COMP1204: Data Management
UNIX Coursework I: Traveladvisor Hotel Reviews

Oliver Rees - or1g18

February 24, 2019

Contents

1 UNIX Scripts

1.1 Count Reviews Bash Script

countreviews.sh is run via the command line; the exact commands written are:

$ ./countreviews.sh [path to hotel files]

The script counts the number of reviews in a *hotel.dat* file. It does this by counting the occurrence of *<Author>* in the file with **grep**: *grep -c "<Author>" $file*. A run through of the bash script is given below:

countreviews.sh

```bash
#!/bin/bash

hotelDir=$1

for file in $hotelDir/*
do
   fileName=$(basename $file .dat)
   reviewCount=$(grep -c "<Author>" $file)
   echo $fileName $reviewCount
done | sort -n -k2 -r
```

```bash
#!/bin/bash
```

This line tells the shell that the script uses the **bash shell**, and the bash shell is located in */bin/bash*. Interestingly, the character # starts a comment in bash.

```bash
hotelDir=$1
```

This line stores the command line's argument in the variable **hotelDir**. The *$1* calls the value of the first argument.

```bash
for file in $hotelDir/*
do
   fileName=$(basename $file .dat)
   reviewCount=$(grep -c "<Author>" $file)
   echo $fileName $reviewCount
done | sort -n -k2 -r
```

The first line here begins an enhanced **for loop around all the files** in the argument-provided path; the use of the wildcard in */** enforces that all files in the directory are checked. Likewise, the second line starts said for loop with **do**, and the sixth line ends it with **done**.

```bash
   fileName=$(basename $file .dat)
   reviewCount=$(grep -c "<Author>" $file)
```

Within this snippet of code is the assignment of two variables, **fileName** and **reviewCount**.
fileName uses the **basename** command to remove the path from the file's name, as the command normally does, as well as the expected extension which is *.dat*.
reviewCount uses **grep -c** to count the number of times the expression <*Author*> appears in each file.

```
   echo $fileName $reviewCount
```

The use of **echo** with the call of fileName and reviewCount is done to **print** out the expected output onto the terminal screen.

```
done | sort -n -k2 -r
```

This last line involves a **pipe** to **sort**. The important options in sort are *-k2*, which means the **second field** (the review count) are the values sorted, and *-r* which means to sort in **descending order**.

1.2 Average Reviews Bash Script

averagereviews.sh is run via the command line; the exact commands written are:

$ ./averagereviews.sh [path to hotel files]

The script **calculates the average of all the reviews** in a *hotel.dat* file. It does this by dividing the aggregate overall scores of each review with the review count (as done in the countreviews script), using **awk**. A run through of the bash script is given below:

average reviews

```
#!/bin/bash

hotelDir=$1

for file in $hotelDir/*
do

   fileName=$(basename $file .dat)

   reviewCount=$(grep -c "<Author>" $file)

   ratingSum=$(grep "<Overall>" $file | sed 's/<Overall>//g'
      | awk '{sum += $1}END{print sum}')

   averageRating=$(awk -v c=$reviewCount -v s=$ratingSum
      'BEGIN {printf("%0.2f", (s/c));}')

   echo $fileName $averageRating

done | sort -n -k2 -r
```

In this script, the first nine lines were explained in 1.1, so I won't explain them again. The new code is explained below:

```
   ratingSum=$(grep "<Overall>" $file | sed 's/<Overall>//g'
      | awk '{sum += $1}END{print sum}')
```

These two lines, which is really one line, count the values associated with the **Overall** key in each review.
The use of **grep** prints out each line with the expression <*Overall*>, which will contain the values we want to count. These lines are piped to **sed**.
sed substitutes the <Overall> tag in every line for nothing: **sed removes the <Overall> tag**. The result of the sed operation is that **the value is piped to awk**.
Awk adds up the aforementioned value to **sum**. The use of *END* means that

the summation is looped around all the lines, as opposed to the value on the last line. Sum is returned and stored in the variable **ratingSum**.

```
averageRating=$(awk -v c=$reviewCount -v s=$ratingSum
   'BEGIN {printf("%0.2f", (s/c));}')
```

averageRating stores the **average overall rating** of the hotel, by dividing the previously calculated variables, ratingSum with reviewCount. It uses *printf("%0.2f")* so that the value returned is to **2 decimal places**. The division is done via **awk** in order to have the division return an non-integer result. The variables are assigned to **s** and **c**.

```
echo $fileName $averageRating
```

These last lines are the same as in 1.1, with the only *slight* difference being the call to averageRating instead of countReviews.

2 TripAdvisor Script Review

2.1 Review of the Script's Limitations

2.1.1 Displaying Data

The output of both scripts is quite lengthy, with the calculated data from each hotel printed on its own line. If there were more than a million hotels, this would mean more than a million lines to search; searching data for a small subset of hotels would be especially cumbersome.
It's possible that such an issue can be avoided by storing the results in a structured document and allowing a querying language like SQL to request the subset of hotel data the user wants. This is further explored in 2.4.

2.1.2 Assumptions made in the script

The scripts relied a great deal on both the format of the hotel files, as well as its metadata. In this section, I'll look at what information the scripts relied on, how this could affect their accuracy, and how to reduce that reliance.

The scripts assumed that for each review in the hotel data file there is an <Author> tag, and an <Overall> tag.
If either of these tags failed to appear in a review, then the **reviewCount** and **ratingSum** would be inaccurate, as the reviewCount would be lower than the number of reviews, and the ratingSum would be the rating sum of a unspecified subset of the total reviews. Altogether, the countreviews.sh and averagereviews.sh would print inaccurate results.
A solution to this would be to force the generation of these tags, in particular, when the hotel data file is generated.

Another assumption made in relation to the format of the hotel data file's content was that the Overall tag always had a value attached to it. If a value failed to appear next to the Overall tag then the value for that review would be treated as 0, even if the review was positive, which would skewer the average rating against the hotel's favour.
A solution to reduce the skew is to force the user to submit an overall value when they're writing the review.

Another assumption made by the scripts is that all the files have a *.dat* extension.
If a targeted folder, which we'll assume only contains hotel files, had some hotel files listed with an extension *other* than *.dat*, this would cause the extension to not be omitted, as per the specification.
A solution to this could be to use sed to remove the extension by using the dot as a field separator. However, this would mean the filename can't have any dots in it. This is preferable to restricting the file extensions in the directory,

as having dots in the filename is irregular, whilst the alternative is common.

The script relies on all the files in the given directory, adhering to the hotel data format, but there may be some files such as a log file, or even unrelated files. This would mean that they'd be counted in the display, and for averagereviews.sh they may return an error as there would be no Overall tag to replace.

2.2 Authentication of the Reviewers

2.2.1 Authentication Issues

Issues may arise due to the authenticity of the reviews.
Sometimes, some people may review hotels they have never reviewed. The Unicode text (*.dat* files) do not support such verification, but such verification is possible on the client side.
Although this is typically the case, what if multiple reviews were made by one person in some coordinated attack on the hotel? Would it be fair to count those multiple reviews?
A solution to this would be to pipe the grep result to sort and then to uniq, to count the reviews as one review,

```
$ grep "<Author>" | sort -t'>' -k2 | uniq
```

2.3 Assessment of the Script's performance

2.3.1 Efficiency Problems

Firstly, a massive challenge in terms of execution time is how quickly the scripts can loop across a large number of files.
Secondly, the scripts need to loop through each line on a file. This can be an issue for files with a large number of lines (over a million).
These are both small issues, but they create an annoyance and can be avoided.

2.3.2 Improving the Performance

There are two solutions to improve the performance: use a relational database, or use a compiled language as opposed to the slower, interpreted bash.
Using a relational database would mean that not every line would need to be looped; all the script would need do to is count the number of records to calculate the reviewCount, or sum up every value in the Overall column: some fields would be ignored. This would fix the hit to performance caused by looping through every line.
Using a compiled language would mean that the time taken to execute the script would be significantly less. Such a language is advisable to use, since the script's content is never changed, and doesn't need to be recompiled each time it's used.

Due to UNIX's close history with the C language (a quick compiled language), it's possible to write the 'scripts' in C.

2.4 A short look at Structured Data vs Unstructured Markup

2.5 Using Flat Files

Flat files are databases stored in a plain text file. Tripadvisor's hotel data files are stored in a *.dat* file; according to the file process, *.dat* files are "UTF-8 Unicode text, with very long lines, with CRLF line terminators". Therefore, the hotel data is stored in flat files.
The advantages of using flat files is that they're memory efficient, and so a lot of them can be stored in a secondary drive, and they're easy for a human to understand, making it easy for anyone to write the hotel data file.
However, searching through flat files takes much more time than with relational files, and so our scripts are less efficient. This is because there is repetitive and redundant information in flat files, as a result of their lack of relational information; this unnecessarily large information leads to more lines being looped around.
Furthermore, flat files offer a major security flaw: there's no encryption in the files. This means it's easy for a hacker to confidential data on reviewers and hotels; however, if the data files exist to store the model of a public review site, then it would be different as the data would be public and not as necessary to protect.

Overall, for our hotel files, it would be preferable for the hotel data to use flat files for small amounts of information, as the execution times of the script would be much less; however, if the files are stored in one root directory, then it would be preferable to use flat files, because of their memory efficiency.
I am ignoring the security flaw here as I assume TripAdvisor would want to make this information public for potential customers of a hotel; however, if TripAdvisor were using these files to give a confidential report to a board or business, it would be imperative they use relational files.