Data Wrangling 数据整理

您是否曾经有过这样的需求，将某种格式存储的数据转换成另外一种格式? 肯定有过，对吧！ 这也正是我们这节课所要讲授的主要内容。具体来讲，我们需要不断地对数据进行处理，直到得到我们想要的最终结果。

在之前的课程中，其实我们已经接触到了一些数据整理的基本技术。可以这么说，每当您使用管道运算符的时候，其实就是在进行某种形式的数据整理。

例如这样一条命令 `journalctl | grep -i intel`，它会找到所有包含intel(不区分大小写)的系统日志。您可能并不认为这是数据整理，但是它确实将某种形式的数据（全部系统日志）转换成了另外一种形式的数据（仅包含intel的日志）。大多数情况下，数据整理需要您能够明确哪些工具可以被用来达成特定数据整理的目的，并且明白如何组合使用这些工具。

让我们从头讲起。既然是学习数据整理，那有两样东西自然是必不可少的：用来**整理的数据**以及**相关的应用场景**。**日志处理**通常是一个比较典型的使用场景，因为我们经常需要在日志中查找某些信息，这种情况下通读日志是不现实的。现在，让我们研究一下系统日志，看看哪些用户曾经尝试过登录我们的服务器：

```
1 ssh myserver journalctl
```

内容太多了。现在让我们把涉及 sshd 的信息过滤出来：

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

注意，这里我们使用管道将一个远程服务器上的文件传递给本机的 `grep` 程序！ `ssh` 太牛了，下一节课我们会讲授命令行环境，届时我们会详细讨论 `ssh` 的相关内容。此时我们打印出的内容，仍然比我们需要的要多得多，读起来也非常费劲。我们来改进一下：

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

多出来的引号是什么作用呢？这么说吧，我们的日志是一个非常大的文件，把这么大的文件流直接传输到我们本地的电脑上再进行过滤是对流量的一种浪费。因此我们采取另外一种方式，我们先在远端机器上过滤文本内容，然后再将结果传输到本机。 `less` 为我们创建来一个**文件分页器（pager）**，使我们可以通过翻页的方式浏览较长的文本。为了进一步节省流量，我们甚至可以将当前过滤出的日志保存到文件中，这样后续就不需要再次通过网络访问该文件了：

```
1 $ ssh myserver 'journalctl | grep sshd | grep "Disconnected from"' > ssh.log
  $ less ssh.log
2
```

过滤结果中仍然包含不少没用的数据。我们有很多办法可以删除这些无用的数据，但是让我们先研究一下 `sed` 这个非常强大的工具。

`sed` 是一个基于文本编辑器 `ed` 构建的”流编辑器”。在 `sed` 中，您基本上是利用一些简短的命令来**修改**文件，而不是直接操作文件的内容（尽管您也可以选择这样做）。相关的命令行非常多，但是最常用的是 `s`，即*替换*命令，例如我们可以这样写：

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

上面这段命令中，我们使用了一段简单的*正则表达式*。正则表达式是一种非常强大的工具，可以让我们基于某种模式来对字符串进行匹配。`s` 命令的语法如下：`s/REGEX/SUBSTITUTION/`，其中 `REGEX` 部分是我们需要使用的正则表达式，而 `SUBSTITUTION` 是用于替换匹配结果的文本。

正则表达式

正则表达式非常常见也非常有用，值得您花些时间去理解它。让我们从这一句正则表达式开始学习：`/.*Disconnected from /`。正则表达式通常以（尽管并不总是）`/` 开始和结束。大多数的 ASCII 字符都表示它们本来的含义，但是有一些字符确实具有表示匹配行为的“特殊”含义。不同字符所表示的含义，根据正则表达式的实现方式不同，也会有所变化，这一点确实令人沮丧。常见的模式有：

- `.` 除换行符之外的”任意单个字符”
- `*` 匹配前面字符零次或多次
  - `(ab|bc)*`
- `+` 匹配前面字符一次或多次
- `[abc]` 匹配 `a`，`b` 和 `c` 中的**任意**一个
  - 只会匹配一次
  - `echo 'bba' | sed 's/[ab]//'`
    - 输出 `ba`
  - `echo 'bba' | sed 's/[ab]//g'`
    - 不输出
    - 加g就可以匹配多次
- `(RX1|RX2)` 任何能够匹配 `RX1` 或 `RX2` 的结果
- `^` 行首
- `$` 行尾

`sed` 的正则表达式有些时候是比较奇怪的，它需要你在这些模式前添加 `\` 才能使其具有特殊含义。或者，您也可以添加 `-E` 选项来支持这些匹配。

回过头我们再看 `/.*Disconnected from /`，我们会发现这个正则表达式可以匹配任何以若干任意字符开头，并接着包含”Disconnected from “的字符串。这也正是我们所希望的。但是请注意，正则表达式并不容易写对。如果有人将 “Disconnected from” 作为自己的用户名会怎样呢？

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

正则表达式会如何匹配？ `*` 和 `+` 在默认情况下是贪婪模式，也就是说，它们会尽可能多的匹配文本。因此对上述字符串的匹配结果如下：

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

这可不是我们想要的结果。对于某些正则表达式的实现来说，您可以给 `*` 或 `+` 增加一个 `?` 后缀使其变成非贪婪模式，但是很可惜 `sed` 并不支持该后缀。不过，我们可以切换到 perl 的命令行模式，该模式支持编写这样的正则表达式：

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

让我们回到 `sed` 命令并使用它完成后续的任务，毕竟对于这一类任务， `sed` 是最常见的工具。 `sed` 还可以非常方便的做一些事情，例如打印匹配后的内容，一次调用中进行多次替换搜索等。但是这些内容我们并不会在此进行介绍。 `sed` 本身是一个非常全能的工具，但是在具体功能上往往能找到更好的工具作为替代品。

好的，我们还需要去掉用户名后面的后缀，应该如何操作呢？

想要匹配用户名后面的文本，尤其是当这里的用户名可以包含空格时，这个问题变得非常棘手！这里我们需要做的是匹配*一整行*：

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

让我们借助正则表达式在线调试工具regex debugger 来理解这段表达式。OK，开始的部分和以前是一样的，随后，我们匹配两种类型的“user”（在日志中基于两种前缀区分）。再然后我们匹配属于用户名的所有字符。接着，再匹配任意一个单词（ `[^ ]+` 会匹配任意非空且不包含空格的序列）。紧接着后面匹配单“port”和它后面的一串数字，以及可能存在的后缀 `[preauth]` ，最后再匹配行尾。

注意，这样做的话，即使用户名是“Disconnected from”，对匹配结果也不会有任何影响，您知道这是为什么吗？

问题还没有完全解决，日志的内容全部被替换成了空字符串，整个日志的内容因此都被删除了。我们实际上希望能够将用户名*保留*下来。对此，我们可以使用“捕获组（capture groups）”来完成。被圆括号内的正则表达式匹配到的文本，都会被存入一系列以编号区分的捕获组中。捕获组的内容可以在替换字符串时使用（有些正则表达式的引擎甚至支持替换表达式本身），例如 `\1` 、 `\2` 、 `\3` 等等，因此可以使用如下命令：

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

想必您已经意识到了，为了完成某种匹配，我们最终可能会写出非常复杂的正则表达式。例如，这里有一篇关于如何匹配电子邮箱地址的文章e-mail address，匹配电子邮箱可一点也不简单。网络上还有很多关于如何匹配电子邮箱地址的讨论。人们还为其编写了测试用例及 测试矩阵。您甚至可以编写一个用于判断一个数是否为质数的正则表达式。

正则表达式是出了名的难以写对，但是它仍然会是您强大的常备工具之一。

回到数据整理

OK，现在我们有如下表达式：

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

`sed` 还可以做很多各种各样有趣的事情，例如文本注入：(使用 `i` 命令)，打印特定的行 (使用 `p` 命令)，基于索引选择特定行等等。详情请见 `man sed`！

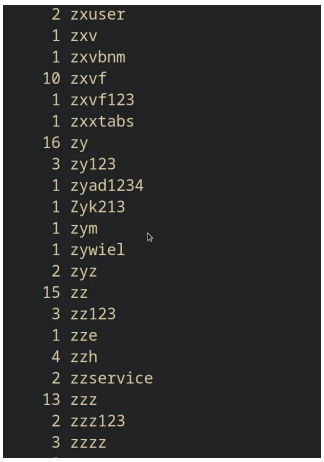
现在，我们已经得到了一个包含用户名的列表，列表中的用户都曾经尝试过登陆我们的系统。但这还不够，让我们过滤出那些最常出现的用户：

```
wc -l # 显示行数，words counting line
```

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


```

`sort` 会对其输入数据进行排序，但是还是太多了。`uniq -c` 会把连续出现的**行折叠为一行并使用出现次数作为前缀**。我们希望按照出现次数排序，过滤出最常出现的用户名：




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

`sort -n` 会按照数字顺序对输入进行排序（默认情况下是按照字典序排序 `-k1,1` 则表示“仅基于以空格分割的第一列进行排序”。 `,n` 部分表示“仅排序到第n个部分”，默认情况是到行尾。就本例来说，针对整个行进行排序也没有任何问题，我们这里主要是为了学习这一用法！

如果我们希望得到登陆次数最少的用户，我们可以使用 `head` 来代替 `tail`。或者使用 `sort -r` 来进行倒序排序。

相当不错。但我们只想获取用户名，而且不要一行一个地显示。

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

如果您使用的是 MacOS：注意这个命令并不能配合 MacOS 系统默认的 BSD `paste` 使用。参考课程概览与shell的习题内容获取更多相关信息。

我们可以利用 `paste` 命令来合并行(`-s`)，并指定一个分隔符进行分割 (`-d`)，那 `awk` 的作用又是什么呢?

awk – 另外一种编辑器

`awk` 其实是一种编程语言，只不过它碰巧非常善于处理文本。关于 `awk` 可以介绍的内容太多了，限于篇幅，这里我们仅介绍一些基础知识。

首先， `{print $2}` 的作用是什么？ `awk` 程序接受一个模式串（可选），以及一个代码块，指定当模式匹配时应该做何种操作。默认当模式串即匹配所有行（上面命令中当用法）。 在代码块中， `$0` 表示整行的内容， `$1` 到 `$n` 为一行中的 n 个区域，区域的分割基于 `awk` 的域分隔符（默认是空格，可以通过 `-F` 来修改）。在这个例子中，我们的代码意思是：对于每一行文本，打印其第二个部分，也就是用户名。

让我们康康，还有什么炫酷的操作可以做。让我们统计一下所有以 `c` 开头，以 `e` 结尾，并且仅尝试过一次登陆的用户。

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

让我们好好分析一下。首先，注意这次我们为 `awk` 指定了一个匹配模式串（也就是 `{...}` 前面的那部分内容）。该匹配要求文本的第一部分需要等于1（这部分刚好是 `uniq -c` 得到的计数值），然后其第二部分必须满足给定的一个正则表达式。代码块中的内容则表示打印用户名。然后我们使用 `wc -l` 统计输出结果的行数。

不过，既然 `awk` 是一种编程语言，那么则可以这样：

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

`BEGIN` 也是一种模式，它会匹配输入的开头（ `END` 则匹配结尾）。然后，对每一行第一个部分进行累加，最后将结果输出。事实上，我们完全可以抛弃 `grep` 和 `sed` ，因为 `awk` 就可以解决所有问题。至于怎么做，就留给读者们做课后练习吧。

分析数据

想做数学计算也是可以的！例如这样，您可以将每行的数字加起来：

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

下面这种更加复杂的表达式也可以：

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

您可以通过多种方式获取统计数据。如果已经安装了R语言， `st` 是个不错的选择：

```
ssh myserver journalctl
  | grep sshd
  | grep "Disconnected from"
  | sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^ ]+ port [0-9]+( \[
  | sort | uniq -c
  | awk '{print $1}' | R --slave -e 'x <- scan(file="stdin", quiet=TRUE); summary(x)'
```

R 也是一种编程语言，它非常适合被用来进行数据分析和绘制图表。这里我们不会讲的特别详细， 您只需要知道 `summary` 可以打印某个向量的统计结果。我们将输入的一系列数据存放在一个向量后，利用R语言就可以得到我们想要的统计数据。

如果您希望绘制一些简单的图表， `gnuplot` 可以帮助到您：

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

利用数据整理来确定参数

有时候您要利用数据整理技术从一长串列表里找出你所需要安装或移除的东西。我们之前讨论的相关技术配合 `xargs` 即可实现：

```
rustup toolchain list | grep nightly | grep -v ‘nightly-x86' | grep 2019
rustup toolchain list | grep nightly | grep -vE "nightly-x86" | sed 's/-x86.*//' | xargs ru
```

整理二进制数据

虽然到目前为止我们的讨论都是基于文本数据，但对于二进制文件其实同样有用。例如我们可以用 ffmpeg 从相机中捕获一张图片，将其转换成灰度图后通过SSH将压缩后的文件发送到远端服务器，并在那里解压、存档并显示。

```
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 -'
```

课后练习

习题解答

1. 学习一下这篇简短的 交互式正则表达式教程.
2. 统计words文件 (`/usr/share/dict/words`) 中包含至少三个 `a` 且不以 `'s` 结尾的单词个数。这些单词中，出现频率前三的**末尾两个字母**是什么？ `sed` 的 `y` 命令，或者 `tr` 程序也许可以帮你解决大小写的问题。共存在多少种词尾两字母组合？还有一个很 有挑战性的问题：哪个组合从未出现过？

```
> cat /usr/share/dict/words | tr "[:upper:]" "[:lower:]" | grep -E "^([^a]*a){3}.*$" | grep -v "'s$" | wc -l
    7595
!w /
```

```
> cat /usr/share/dict/words | tr "[:upper:]" "[:lower:]" | grep -E "^([^a]*a){3}.*$" | grep -v "'s$" | sed -E "s/.*([a-z]{2})$/\1/" | sort | uniq -c | sort | tail -n3
 763 an
 814 ia
1039 al
```

```
> cat /usr/share/dict/words | tr "[:upper:]" "[:lower:]" | grep -E "^([^a]*a){3}.*$" | grep -v "'s$" | sed -E "s/.*([a-z]{2})$/\1/" | sort | uniq | wc -l
     156
```

注意修改all.sh文档的权限

```
> ls -alh
total 16
drwxr-xr-x  5 fangchengrui  staff   160B  7 19 11:54 .
drwxr-xr-x  7 fangchengrui  staff   224B  7  6 23:28 ..
-rw-r--r--  1 fangchengrui  staff    79B  7 19 11:53 all.sh
-rw-r--r--  1 fangchengrui  staff     0B  7 19 11:56 all.txt
-rw-r--r--  1 fangchengrui  staff   468B  7 19 11:54 occurance.txt
> sudo chmod -R 711 all.sh
> ls -alh
total 16
drwxr-xr-x  5 fangchengrui  staff   160B  7 19 11:54 .
drwxr-xr-x  7 fangchengrui  staff   224B  7  6 23:28 ..
-rwx--x--x  1 fangchengrui  staff    79B  7 19 11:53 all.sh
-rw-r--r--  1 fangchengrui  staff     0B  7 19 11:56 all.txt
-rw-r--r--  1 fangchengrui  staff   468B  7 19 11:54 occurance.txt
> sudo ./all.sh > all.txt
> bat all.txt
>
> diff --unchanged-group-format='' <(cat occurance.txt) <(cat all.txt) | wc -l
     520
```

3. 进行原地替换听上去很有诱惑力，例如： `sed s/REGEX/SUBSTITUTION/ input.txt > input.txt` 。但是这并不是一个明智的做法，为什么呢？还是说只有 `sed` 是这样的？查看 `man sed` 来完成这个问题

`sed s/REGEX/SUBSTITUTION/ input.txt > input.txt` 表达式中后一个 `input.txt` 会首先被清空，而且是发生在前的。所以前面一个 `input.txt` 在还没有被 `sed` 处理时已经为空了。在使用正则处理文件前最好是首先备份文件。

```
sed -i.bak s/REGEX/SUBSTITUTION/ input.txt > input.txt
```

可以自动创建一个后缀为 `.bak` 的备份文件。

4. 找出您最近十次开机的开机时间平均数、中位数和最长时间。在Linux上需要用到 `journalctl` ，而在macOS 上使用 `log show` 。找到每次起到开始和结束时的时间戳。在Linux上类似这样操作：

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

和

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

在 macOS 上, 查找:

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

和

```
Previous shutdown cause: 5
```

查看之前三次重启启动信息中不同的部分(参见 `journalctl` 的 `-b` 选项)。将这一任务分为几个步骤，首先获取之前三次启动的启动日志，也许获取启动日志的命令就有合适的选项可以帮助您提取前三次启动的日志，亦或者您可以使用 `sed '0,/STRING/d'` 来删除 `STRING` 匹配到的字符串前面的全部内容。然后，过滤掉每次都不相同的部分，例如时间戳。下一步，重复记录输入行并对其计数(可以使用 `uniq`)。最后，删除所有出现过3次的内容（因为这些内容是三次启动日志中的重复部分）。

在网上找一个类似 这个 或者这个的数据集。或者从这里找一些。使用 `curl` 获取数据集并提取其中两列数据，如果您想要获取的是HTML数据，那么 `pup` 可能会更有帮助。对于JSON类型的数据，可以试试 `jq` 。请使用一条指令来找出其中一列的最大值和最小值，用另外一条指令计算两列之间差的总和。