Project 6: Data Manipulation With Unix Shell Commands

Overview

We often need to quickly analyze and process large data in a text-based format, such as comma-separated values (CSV). Most commonly, CSV-type data is handled using a standard spreadsheet application, such as Microsoft Excel. Spreadsheet applications, however, are not designed to deal with data larger than a machine's main memory. Opening a file larger than a few gigabytes leads to many frustrating and wasted hours, frozen screens, and application crashes.

There is a savior to this familiar ordeal: many basic data-processing tasks can be conducted with a few lines of Unix shell commands. In fact, as you will see later, most "relational operators" that we learned in the class are available as Unix shell commands, so it is possible to write complex "SQL queries" on text-based files using Unix shell commands! Familiarity with Unix commands, therefore, can be a powerful skill that can save your time and frustration down the road. In this project, we will learn popular Unix text-based shell commands that are useful for basic data processing and analysis tasks.

Development Environment

The main development of Project 6 will be done using a new docker container named "unix" created from "junghoo/unix". Use the following command to create and start this new container:

```
$ docker run -it -v {your_shared_dir}:/home/cs143/shared --name unix junghoo/unix
```

Make sure to replace `{your_shared_dir}` with the name of the shared directory on your host. The above command creates a docker container named `unix` with appropriate directory sharing.

As before, the default username inside the container is "cs143" with password "password". Once set up, the container can be restarted any time using the following command:

```
$ docker start -i unix
```

Part A: Learn Unix Data-Manipulation Commands

In the first part, you will learn key Unix shell commands that can be useful for processing data in a text file by going over a few simple examples. The examples in this part are based on the adult dataset from the UCI Machine Learning repository. Please first download the dataset via the `wget` command:

```
$ wget https://oak.cs.ucla.edu/classes/cs143/project6/adult.data
```

This dataset is commonly used to predict whether income exceeds $50K/yr based on census data. With ~50K rows, it is not a large dataset but will be perfect as an example. Note that the data has the extension `.data` but it is a well-formatted CSV file.

Looking at a large file: `cat` and `less`

The well-known `cat` command prints out the content of a text file. For example, the following command will print out all contents of the `adult.data` file on the screen:

```
$ cat adult.data
```

Unfortunately, this command is not very helpful when the file is large since pages after pages of data instantly scroll past on the screen. To investigate the data in a large file one page at a time, you can use the `less` command like the following:

```
$ less adult.data
```

This will show the content of the file one screen at a time, letting you go to the next page by pressing the space bar and go back to a previous page by pressing the key `b`.

Subset a large file: `head`, `tail` and `shuf`

You can look at just the first few lines of a file with the `head` command. By default, `head` outputs 10 lines, but you can adjust this count with the `-n <line count>` flag.

```
$ head -n 2 adult.data
39, State-gov, 77516, Bachelors, 13, Never-married, Adm-clerical, Not-in-family, Whit
50, Self-emp-not-inc, 83311, Bachelors, 13, Married-civ-spouse, Exec-managerial, Husb
```

The command `tail` is very similar to `head` except that it outputs the *last* 10 lines of an input file, which can be adjusted using the `-n` flag:

```
$ tail -n 4 adult.data
40, Private, 154374, HS-grad, 9, Married-civ-spouse, Machine-op-inspct, Husband, Whit
58, Private, 151910, HS-grad, 9, Widowed, Adm-clerical, Unmarried, White, Female, 0,
22, Private, 201490, HS-grad, 9, Never-married, Adm-clerical, Own-child, White, Male,
52, Self-emp-inc, 287927, HS-grad, 9, Married-civ-spouse, Exec-managerial, Wife, Whit
```

Using the output of a command as input to another command using the pipe symbol `|` is a useful shell pattern called pipelining. You can extract an arbitrary subset of a file by "piping together" the `head` and `tail` commands. For example, you can extract the lines 101 through 120 of `adult.data` and save it to `adult_sample.csv` using the following command:

```
$ head -n 120 adult.data | tail -n 20 > adult_sample.csv
```

Note that the first `head` command keeps lines 1 through 120 (`-n 120`) of the input. The second `tail` command takes the output from `head` as its input and keeps only the last 20 lines (`-n 20`). Combined together, this sequence of two "piped commands" keeps the lines 101 through 120 from the original input. The last part, `> adult_sample.csv` "redirects the output" to a file named `adult_sample.csv` and saves it.

Both `head` and `tail` support `-<line count>` as a shorthand of the `-n <line count>` flag. So the above command is equivalent to:

```
$ head -120 adult.data | tail -20 > adult_sample.csv
```

Sometimes, it may be useful to take a random sample from a large file. The `shuf` command, which randomly shuffles the rows in a file, can be used for this purpose. For example, the following command selects 10 random lines from `adult.data`

```
$ shuf adult.data | head -10
```

More Useful Tip: `tail` also takes a `-f` flag where `f` stands for `--follow`. When this flag is specified, the command stays open, monitors any newly-appended lines to the file, and continuously outputs them. This flag is particularly convenient to monitor log files like `tail -f <logfile>`.

Count with `wc`

You can count the number of lines and words in a file using the `wc` command:

```
$ wc adult.data
  32561  488415 3974304 adult.data
```

Here, 32561 is the number of lines in the file. 488415 is the number of words. 3974304 is the number of characters.

The `wc` command comes in handy when you want to collect quick statistics. For example, you can count the number of files in a directory by using the output of the `ls` command as the input to `wc`:

```
$ ls -l <directory> | wc
      7      56     374
```

From the result, we see that the output of `ls -l` has 7 lines, which means that the directory has 6 files (since the first output line of `ls -l` is something other than a filename).

Project operator: `cut`

Unix shell also has the `cut` command that is equivalent to the "project operator" in the relational model. `cut` prints out a (subset of) column(s) in a file and takes two main flags: `-d` to specify the column delimiter and `-f` to specify the columns you want to output. For example, the following command will print the 2nd and 5th columns of the first 10 lines of `adult.data`:

```
$ cut -d , -f 2,5 adult.data | head -10
 State-gov, 13
 Self-emp-not-inc, 13
 Private, 9
 Private, 7
 Private, 13
```

Note that there is no space in "2,5" in the above command. Otherwise, anything after the space will not be considered as part of the `-f` flag.

Select operator: `grep` and `awk`

We can use `grep` and `awk` Unix commands like the "select operator" of the relational model.

`grep` outputs only the lines that match a given regular expression. For example, the following command will print only the lines that contain the word "Jamaica":

```
$ grep Jamaica adult.data
49, Private, 160187, 9th, 5, Married-spouse-absent, Other-service, Not-in-family, Bla
40, Private, 229148, 12th, 8, Married-civ-spouse, Other-service, Husband, Black, Male
31, Private, 184307, Some-college, 10, Married-civ-spouse, Exec-managerial, Husband,
...
```

In place of Jamaica, you can use arbitrary regular expressions like 'Ja.*ca'. Popular options of `grep` include `-i` that performs case-insensitive matches, `-c` that prints out just the count of matching lines, and `-v` that "inverts" the matching logic and prints out only the *non-matching lines*. `grep` is a tool that is very frequently used, so it is worthwhile to get familiar with it. To learn more, see this page for more useful details.

While useful, `grep` does not have the notion of "column" and it is not easy to select a line based on a particular column value. For that, you can use `awk`. For example, the following command will output only the lines whose 3rd column value is larger than 1.2 million.

```
$ awk -F , '$3 > 1200000' adult.data
```

Here, `-F` flag specifies the field delimiter. Note that the main "select condition" appears inside a *single quote*. You must use single quotes here to prevent automatic shell variable expansion.

It is possible to select lines based on more complicated conditions. For example, the following command will print only the lines whose third column is larger than 1.2 million as long as its 10th column is "Female" or 9th column is not "White".

```
$ awk -F ,  '($3 > 1200000) && (($10 = "Female") || ($9 != "White"))' adult.data
```

The single equal sign `=` is the equality operator and `!=` is the inequality operator.

While we explained the use of `awk` as a column-based select operator, awk in fact isn't just a simple "command," but is a full programming language. Mastery of the `awk` programming language will give you a powerful tool to manipulate a text file in sophisticated ways. Read the The GNU Awk User's Guide if you want to learn more.

Sorting data

For many data manipulation tasks, "sorting" the data by a particular (set of) column(s) is important. You can use the `sort` command for this purpose with `-t <field delimiter>` and `-k <sort_field_start>,<sort_field_end>` flags. For example, the following command sort all lines in `adult.data` by the third column and prints the last 5 lines:

```
$ sort -t , -k 3,3 adult.data | tail -5
31, Private, 99928, Masters, 14, Married-civ-spouse, Prof-specialty, Wife, White, Fem
37, Local-gov, 99935, Masters, 14, Married-civ-spouse, Protective-serv, Husband, Whit
24, Private, 99970, Bachelors, 13, Never-married, Tech-support, Own-child, White, Mal
45, Private, 99971, HS-grad, 9, Married-civ-spouse, Transport-moving, Husband, White,
51, Private, 99987, 10th, 6, Separated, Machine-op-inspct, Unmarried, Black, Female,
```

By default, the sort order is lexicographical (*not numeric*) and ascending. If you want to change from this default, add the suffix `n` to the `-k` flag to sort numerically and `r` to sort in the reverse (or descending) order, like the following example.

```
$ sort -t , -k 3,3rn adult.data | tail -5
```

You can sort by multiple columns by adding more `-k` flags to the command. For example, if you have `-k1,1n -k 3,3r`, it will sort the input first by the first column numerically and then by the third column lexicographically in the reverse order in case two rows have the same first column values.

```
$ sort -t , -k 1,1n -k 3,3r adult.data | head -5
```

Note: To handle files larger than the main memory size, `sort` uses the disk-based merge-sort algorithm. During a sort, it stores "intermediate sorted runs" in the `/tmp/` directory. You need to have at least twice as much space as the original file in the `/tmp/` directory.

Finding duplicates with `uniq`

uniq modifies the input by collapsing identical *consecutive* lines into a single copy. On its own, this may not seem too interesting, but when used to build pipelines, this can be very useful. For example, the following command gives us the number of times that each value of the 4th column appears in our dataset:

```
$ cut -d , -f 4 adult.data | sort | uniq -c
    933  10th
   1175  11th
    433  12th
    168  1st-4th
    333  5th-6th
    646  7th-8th
    514  9th
   1067  Assoc-acdm
   1382  Assoc-voc
   5355  Bachelors
    413  Doctorate
  10501  HS-grad
   1723  Masters
     51  Preschool
    576  Prof-school
   7291  Some-college
```

That is, the first `cut` command in the pipeline projects the data only on the 4th column. Then the second `sort` command sorts the projected data in the lexicographical order, so that the same values appear consecutively. Then the last `uniq` command "collapses" consecutive identical lines into one, and prepends the collapsed count (`-c` flag) before each value.

Note the use of `sort` before `uniq`. `uniq` detects repeated lines only if they are adjacent, so it is crucial that we `sort` the file first before `uniq`.

`uniq` supports many flags other than `-c` including:

- `-d`: output a line only if it appears *multiple times*
- `-u`: output a line only if it appears *only once*

There are many more sophisticated uses of `uniq` when we combine it with other commands in a pipeline. To further explore the `uniq` command, read its man page by `man uniq`.

Aggregation Functions: `datamash`

We just saw how we can simulate the "COUNT()" function using `uniq`. `datamash` command can be used to apply other aggregation functions to text-based data. The general syntax of `datamash` is:

```
$ datamash [option] operation [field_num] [operation field_num ...]
```

where `option` is an optional list of flags (e.g., `-t ,` to specify `,` as the field separator) and `operation` is the operation (e.g., `sum`) to be applied to a particular field `field_num`. For example, the following command

```
$ sort -t, -k 10,10 -k 2,2 adult.data | datamash -t, groupby 10,2 sum 3
```

“groups the data by” the 10th (sex) and the 2nd (types of employment) columns (`groupby 10,2`) and sum the values of the third column in each group (`sum 3`). Note the use of `sort` before `datamash`. Similarly to `uniq`, the `groupby` operator in `datamash` assumes that the rows with the same groupby value appear consecutively, so it is important that we sort the data based on the groupby attributes first. Since soring-before-grouping is a frequently used pattern, `datamash` supports the optional `--sort` flag, which sorts the input data by the `groupby` attributes first. Using this flag, the above command can be rewritten as:

```
$ datamash --sort -t, groupby 10,2 sum 3 < adult.data
```

The following command shows a more complicated example of the `datamash` command. It prints the row count, max of the first column, and the average of the third column within each group of 10th and 2nd column values.

```
$ datamash --sort -t, groupby 10,2 count 1 max 1 mean 3 < adult.data
```

Note: We could have used any column number for the `count` operator (not just 1) and got the same result.

By default, `datamash` outputs the groupby columns and the results of the operations specified in the command. If we want to print *all input columns*, not just those, you can use the `--full` flag. For example, the following command will print out the whole line that contains the maximum 3rd-column value (and append the identified maximum value at the end).

```
$ datamash --full -t, max 3 < adult.data
```

It is also possible to explicitly add a particular column value to the output using the `cut` operator. For example, `cut 2` will add the value from the second column to the output.

To explore more, read `datamash`’s man page by `man datamash`.

“Update” a file with `sed`

sed is a stream editor, yet another text processing and transformation tool, similar to `awk`.

The generic `sed` pattern is

```
$ sed 's/<string to replace>/<string to replace it with>/g' <source_file> > <target_f
```

For example, the sed command below changes all occurrences of the symbol "?" in the input to "NULL":

```
$ sed 's/?/NULL/g' adult.data > adult.null
```

Let us make sure that this is indeed what happened:

```
$ grep '?' adult.data | head -2
40, Private, 121772, Assoc-voc, 11, Married-civ-spouse, Craft-repair, Husband, Asian-
54, ?, 180211, Some-college, 10, Married-civ-spouse, ?, Husband, Asian-Pac-Islander,

$ grep 'NULL' adult.null | head -2
40, Private, 121772, Assoc-voc, 11, Married-civ-spouse, Craft-repair, Husband, Asian-
54, NULL, 180211, Some-college, 10, Married-civ-spouse, NULL, Husband, Asian-Pac-Isla
```

We can see that all occurrences of ? in the first two lines have been substituted with NULL. Furthermore,

```
$ grep '?' adult.data | wc
   2399   35985  326606
$ grep 'NULL' adult.null | wc
   2399   35985  339392
```

we can see that the number of lines that contain ? in the original file is identical to the number of lines that contain the string NULL in the output file.

A very common use of sed is when a file is corrupted or badly formatted, such as with non-UTF-8 characters or a misplaced comma. You can correct that file without actually opening it using the above sed command.

Here, we showed the use of sed as a simple string substitution example, sed supports much more complicated syntax and operations. To learn more, read the GNU sed user guide.

More useful commands

In this part, we went over a few Unix tools that are useful for data processing and analysis tasks on text-based data. There are many more useful tools available, such as join (the "join operator" for two data files) comm (the "set operator" for two files), and jq (command-line JSON parser). As you can see, the majority of SQL operators are available as Unix shell commands, so you can express most SQL queries as Unix shell commands as well. These commands come quite handy especially when we need to perform a one-time quick data analysis task on text-based data. Sometimes, the overhead of setting up a database server and cleaning, parsing, and formatting, and loading the text-based data into the database may not be worthwhile for the given task. Be sure to remember and use these text-based Unix command-line tools. They can be your great friends can boost your productivity significantly!

Part B: Analyzing Google N-gram Data

Your job in the second part of the project is to perform a few data analysis tasks on the Google Books N-gram dataset using the tools that you learned in Part A.

Download Google Books N-Gram Data

Google books N-gram is an n-gram (a fancy name for n consecutive words) dataset constructed by Google from a corpus of digitized texts containing about 4% of all books ever printed. Since its initial release, this dataset has been used for many research projects, such as tracking the popularity of words over time since 1800.

To perform data analysis on this dataset, download the 1gram-s file into the `/home/cs143/data/` folder first:

```
$ mkdir -p /home/cs143/data
$ cd ~/data/
$ wget http://storage.googleapis.com/books/ngrams/books/googlebooks-eng-all-1gram-201
```

This is a small subset of the Google N-gram data, but is still fairly large (2.2GB and 440MB before and after compression, respectively). The downloaded file contains statistics on 1-grams (single word) that start with the alphabet `s`. The data in the file has the following format:

```
ngram TAB year TAB match_count TAB volume_count NEWLINE
```

For example, here are the 10,000,001st and 10,000,002nd lines from the downloaded 1-grams (googlebooks-eng-all-1gram-20120701-s.gz):

```
$ zcat /home/cs143/data/googlebooks-eng-all-1gram-20120701-s.gz | head -10000002 | ta
Somertons_NOUN  2006    6   2
Sommes  1700    1   1
```

The first line tells us that in 2006, the word "Somertons" (as a NOUN) occurred 6 times in total in 2 distinct books.

Note:

1. The `zcat` command lets us "cat" a compressed file on the fly without uncompressing it first.
2. When a word can be used as multiple parts of speech (POS), Google N-gram dataset distinguishes different POS uses of the word by adding the suffix `_POS-TAG` to the word like `Somertons_NOUN`.

Note that the data in the downloaded file has been sorted first by the 1-gram and then by the year.

Write Queries Using Unix Commands

Now that you have a reasonable understanding of the N-gram data, write the following queries ***using only the Unix text-processing tools available in our `unix` container***.

In all queries below, maybe except Query 5, if a word has multiple POS suffixes, consider each of them as a distinct and separate 1-gram. Also, assume that 1-grams are case-sensitive. For example, the 1-gram "fair" should be considered a distinct 1-gram from "Fair" or "FAIR".

1. Find the 1-gram and the year in which the 1-gram's match count in that year is at least 1,000 times as large as its volume count. For each of such (1-gram, year) pairs, print "1-gram TAB year".

   Note: The field separator in the Google N-gram data is the TAB character. Fortunately, in all Unix commands that we learned in Part A, the TAB character works well as a field separator by default. Therefore, you do not need to explicitly specify the field separator. For this reason, the TAB character is the most frequently used field-separator in Unix.

2. Find the earliest year in which there exists 1-gram that appeared in 10,000 or more volumes in that year.

   Note: Differently from RDBMS, when you pipe multiple commands, the order in which they appear in the pipeline may have significant performance implications. You may want to experiment with different orderings of the commands and check how they affect the performance.

3. For every 1-gram, sum up its match counts over all years. Return "1-gram TAB total-match-count" for each 1-gram whose total match count is 1,000,000 or more.
4. For every year since 1900 (inclusive), find the most frequent 1-gram in the year (in terms of its match count) and return "most-frequent-gram TAB year TAB gram's-match-count" triple per each year.
5. Assume that if any 1-gram contains the character `_`, it is suffixed with a POS tag. Remove all 1-grams that have a POS suffix. Then for each remaining 1-gram, sum up its match count since 2000 (inclusive), and return the "1-gram TAB total-match-count" pair for top-10 most frequent 1-grams.

Example results

Writing and debugging code on a large dataset is often time-consuming and difficult, so when you develop code, it is a good idea to work on a smaller dataset than your real dataset. This way, you can iterate over and improve your code more quickly. Also, your code will produce smaller outputs that are easier to investigate and verify.

To help you debug your code, here are a few example outputs on subsets of our dataset.

Results on 1% subset

For the first 1,000,000 lines of the `googlebooks-eng-all-1gram-20120701-s.gz` data, you will get the following results:

- Q1: No such 1-gram exists in this subset
- Q2: 1899
- Q3:

```
Sacramento_NOUN 1439069
Schmidt_NOUN    2376015
Seminar_NOUN    1050107
Sister_NOUN     3110162
salt    16517989
school_ADJ      4745646
segregation     3254460
serial  3199018
showers_NOUN    1335121
side_VERB       1336557
simplify        1657953
sinister        1749349
sufferings      3237827
Sultan_NOUN     3245143
saline_ADJ      1050863
scan_NOUN       1713771
society 74089228
steer_VERB      1442579
stillness       1367259
submarine_ADJ   1489777
suitably        1243347
SS      3623180
Sanskrit_ADJ    1116346
Shanghai        3165636
Summer  4779365
scared_VERB     2789288
seen_VERB       115306744
shareholder_NOUN        2087352
sketch_NOUN     4914194
solitary_ADJ    4416134
stem    8275024
subordination_NOUN      1734265
surrounding_VERB        11871571
seasoned        1093301
semi    7395838
shining 4631497
```

- Q4: `seen_VERB` is the most frequent 1-gram in all years between 1900 and 2008.
- Q5:

```
society    20045686
salt       3640604
stem       2319936
semi       1803709
Summer     1348437
shining    1185450
SS 1177424
serial     1111979
segregation        1027243
Shanghai   990835
```

Results on 10% Subset

For the first 10,000,000 lines of the `googlebooks-eng-all-1gram-20120701-s.gz` data, you will get the following results:

- Q1:

```
Sangalan    2005
seyis   2005
Sophrina    1991
Sumico  1957
SPIEGELBERG_NOUN    2006
Srch_X  1986
Shearstown  2000
SISON_NOUN  1977
Softco  2003
```

- Q2: 1899
- Q3: There are 403 such 1-grams. Here are ten example 1-grams and their counts:

```
swamps  1269751
subordination   1767368
schooling   3389113
speeches    5832682
Schmidt_NOUN    2376015
scratch 2009333
Sale    1858853
stem    8275024
sufferings  3237827
saddle_NOUN 3348571
```

- Q4: `States` is the most frequent 1-gram in the years 1941, 1943, and 1976. Other than these years, `shall_VERB` is the most frequent 1-gram between 1900 and 1977. Between 1978 and 2008, `She_PRON` is the most frequent 1-gram.
- Q5:

```
say 42194666
States  30389606
sense   27438733
short   21400700
society 20045686
seemed  19207838
South   16699076
started 14542589
seem    14082903
status  12458752
```

What to Submit

Create Query Scripts

Once you finish writing and debugging the queries, write five shell script files, named as `q1.sh`, ..., `q5.sh`, that contain each of the Unix-command queries that you wrote in Part B. In writing your scripts, please make sure that they meet the following requirements.

Script requirements

1. The names of your query scripts must be `q?.sh`, where `?` is the query number.
2. Your code must take the file at `/home/cs143/data/googlebooks-eng-all-1gram-20120701-s.gz` as its input data.
3. Each of your query scripts should be executable simply by the command `sh ./q?.sh`. For example, we should get a result similar to the following when we execute:

```
$ sh ./q2.sh
1898
```

4. Your code must print the output to `stdout` (i.e., on screen).
5. Each line of your output must contain only the specified columns exactly in the given column order.
6. Preferably, each column of your output should be separated by TAB, but it is OK to use (multiple consecutive) white space(s) as the field separator.
7. The output row order does not matter. As long as your code produces exactly those rows that it is supposed to produce, you will be fine.
8. Your code must use only the Unix command-line tools that are preinstalled and available in our `unix` docker container. Do not install or use any other programming languages (e.g., Python or PHP) or tools.

9. Your code must not leave any traces behind except the printed output. In particular, if your script created any file to save an intermediate result, your script must delete the file before it exits.

Create Zip File

Once you finish preparing the query scripts according to the spec, create `project6.zip` that has the following packaging structure.

```
project6.zip
 +- q1.sh
 +- q2.sh
 +- q3.sh
 +- q4.sh
 +- q5.sh
 +- README.txt (optional)
```

To help you package your submission zip file, we have made a packaging script p6_package, which can be run like the following in the directory where your query scripts reside:

```
$ ./p6_package
zip project6.zip q1.sh q2.sh q3.sh q4.sh q5.sh
  adding: q1.sh (deflated 2%)
  adding: q2.sh (deflated 12%)
  adding: q3.sh (deflated 7%)
  adding: q4.sh (deflated 11%)
  adding: q5.sh (deflated 15%)
[SUCCESS] Created '/home/cs143/shared/project6.zip'
```

(You may need to use “chmod +x p6_package” if there is a permission error.)

When executed, our packaging script will collect all necessary (and optional) files and create the `project6.zip` file according to our specification that can be submitted to GradeScope.

Submit Your Zip File

Visit GradeScope to submit your zip file electronically by the deadline. In order to accommodate the last-minute snafu during submission, you will have 1-hour window after the deadline to finish your submission process. That is, as long as you start your submission before the deadline and complete it within 1 hour after the deadline, you are OK.