Project 2
CSC8622

Pedro Pinto da Silva, Alex Kell

November 28, 2016

Design and Implementation Report

Categorisation of Artefacts in the CSV

The CSV file provided did not store categories for each item in the relevant row. They were instead present in their own row at the beginning of each section. This had certain draw backs, such as not retaining the information when querying the data, as ordering may be lost.

AWK was used to do this, which is a programming language built for the purpose of extracting and transforming text.
The code use is shown below:

```
awk -F "," '/,,,,,,,,,/{flag=1; a=$1; next} BEGIN{OFS= ","}
        {$1=a FS $(1) ; print} ; /,,,,,,,,,/{flag=0} ' $1 > $2
sed -i.old '1s;^;Category;' $2
```

The code works as follows:

1. The awk function refers to the Unix program which runs scripts in the AWK programming language
2. As the text file we were using was seperated by commas (CSV), the −F "," function was used to specify how each field was split
3. /,,,,,,,,,/ is a regular expression which searches for rows with a series of empty columns
4. {flag=1; a=$1; next} When the regular expression is found, the flag is set to (ON), and the first entry of the field (Category name) is saved as "a". This value of "a" is then inserted into the first columns of all the rows until the next row with a single entry in the first row is found.
5. The sed −i.old function names the newly created category.

The problem with this code, is that it expects the data format to be provided in the same way each time. ie. With the category in its own row, and no entries in the subsequent elements. If an entire category is removed this would not work.

Extraction of useful columns/fields

This section returned only the first 5 columns, which were necessary for analysis. Whilst this problem seemed trivial at first, it was not an easy task with awk due to some fields containing commas. For instance addresses were returned as "Heworth, Gateshead". AWK would interpret "Heworth and Gateshead" as seperate fields, as opposed to a single field.

The chosen solution was to use csvkit. A suite of utilities created specifically for working with CSV files. This program automatically ignored commas which were found in between quotation marks. The code when using csvkit is simple and shown below.

```
csvcut --columns 2,3,1,4,5 $1 > $2
```

An assumption made here was that the columns would remain in the same order, and that strings which contained commas would be encapsulated by quotation marks.

Including information from alternative sources

This section was concerned with the use of additional text sources, and formatting these in a way so that they could be integrated with the main database. Bash was used to do this.

This was done by reading the file, and using an "=" as a delimiter. The file was iterated through, extracting the second field. This second field, which contained the valuable information was then printed to the file, in the correct order with a comma as a delimiter. The same format as the main CSV file.

Merging sources of data

This section concerns itself with the merging of the different data sources which we now contained. This was done through the use of bash.

The columns names were hard coded and printed in the first line. A for loop was then created. This for loop used each of the files parsed as arguments to iterate through. All the lines of each of the files was then printed and saved as a file.

Removal of duplicate data

This section concerned itself with the removal of duplicate items within the database.

To do this, awk was used once again.

First of all quotation marks are removed.
The awk internal function (seen) function evaluates to true if an element has not been seen before. It is therefore evaluated on the entire first column. At values that it is true, the column is printed.

Importing dataset into a database

This section concerns itself with the creation of a database, and insertion of the data. Python was used to do this due to the packages it contains to do this.

The first sections of the code take the arguments of database name to be created, and CSV file to be imported. Deleting the existing database if one with the same name exists.

The database is created using sqlite3 package commands. Name each of the columns. The delimiter is set to a comma, and each of the rows are printed in their respective columns. Once this has been done for the entire length of the CSV file, the connection is closed and database is saved.

Database Queries

This section concerns itself with the creation of a few queries to return certain values from the created database.

The sqlite3 package for python was used once more here. SQL language was used for each of these queries, and the results were written to a file.

Linking high-resolution images to the database

Pipeline wrapper

This section concerns itself with the creation of a pipeline wrapper. To do this Make was used.

For this, each of the parts previously mentioned had their own section in the make file, where the output of the previous section was used to create the output of the current system, and forwarded to the next section and so forth.

Once all of the files had been created, all of the intermediate files were deleted, returning only the original file.

Wildcard Exercise

To trigger a test sequence everytime we pushed onto the remote repository, we decided to use Continuous Integration, namely a platform called Travis CI. Travis CI has direct integration with Github: you can login using your account and activate directly which repositories use continuous integration and which don't. We set up travis to run our pipeline (until the point at which he was implemented) everytime there was a push into the Github repo. Setting up travis yaml configuration file was straightforward. We quickly specified the target language (bash, python), test script, dependencies and even set up travis no notify us about the result of our builds via slack and email. In overall, I think we had a productive first contact with continuous integration and got the grip of the importance that underlies the concept. We also aimed to build a customized docker container for our pipeline but we failed to do so due to lack or mismanagement of time.

What we missed and could have improved

Our work fails in the aspect of test-development, as we at times relied in manual inspection of output files to verify the success of the operation. It was not easy to set up well-defined tests for each of the stages of the pipeline because testing scripts like the ones we developed would required extensive test-suites for which there was no time to work on. We could have developed small tests and assertions when we knew what the outcome of a given stage of the pipeline was, e.g. number of lines in output csv file. However, these would become irrelevant in the presence of a very large dataset for which human introspection would not be possible.

Other aspects could have been improved too, such as writing a more generic makefile and improve on the directory structure of the package.