Data Wrangling · the missing semester of your cs education

missing.csail.mit.edu/2020/data-wrangling

Data Wrangling

Have you ever wanted to take data in one format and turn it into a different format? Of course you have! That, in very general terms, is what this lecture is all about. Specifically, massaging data, whether in text or binary format, until you end up with exactly what you wanted.

We've already seen some basic data wrangling in past lectures. Pretty much any time you use the `|` operator, you are performing some kind of data wrangling. Consider a command like `journalctl | grep -i intel`. It finds all system log entries that mention Intel (case insensitive). You may not think of it as wrangling data, but it is going from one format (your entire system log) to a format that is more useful to you (just the intel log entries). Most data wrangling is about knowing what tools you have at your disposal, and how to combine them.

Let's start from the beginning. To wrangle data, we need two things: data to wrangle, and something to do with it. Logs often make for a good use-case, because you often want to investigate things about them, and reading the whole thing isn't feasible. Let's figure out who's trying to log into my server by looking at my server's log:

```
ssh myserver journalctl
```

That's far too much stuff. Let's limit it to ssh stuff:

```
ssh myserver journalctl | grep sshd
```

Notice that we're using a pipe to stream a *remote* file through `grep` on our local computer! `ssh` is magical, and we will talk more about it in the next lecture on the command-line environment. This is still way more stuff than we wanted though. And pretty hard to read. Let's do better:

```
ssh myserver 'journalctl | grep sshd | grep "Disconnected from"' | less
```

Why the additional quoting? Well, our logs may be quite large, and it's wasteful to stream it all to our computer and then do the filtering. Instead, we can do the filtering on the remote server, and then massage the data locally. `less` gives us a "pager" that allows us to scroll up and down through the long output. To save some additional traffic while we debug our command-line, we can even stick the current filtered logs into a file so that we don't have to access the network while developing:

```
$ ssh myserver 'journalctl | grep sshd | grep "Disconnected from"' > ssh.log
$ less ssh.log
```

There's still a lot of noise here. There are *a lot* of ways to get rid of that, but let's look at one of the most powerful tools in your toolkit: `sed`.

`sed` is a "stream editor" that builds on top of the old `ed` editor. In it, you basically give short commands for how to modify the file, rather than manipulate its contents directly (although you can do that too). There are tons of commands, but one of the most common ones is `s`: substitution. For example, we can write:

```
ssh myserver journalctl
 | grep sshd
 | grep "Disconnected from"
 | sed 's/.*Disconnected from //'
```

What we just wrote was a simple *regular expression*; a powerful construct that lets you match text against patterns. The `s` command is written on the form: `s/REGEX/SUBSTITUTION/`, where `REGEX` is the regular expression you want to search for, and `SUBSTITUTION` is the text you want to substitute matching text with.

(You may recognize this syntax from the "Search and replace" section of our Vim lecture notes! Indeed, Vim uses a syntax for searching and replacing that is similar to `sed`'s substitution command. Learning one tool often helps you become more proficient with others.)

Regular expressions

Regular expressions are common and useful enough that it's worthwhile to take some time to understand how they work. Let's start by looking at the one we used above: `/.*Disconnected from /`. Regular expressions are usually (though not always) surrounded by `/`. Most ASCII characters just carry their normal meaning, but some characters have "special" matching behavior. Exactly which characters do what vary somewhat between different implementations of regular expressions, which is a source of great frustration. Very common patterns are:

- `.` means "any single character" except newline
- `*` zero or more of the preceding match
- `+` one or more of the preceding match
- `[abc]` any one character of `a`, `b`, and `c`
- `(RX1|RX2)` either something that matches `RX1` or `RX2`
- `^` the start of the line
- `$` the end of the line

`sed`'s regular expressions are somewhat weird, and will require you to put a `\` before most of these to give them their special meaning. Or you can pass `-E`.

So, looking back at `/.*Disconnected from /`, we see that it matches any text that starts with any number of characters, followed by the literal string "Disconnected from ". Which is what we wanted. But

beware, regular expressions are trixy. What if someone tried to log in with the username "Disconnected from"? We'd have:

```
Jan 17 03:13:00 thesquareplanet.com sshd[2631]: Disconnected from invalid user
Disconnected from 46.97.239.16 port 55920 [preauth]
```

What would we end up with? Well, `*` and `+` are, by default, "greedy". They will match as much text as they can. So, in the above, we'd end up with just

```
46.97.239.16 port 55920 [preauth]
```

Which may not be what we wanted. In some regular expression implementations, you can just suffix `*` or `+` with a `?` to make them non-greedy, but sadly `sed` doesn't support that. We *could* switch to perl's command-line mode though, which *does* support that construct:

```
perl -pe 's/.*?Disconnected from //'
```

We'll stick to `sed` for the rest of this, because it's by far the more common tool for these kinds of jobs. `sed` can also do other handy things like print lines following a given match, do multiple substitutions per invocation, search for things, etc. But we won't cover that too much here. `sed` is basically an entire topic in and of itself, but there are often better tools.

Okay, so we also have a suffix we'd like to get rid of. How might we do that? It's a little tricky to match just the text that follows the username, especially if the username can have spaces and such! What we need to do is match the *whole* line:

```
 | sed -E 's/.*Disconnected from (invalid |authenticating )?user .* [^ ]+ port [0-
9]+( \[preauth\])?$//'
```

Let's look at what's going on with a regex debugger. Okay, so the start is still as before. Then, we're matching any of the "user" variants (there are two prefixes in the logs). Then we're matching on any string of characters where the username is. Then we're matching on any single word (`[^ ]+` ; any non-empty sequence of non-space characters). Then the word "port" followed by a sequence of digits. Then possibly the suffix `[preauth]` , and then the end of the line.

Notice that with this technique, as username of "Disconnected from" won't confuse us any more. Can you see why?

There is one problem with this though, and that is that the entire log becomes empty. We want to *keep* the username after all. For this, we can use "capture groups". Any text matched by a regex surrounded by parentheses is stored in a numbered capture group. These are available in the substitution (and in some engines, even in the pattern itself!) as `\1` , `\2` , `\3` , etc. So:

```
 | sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^ ]+ port
[0-9]+( \[preauth\])?$/\2/'
```

As you can probably imagine, you can come up with *really* complicated regular expressions. For example, here's an article on how you might match an e-mail address. It's not easy. And there's lots of discussion. And people have written tests. And test matrices. You can even write a regex for determining if a given number is a prime number.

Regular expressions are notoriously hard to get right, but they are also very handy to have in your toolbox!

Back to data wrangling

Okay, so we now have

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

`sed` can do all sorts of other interesting things, like injecting text (with the `i` command), explicitly printing lines (with the `p` command), selecting lines by index, and lots of other things. Check `man sed`!

Anyway. What we have now gives us a list of all the usernames that have attempted to log in. But this is pretty unhelpful. Let's look for common ones:

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

`sort` will, well, sort its input. `uniq -c` will collapse consecutive lines that are the same into a single line, prefixed with a count of the number of occurrences. We probably want to sort that too and only keep the most common usernames:

```
ssh myserver journalctl
 | grep sshd
 | grep "Disconnected from"
 | sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^ ]+ port
[0-9]+( \[preauth\])?$/\2/'
 | sort | uniq -c
 | sort -nk1,1 | tail -n10
```

`sort -n` will sort in numeric (instead of lexicographic) order. `-k1,1` means "sort by only the first whitespace-separated column". The `,n` part says "sort until the `n` th field, where the default is the end of the line. In this *particular* example, sorting by the whole line wouldn't matter, but we're here to learn!

If we wanted the *least* common ones, we could use `head` instead of `tail`. There's also `sort -r`, which sorts in reverse order.

Okay, so that's pretty cool, but what if we'd like these extract only the usernames as a comma-separated list instead of one per line, perhaps for a config file?

```
ssh myserver journalctl
 | grep sshd
 | grep "Disconnected from"
 | sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^ ]+ port
[0-9]+( \[preauth\])?$/\2/'
 | sort | uniq -c
 | sort -nk1,1 | tail -n10
 | awk '{print $2}' | paste -sd,
```

If you're using macOS: note that the command as shown won't work with the BSD `paste` shipped with macOS. See exercise 4 from the shell tools lecture for more on the difference between BSD and GNU coreutils and instructions for how to install GNU coreutils on macOS.

Let's start with `paste`: it lets you combine lines (`-s`) by a given single-character delimiter (`-d`; `,` in this case). But what's this `awk` business?

awk – another editor

`awk` is a programming language that just happens to be really good at processing text streams. There is *a lot* to say about `awk` if you were to learn it properly, but as with many other things here, we'll just go through the basics.

First, what does `{print $2}` do? Well, `awk` programs take the form of an optional pattern plus a block saying what to do if the pattern matches a given line. The default pattern (which we used above) matches all lines. Inside the block, `$0` is set to the entire line's contents, and `$1` through `$n` are set to the `n` th *field* of that line, when separated by the `awk` field separator (whitespace by default, change with `-F`). In this case, we're saying that, for every line, print the contents of the second field, which happens to be the username!

Let's see if we can do something fancier. Let's compute the number of single-use usernames that start with `c` and end with `e`:

```
 | awk '$1 == 1 && $2 ~ /^c[^ ]*e$/ { print $2 }' | wc -l
```

There's a lot to unpack here. First, notice that we now have a pattern (the stuff that goes before `{...}`). The pattern says that the first field of the line should be equal to 1 (that's the count from `uniq -c`), and that the second field should match the given regular expression. And the block just says to print the username. We then count the number of lines in the output with `wc -l`.

However, `awk` is a programming language, remember?

```
BEGIN { rows = 0 }
$1 == 1 && $2 ~ /^c[^ ]*e$/ { rows += $1 }
END { print rows }
```

`BEGIN` is a pattern that matches the start of the input (and `END` matches the end). Now, the per-line block just adds the count from the first field (although it'll always be 1 in this case), and then we print it out at the end. In fact, we *could* get rid of `grep` and `sed` entirely, because `awk` can do it all, but we'll leave that as an exercise to the reader.

Analyzing data

You can do math directly in your shell using `bc`, a calculator that can read from STDIN! For example, add the numbers on each line together by concatenating them together, delimited by `+`:

```
 | paste -sd+ | bc -l
```

Or produce more elaborate expressions:

```
echo "2*($(data | paste -sd+))" | bc -l
```

You can get stats in a variety of ways. st is pretty neat, but if you already have R:

```
ssh myserver journalctl
 | grep sshd
 | grep "Disconnected from"
 | sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^ ]+ port
[0-9]+( \[preauth\])?$/\2/'
 | sort | uniq -c
 | awk '{print $1}' | R --no-echo -e 'x <- scan(file="stdin", quiet=TRUE);
summary(x)'
```

R is another (weird) programming language that's great at data analysis and plotting. We won't go into too much detail, but suffice to say that `summary` prints summary statistics for a vector, and we created a vector containing the input stream of numbers, so R gives us the statistics we wanted!

If you just want some simple plotting, `gnuplot` is your friend:

```
ssh myserver journalctl
 | grep sshd
 | grep "Disconnected from"
 | sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^ ]+ port
[0-9]+( \[preauth\])?$/\2/'
 | sort | uniq -c
 | sort -nk1,1 | tail -n10
 | gnuplot -p -e 'set boxwidth 0.5; plot "-" using 1:xtic(2) with boxes'
```

Data wrangling to make arguments

Sometimes you want to do data wrangling to find things to install or remove based on some longer list. The data wrangling we've talked about so far + `xargs` can be a powerful combo.

For example, as seen in lecture, I can use the following command to uninstall old nightly builds of Rust from my system by extracting the old build names using data wrangling tools and then passing them via `xargs` to the uninstaller:

```
rustup toolchain list | grep nightly | grep -vE "nightly-x86" | sed 's/-x86.*//' |
xargs rustup toolchain uninstall
```

Wrangling binary data

So far, we have mostly talked about wrangling textual data, but pipes are just as useful for binary data. For example, we can use ffmpeg to capture an image from our camera, convert it to grayscale, compress it, send it to a remote machine over SSH, decompress it there, make a copy, and then display it.

```
ffmpeg -loglevel panic -i /dev/video0 -frames 1 -f image2 -
 | convert - -colorspace gray -
 | gzip
 | ssh mymachine 'gzip -d | tee copy.jpg | env DISPLAY=:0 feh -'
```

Exercises

1. Take this short interactive regex tutorial.
2. Find the number of words (in `/usr/share/dict/words`) that contain at least three `a`s and don't have a `'s` ending. What are the three most common last two letters of those words? `sed`'s `y` command, or the `tr` program, may help you with case insensitivity. How many of those two-letter combinations are there? And for a challenge: which combinations do not occur?
3. To do in-place substitution it is quite tempting to do something like `sed s/REGEX/SUBSTITUTION/ input.txt > input.txt`. However this is a bad idea, why? Is this particular to `sed`? Use `man sed` to find out how to accomplish this.

4. Find your average, median, and max system boot time over the last ten boots. Use `journalctl` on Linux and `log show` on macOS, and look for log timestamps near the beginning and end of each boot. On Linux, they may look something like:

   ```
   Logs begin at ...
   ```

   and

   ```
   systemd[577]: Startup finished in ...
   ```

   On macOS, look for:

   ```
   === system boot:
   ```

   and

   ```
   Previous shutdown cause: 5
   ```

5. Look for boot messages that are *not* shared between your past three reboots (see `journalctl`'s `-b` flag). Break this task down into multiple steps. First, find a way to get just the logs from the past three boots. There may be an applicable flag on the tool you use to extract the boot logs, or you can use `sed '0,/STRING/d'` to remove all lines previous to one that matches `STRING`. Next, remove any parts of the line that *always* varies (like the timestamp). Then, de-duplicate the input lines and keep a count of each one (`uniq` is your friend). And finally, eliminate any line whose count is 3 (since it *was* shared among all the boots).
6. Find an online data set like this one, this one, or maybe one from here. Fetch it using `curl` and extract out just two columns of numerical data. If you're fetching HTML data, pup might be helpful. For JSON data, try `jq`. Find the min and max of one column in a single command, and the difference of the sum of each column in another.
