sourcearray)
tlist = list(targetarray)

# find the node index value for each source and target
for n, i in enumerate(nodeid):
    for j, s in enumerate(slist):
        if s == i:
            slist[j] = n - 1
    for k, t in enumerate(tlist):
        if t == i:
            tlist[k] = n - 1

# write out the new edge list with index values
with open('edgelistIndex.csv', 'wb') as indexfile:
    iwriter = csv.writer(indexfile)
    for c in range(len(slist)):
        iwriter.writerow([slist[c], tlist[c]])
```
5. Now, go back to the nodesCut.csv file and remove the id column:
   ```
csvcut -c 2 nodesCut.gdf > nodesCutName.gdf
   ```

6. Construct a small Python script that takes each of these files and writes them out to a complete JSON file, ready for D3 processing:

   ```python
   import csv
   import json

   # read in the nodes file
   with open('nodesCutName.gdf') as nodefile:
       nodefile_csv = csv.DictReader(nodefile)
       noutput = '[
       ncounter = 0;

       # process each dictionary row
       for nrow in nodefile_csv:
           # look for ' in node names, like O'Connor
           nrow['name'] = \
           str(nrow['name']).replace('"','\"').replace('"','\"')
           # put a comma between the entities
           if ncounter > 0:
               noutput += ','
           noutput += json.dumps(nrow)
           ncounter += 1
       noutput += ']
       # write out a new file to disk
       f = open('complete.json','w')
       f.write('{
       f.write('"nodes":' )
       f.write(noutput)

   # read in the edge file
   with open('edgelistIndex.csv') as edgefile:
       edgefile_csv = csv.DictReader(edgefile)
       eoutput = '[
       ecounter = 0;

       # process each dictionary row
       for erow in edgefile_csv:
           # make sure numeric data is coded as number not string
           for ekey in erow:
               try:
                   erow[ekey] = int(erow[ekey])
   ```
except ValueError:
    # not an int
    pass
    # put a comma between the entities
    if ecounter > 0:
        eoutput += ','
eoutput += json.dumps(erow)
ecounter += 1
eoutput += '}

    # write out a new file to disk
f.write(','
f.write('"links":')
f.write(eoutput)
f.write('})
f.close()

Step four – build a D3 diagram
This section shows how to feed our JSON file of nodes and links into a boilerplate example of building a force-directed graph in D3. This code example came from the D3 website and builds a simple graph using the JSON file provided. Each node is shown as a circle, and when you hover your mouse over the node, the person's name shows up as a tooltip:

<!DOCTYPE html>
<!-- this code is based on the force-directed graph D3 example given at: https://gist.github.com/mbostock/4062045 -->

<meta charset="utf-8">
<style>
.node {
    stroke: #fff;
    stroke-width: 1.5px;
}

.link {
    stroke: #999;
    stroke-opacity: .6;
}

</style>
Speaking the Lingua Franca – Data Conversions

```html
<body>
<!-- make sure you have downloaded the D3 libraries and stored them locally -->
<script src="d3.min.js"></script>

<script>
var width = 960, height = 500;
var color = d3.scale.category20();
var force = d3.layout.force()
  .charge(-25)
  .linkDistance(30)
  .size([width, height]);

var svg = d3.select("body").append("svg")
  .attr("width", width)
  .attr("height", height);

d3.json("complete.json", function(error, graph) {
  force
    .nodes(graph.nodes)
    .links(graph.links)
    .start();

  var link = svg.selectAll(".link")
    .data(graph.links)
    .enter().append("line")
    .attr("class", "link")
    .style("stroke-width", function(d) { return Math.sqrt(d.value); });

  var node = svg.selectAll(".node")
    .data(graph.nodes)
    .enter().append("circle")
    .attr("class", "node")
    .attr("r", 5)
    .style("fill", function(d) { return color(d.group); })
    .call(force.drag);

  node.append("title")
    .text(function(d) { return d.name; });

  force.on("tick", function() {
    link.attr("x1", function(d) { return d.source.x; })
    .attr("y1", function(d) { return d.source.y; });
  });
});
</script>
```
The following screenshot shows an example of this social network. One of the nodes has been hovered over, showing the tooltip (name) of that node.

**Step five – convert data to the Pajek file format**

So far, we have converted a GDF file to CSV, and then to JSON, and built a D3 diagram of it. In the next two steps, we will continue to pursue our goal of getting the data in such a format that we can calculate some social network metrics on it.

For this step, we will take the original GDF file and tweak it to become a valid Pajek file, which is the format that is needed by the social network tool called networkx.
The word *pajek* means *spider* in Slovenian. A social network can be thought of as a web made up of nodes and the links between them.

The format of our Facebook GDF file converted to a Pajek file looks like this:

```
*vertices 296
1234 Bugs_Bunny male en_US 296
2456 Daffy_Duck male en_US 295
9876 Minnie_Mouse female en_US 294
*edges
1234 2456
2456 9876
2456 3456
```

Here are a few important things to notice right away about this Pajek file format:

- It is space-delimited, not comma-delimited.
- Just like in the GDF file, there are two main sections of data, and these are labeled, starting with an asterisk `*`. The two sections are the vertices (another word for nodes) and the edges.
- There is a count of how many total vertices (nodes) there are in the file, and this count goes next to the word vertices on the top line.
- Each person's name has spaces removed and replaced with underscores.
- The other columns are optional in the node section.

To convert our GDF file into Pajek format, let's use the text editor, as these changes are fairly straightforward and our file is not very large. We will perform the data cleaning tasks as follows:

1. Save a copy of your GDF file as a new file and call it something like `fbPajek.net` (the `.net` extension is commonly used for Pajek network files).
2. Replace the top line in your file. Currently, it looks like this:
   
   ```
   nodedef>name VARCHAR,label VARCHAR,sex VARCHAR,locale VARCHAR,agerank INT
   ```

   You will need to change it to something like this:
   
   ```
   *vertices 296
   ```

   Make sure the number of vertices matches the number you have in your actual file. This is the count of nodes. There should be one per line in your GDF file.
3. Replace the edges line in your file. Currently, it looks like this:

```sql
edgedef node1 VARCHAR, node2 VARCHAR
```

You will need to change it to look like this:

```text
*edges
```

4. Starting at line 2, replace every instance of a space with an underscore. This works because the only spaces in this file are in the names. Take a look at this:

```
1234, Bugs Bunny, male, en_US, 296
2456, Daffy Duck, male, en_US, 295
3456, Minnie Mouse, female, en_US, 294
```

This action will turn the preceding into this:

```
1234, Bugs_Bunny, male, en_US, 296
2456, Daffy_Duck, male, en_US, 295
3456, Minnie_Mouse, female, en_US, 294
```

5. Now, use find and replace to replace all the instances of a comma with a space. The result for the nodes section will be:

```
*vertices 296
1234 Bugs_Bunny male en_US 296
2456 Daffy_Duck male en_US 295
3456 Minnie_Mouse female en_US 294
```

The result for the edges section will be:

```
*edges
1234 2456
2456 9876
2456 3456
```

6. One last thing; use the find feature of the text editor to locate any of your Facebook friends who have an apostrophe in their name. Replace this apostrophe with nothing. Thus, Cap'n_Crunch becomes:

```
1998988 Capn_Crunch male en_US 137
```

This is now a fully cleaned, Pajek-formatted file.
Step six – calculate simple network metrics

At this point, we are ready to run some simple social network metrics using a Python package like networkx. Even though Social Network Analysis (SNA) is beyond the scope of this book, we can still perform a few calculations quite easily without delving too deeply into the mysteries of SNA.

First, we should make sure that we have the networkx package installed. I am using Canopy for my Python editor, so I will use the Package Manager to search for networkx and install it.

Then, once networkx is installed, we can write some quick Python code to read our Pajek file and output a few interesting facts about the structure of my Facebook network:

```python
import networkx as net

# read in the file
g = net.read_pajek('fb_pajek.net')

# how many nodes are in the graph?
# print len(g)

# create a degree map: a set of name-value pairs linking nodes
# to the number of edges in my network
deg = net.degree(g)

# sort the degree map and print the top ten nodes with the
# highest degree (highest number of edges in the network)
print sorted(deg.iteritems(), key=lambda(k,v): (-v,k))[0:9]
```

The result for my network looks like the following output. The top ten nodes are listed, along with a count of how many of my other nodes each of these links to:

```plaintext
[(u'Bambi', 134), (u'Cinderella', 56), (u'Capn_Crunch', 50),
 (u'Bugs_Bunny', 47), (u'Minnie_Mouse', 47), (u'Cruella_Deville',
  46), (u'Alice_Wonderland', 44), (u'Prince_Charming', 42),
 (u'Daffy_Duck', 42)]
```

This shows that Bambi is connected to 134 of my other friends, but Prince_Charming is only connected to 42 of my other friends.

If you get any Python errors about missing quotations, double-check your Pajek format file to ensure that all node labels are free of spaces and other special characters. In the cleaning procedure explained in the preceding example, we removed spaces and the quotation character, but your friends may have more exotic characters in their names!
Of course, there are many more interesting things you can do with networkx and D3 visualizations, but this sample project was designed to give us a sense of how critical data-cleaning processes are to the successful outcome of any larger analysis effort.

**Summary**

In this chapter, we learned many different ways to convert data from one format to another. Some of these techniques are simple, such as just saving a file in the format you want or looking for a menu option to output the correct format. At other times, we will need to write our own programmatic solution.

Many projects, such as the sample project we implemented in this chapter, will require several different cleaning steps, and we will have to carefully plan out our cleaning steps and write down what we did. Both networkx and D3 are really nifty tools, but they do require data to be in a certain format before we are ready to use them. Likewise, Facebook data is easily available through netvizz, but it too has its own data format. Finding easy ways to convert from one file format to the other is a critical skill in data science.

In this chapter, we performed a lot of conversions between structured and semistructured data. But what about cleaning messy data, such as unstructured text?

In *Chapter 5, Collecting and Cleaning Data from the Web*, we will continue to fill up our data science cleaning toolbox by learning some of the ways in which we can clean pages that we find on the Web.
Where to buy this book

You can buy Clean Data from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.