CS45, Lecture 3: Data Wrangling

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Winter 2023

Contents

1 Lecture Overview

In Lecture 2, we learned about the shell and how to run basic commands in the shell such as `ls`, `cd`, `cat`, `man`, and `wc`. We also learned about how to use the `|` operator to chain commands together. Finally we learned how to redirect output using `<` and `>`, and to append output to the end of a file using `>>`. In today's lecture we will learn how to combine these commands in powerful ways in order to automate tasks effectively, specifically in the context of data manipulation and analysis.

2 What is Data Wrangling?

Have you ever had large amounts of data and needed to look for a specific set of information? Of course you have. Have you ever done so in a tedious, inefficient way, even though you knew there is a better way of doing so? Bets are you have. In this lecture, we learn about how to manipulate data to suit our needs in the most efficient way possible.

The basic idea of data wrangling is that you take some data and convert or transform it into another form that is more useful. Often times, this "other" form is a condensed or sorted subset of the original data.

With the increased reliance on large amounts of data (i.e. "big data"), data wrangling has become increasingly important for effectively analysis.

3 Data Formats

Before we discuss data wrangling itself, it's worth discussing different data formats. The data wrangling technique you choose to use will heavily depend on the file format you are using to store your data. Here are some common file formats that are used to store data and that we will discuss in this class:

- CSV
- XML
- HTML
- JSON
- TXT

A **CSV** file is a comma-separated values file where information is separated by commas. CSV files are plain text files which make them easy to generate. They allow data to be saved in a tabular format (meaning in a table, with rows and columns). CSV files are useful as they can be opened with different applications and are not application specific. CSV files are most often used to analyze data with spreadsheets.

A **XML** file is an Extensible Markup Language (XML) file that is used to store data in hierarchical format. XML files were created for storing documents in a way that both humans and machines could read. A XML file consists of tags that define a hierarchy within the document.

A **HTML** file is a Hypertext Markup Language file that is used to store data in hierarchical format, specifically for a webpage. HTML files are very similar to XML files in that they are both read by both humans and machines and they both contain tags to define a hierarchy within the document. The key difference between XML and HTML files is that HTML files use a predefined set of tags while XML files do not have any constraints on what tags can be used.

A **JSON** file is a JavaScript Object Notation file that stores structured data in the form of JavaScript objects. JSON files are often used for transmitting data in web applications.

A **TXT** file is a plaintext file that contains data in the form of lines. TXT files have no special formatting (bold, italic, etc.).

4 `grep` and `sed`

Now onto data wrangling! We've already seen a basic example of data wrangling with the `|` operator. Recall that the pipe operator is used to transfer the output of one command to become the input of another command. Consider the command `ls /Documents | grep -i transcript`. This command lists all of the files in my `Documents` folder and searches for files that mention transcript (case insensitive). In this case, we start with a command to produce the data (`ls /Documents`) and then we use a second command to wrangle the data (`grep -i transcript`) by searching for files that contain the word transcript. You will find that the `|` command is often used for data wrangling in this way.

One common use case for data wrangling with the `|` operator is when looking at logs. System logs keep a record of operating system events on a machine. As you can imagine, system logs record a *lot* of information. It is infeasible to read through an entire system log, which makes it a perfect candidate for data wrangling.

In order to display a system's log, you can use the `log show` command for macOS, or the `journalctl` command for Linux. Once we have our log, we can use the `grep` command in order to search for log entries of interest. You might consider running `log show | grep -i Chrome` in order to show all log entries related to Chrome. Logs produce a lot of data so we may want to limit the amount of data we are interested in. Let's limit the amount of data to log entries just from the past 24 hours using `log show --last 1d`. Now we can search for all entries mentioning Chrome in the past 24 hours.

We can even use this model to search for entries on a remote server. We'll talk more about `ssh` in a later lecture, but for now, we just need to know that we can use `ssh` to log into a remote machine. I will use `ssh` to log into a "honeypot" server that the CS45 staff set up. A "honeypot" is a server or machine that attempts to lure potential attackers by inviting those attackers to log on or access said machine or server. We can search for everything related to ssh on our honeypot server: `ssh adrazen@192.9.152.85 journalctl | grep sshd`. Note that we are using a pipe to stream a remote file (i.e. our remote file is the system log on the honeypot server) through `grep` on our local computer.

This is still a lot of content. Let's imagine we are interested in just looking at when a user was disconnected from our honeypot server. In that case we might want to search for the phrase "Disconnected from". In this case, we could use

something like this:
`ssh adrazen@192.9.152.85 journalctl | grep sshd | grep "Disconnected from"`

This will work but we can do better. Given we are using a remote machine, we will be sending the data from the log back to our local machine in order to run `grep sshd` and `grep "Disconnected from"`.

Instead, we should try to run the entire data wrangling pipeline on our remote machine in order to avoid having to send data across. We can do this by adding quotes:
`ssh adrazen@192.9.152.85 'journalctl | grep sshd | grep "Disconnected from"'`

This is still fairly noisy. Perhaps we are are just interested in knowing *who* was disconnected. We can use another tool here called `sed` to parse out the usernames of users who were disconnected from the honeypot server.

`sed` is a stream editor built into Unix. `sed` can be used for searching a file (similar to `grep`), deleting lines from a file, adding lines to a file, and substituting text in a file. We will be using `sed` for substitution, known as the `s` command.

The `s` command in `sed` uses the following pattern `s/REGEX/SUBSTITUTION` where `REGEX` represents a regular expression for the phrase we are looking to matched, and `SUBSTITUTION` represents what we want to change the matched phrase to. In our case, we are looking to take out some noise from our log entries so we can use the following `sed` command to remove redundant information about which machine the error came from:
`sed 's/.*Disconnected from // '`

We can now add this to our data wrangling pipeline as follows:
`ssh adrazen@myth.stanford.edu journalctl | grep sshd | grep "Disconnected from" | sed 's/.*Disconnected from //'`

If we are interested in removing noise from only the first few log entries, we can specify a range of lines in our `sed` command. Perhaps we are only interested in removing noise from the first 10 log entries: `sed '1,10 s/.*Disconnected from //'`

This is good, but it isn't good enough. We are interested in only the usernames of users who signed in to our machine. We will need to write a regular expression to match everything in the line except the username.

5 Regular Expressions

A **regular expression** is a series of characters that specifies a search pattern. Most ASCII characters carry their normal meaning but some characters have special matching behavior. There is some variation between different implementations of regular expressions, but here are some general patterns. First, let's look at groups of characters, which specify *which* characters we are interested in:

- `.` means any one single character (except the newline character)
- `[abc]` means any of the characters included inside the square brackets, which in this case would be `a`, `b`, or `c`
- `[a-z]` means any characters in the range `a` through `z`
- `(a|b)` means either `a` or `b`

Next, let's look at quantifiers, which specify *how many* characters we are interested in:

- `*` specifies that 0 or more of the proceeding characters match
- `+` specifies that 1 or more of the proceeding characters match
- `{X}` specifies that exactly `X` characters should match

Finally, we can use *anchors* to specify what we expect at the beginning or end of a multi-component pattern:

- `^` specifies the start of the line
- `$` specifies the end of the line

Getting used to regular expressions can be very tricky so we recommend using a RegEx cheat sheet as well as a RegEx tester.

5.1 RegEx Example: Email Matching

Let's take a look at an example of using a regular expression to match email addresses. (The regex I will be presenting here does not actually match *all* email addresses but it will 99% of them.)

Let's take my email (`adrazen@stanford.edu)` and try to write a regular expression to match it. The easiest approach to writing a regular expression is to find delimiters to help divide the text you are looking to match into manageable chunks. In the case of my email address, we can choose `@` and `.` as our delimiters.

Let's first focus on writing a regular expression for everything that comes before the `@` sign. In other words, let's write a regular expression to match the text `adrazen`. We can begin by thinking about *which* characters are allowed. We know that this first part of an email address can consist of characters `A` through `Z` (both upper and lower case), any digit `0` through `9` as well as `.`, `_`, `%`, `+`, and `-`. Thus, the allowed character set is `[A-Za-z0-9._%+-]`. Next, we will want to think about *how many* of these characters are allowed.

Given that the username portion of the email address can be as long as you want (more or less), we will say that we can use as many of these characters as you want as long as you have at least one. (Technically, the username portion

of the email address should be 64 characters or less.) Altogether, this gives us the following for the first portion of the email address: `[A-Za-z0-9._%+-]+`.

Next we will consider the portion of the email address that is between the `@` symbol and the `.`. This refers to the email domain name and is allowed to consist of any of the following characters: characters `A` through `Z` (both upper and lower case), any digit `0` through `9` as well as `.`, and `-`. The allowed character set is therefore `[A-Za-z0-9.-]`.

Again, the domain name is allowed to be more or less as long as we want, but at least 1 character long. (Technically, the username portion of the email address should be 255 characters or less.) The regex to match this portion of the email address is: `[A-Za-z0-9.-]+`.

Finally, we have the component after the `.`. This is known as the top-level domain (TLD). According to rules for the TLD, it must be at least 2 characters long and is allowed to consist of any alphabetic characters. Thus, the regex to match this portion of the email address is `[A-Za-z]`.

Finally, we can combine these individual regexes together and we get the following: `[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+[A-Za-z]{2,}`.

5.2 Extracting usernames

Now that we have a basic understanding of how regexes work, let's return to our initial example of trying to extract the usernames from our log file on the honeypot server. The lines in our log file look something like this:

```
Jan 13 20:58:45 honeypot sshd[70942]: Disconnected from authenticating user root 52.142.11.171 port 1024 [preauth]
Jan 13 20:59:25 honeypot sshd[70946]: Disconnected from invalid user dachuang 158.101.97.210 port 48426 [preauth]
Jan 13 20:59:44 honeypot sshd[70949]: Disconnected from authenticating user root 143.110.153.150 port 42314 [preauth]
Jan 13 21:00:24 honeypot sshd[70951]: Disconnected from authenticating user root 5.78.40.253 port 56548 [preauth]
Jan 13 21:00:46 honeypot sshd[70953]: Disconnected from invalid user weiwei 170.64.134.218 port 38632 [preauth]
Jan 13 21:00:53 honeypot sshd[70955]: Disconnected from invalid user web_proj 143.110.153.150 port 36600 [preauth]
Jan 13 21:01:04 honeypot sshd[70958]: Disconnected from authenticating user root 92.27.157.252 port 33673 [preauth]
Jan 13 21:01:29 honeypot sshd[70963]: Disconnected from invalid user klaus 52.142.11.171 port 1024 [preauth]
Jan 13 21:01:39 honeypot sshd[70965]: Disconnected from authenticating user games 5.78.40.253 port 38640 [preauth]
Jan 13 21:01:59 honeypot sshd[70967]: Disconnected from authenticating user root 143.110.153.150 port 59124 [preauth]
Jan 13 21:02:22 honeypot sshd[70969]: Disconnected from authenticating user root 92.27.157.252 port 45223 [preauth]
Jan 13 21:02:45 honeypot sshd[70971]: Disconnected from invalid user es 5.78.40.253 port 54572 [preauth]
Jan 13 21:02:46 honeypot sshd[70973]: Disconnected from authenticating user root 170.64.134.218 port 38602 [preauth]
```

Our goal is to extract the usernames from this data. To do this, we will first write a regular expression to match the entire line. We will then work on extracting the username from that regular expression.

Let's divide each line into three components. First, we have the component before the username (e.g. `Jan 13 21:02:46 honeypot sshd[70973]: Disconnected from authenticating user`). Then, we have the username itself (e.g. `root`). Finally, we have everything after the username (e.g. `5.78.40.253 port 54572 [preauth]`). For the first component, we can write the regular expression as follows: `.* Disconnected from (authenticating |invalid )?user`. For the username, we will assume that the username can consist of any characters. Thus, we will use the following pattern to match the username `.*`. Finally, for the component after the username, we will use the following expression:
`[0-9.]+ port [0-9]+ (preauth)?`

Now that we have matched the entire line, we want to extract out the username. In order to extract the username, we first need to identify where in our RegEx our username is located. If we look at our RegEx, we will find that portion in the middle with `.*` is used to match our username. We can now use a capture group to, well, capture the username and then use it later. Generally speaking, in a regular expression, a **capture group** is any grouping inside of parentheses which allows us to remember a value in order to reuse it later. In fact, our regular expression already consists of two capture groups: `(authenticating |invalid )` as well as `( preauth)`. We will now create another capture group by adding parentheses around the matching for a username: `(.*)`. Now, we can refer back to this part of the text. Given this is the second capture group, we will use `\2` to refer to this capture group.

We can no use this in order to build up our `sed` command. Given we want to only extract the username, we will match the entire line and then in the substitution portion, we will simply replace with just the username.

```
ssh adrazen@192.9.152.85 journalctl
    | grep sshd
    | grep "Disconnected from"
    | sed -E 's/.*Disconnected from (invalid |authenticating )?user
     (.*) \[0-9.\]+ port [0-9]+( \[preauth\])?$/\2/'
```

5.3 RegEx Applications

Regular expressions are especially useful given they can be embedded inside of other tools. We've already seen how a regex expression can be used for substitution in `sed`. Regular expressions can also be used inside of applications such as Excel and Google Sheets that support data processing.

6 Useful Commands

6.1 `sort` and `uniq`

Now that we have extracted just the usernames, we can run some analysis on our data. There are a number of useful commands that will help us run this analysis. We can use the `sort` command to sort the usernames alphabetically. (More generally, `sort` is a command that can be used for sorting alphabetically or numerically).

We can also use `uniq` to find unique usernames within our data. If we call `uniq` with the `-c` flag, then we can get the number of occurrences for each username. `uniq -c` will collapse consecutive lines that are the same into a single line, prefixed with a count of the number of occurrences.

If we want to find which usernames appear most often, we can call `sort` again after we have collapsed usernames with their counts. This time, we will call `sort` with the `-n` flag in order to get a numeric sorting. Our final pipeline will look as follows:

```
ssh adrazen@192.9.152.85 journalctl
    | grep sshd
    | grep "Disconnected from"
    | sed -E 's/.*Disconnected from (invalid |authenticating )?user
     (.*) \[0-9.\]+ port [0-9]+( \[preauth\])?$/\2/'
    | sort
    | uniq -c
    | sort -n
```

6.2 `head` and `tail`

If we want to get the usernames of the ten users that were disconnected most often from our honeypot, then we can use the `tail` command with the `-n10` flag. The `tail` command prints the last `X` lines of a file. Given that the file is sorted with ascending counts, we will use the `tail` command.

If we want to get the usernames of the ten users that were disconnected *least* often from our honeypot, then we can use the `head` command with the `-n10` flag. The `head` command prints the first `X` lines of a file.

6.3 `xargs`

The final useful command that is worth knowing about is `xargs`. `xargs` is a command that allows you to use the output of one command as the arguments to another command. Note that this is different from using the output of one command as the input to another command.

To understand how to use `xargs`, we can look at an example. Imagine we have a file named `filenames.txt` which contains a bunch of file names. If we run `cat` on `filenames.txt`, we can see the contents of `filenames.txt` in the Terminal.

```
adrazen@ayelet-computer ~ % cat file_names.txt
homework.txt
program.py
todo-list.txt
random.txt
```

Now let's imagine that we want to create a new file for each filename in `filenames.txt`. One way to do this would be to run `cat filenames.txt` and to manually create new files using the `touch` command. This would work, but would be horribly inefficient.

Let's see how we might use the `xargs` command to streamline this process. Remember that `xargs` takes the *output* from a first command and passes them as the *arguments* to the second command. In our case, the output of `cat filenames.txt` is the names of the files we are interested in creating using `touch`. Using `xargs`, we can pass these file names from the output of `cat` to be the arguments to `touch`:
`cat filenames.txt | xargs touch`

7 Other Tools

This lecture covers just a subset of the tools that you may want to use for data wrangling and analysis. Here are a few other tools that you may be interested in using:

`awk` is a scripting language for manipulating data and generating reports.

`R` is another programming language that is great at data analysis and plotting.

`perl` is a programming language for text manipulation