Overview of CiViC Data Processing

Gabriel Meltzer, 4/23/2019

Project Length ~ 30 hours, Python 2.7.15 (64 bit)

The script, named `civic_data_procesing.py` simply consists of downloading the files through the use of the `urllib` module, with three custom functions, `extract`, `transform`, and `load` called after the definitions.
The `pandas` library was the main third-party library used for the project. The script called the function `read_csv` on each downloaded file; changing the separator to tab through the function argument `sep` allowed `read_csv` to create DataFrames from the `tsv` files.

In order to only retrieve the evidence associated with drugs, the script created a new DataFrame that only contained the rows of the `evidence` DataFrame where the information in the `drugs` columns was not missing; in this case, it is assumed the evidence has nothing to do with drugs.

As part of the task of excluding non-missense variants, the script excludes rows that have missing data in the `variant_types` column. These rows, it was assumed, should not be treated as missense variants, since it is unknown what type of variants they actually were. Then, the script performed the `match` function on the column against the phrase "missense_variant" (the `contains` function was used prior to this, although only the `match` function was needed), and excluded everything from the DataFrame.

Removing non-specific variants was the most difficult part of the assignment, and created a roadblock. It was unclear which column the AA position information should come from, nor was it immediately obvious how non-specific variants would be indicated. Nothing clarifying the situation could be found on either CiViC or the HGVS website, nor was it clear why these nonspecific variants would be grouped on the same row. Nevertheless, despite my uncertainty and questions regarding these matters, I found a tentative path forward. The assumption was made that the `variant` field should be relied upon, rather than the `hgvs_expressions` column, since that field contained less missing information. Regular expressions were used to pull the desired information from the `variant` field. Since the format of the data in this field was not standardized, this was a time-consuming process. Finally, if the numerical portion of the variant description notation was the same for two descriptions within the same row, it was assumed that they should not be treated as non-specific variants. More information about the creation of the regular expressions exists in the comments of the script.

For the creation filtering specified in step 4, the script used the `isin` function to perform filtering on one DataFrame based on the contents of another DataFrame.

Creating a mapping table for `evidence_drug` presented another roadblock. The assumption was made that the drugs were delimited by commas, and nothing else, but there were a few instances where commas existed within parentheses. The resulting information was confusing and impossible for me to interpret. If I had a little more information, it would be possible to again use regular expressions to solve the problem. How many drugs were referenced in these particular rows? What terms referred to different drugs, and what terms referred to different names for the same drug? Since I did not have the knowledge to make these calls or even make reasonable deductions, I opted to continue with my initial assumption.

The rest of the procedure of separating the different drugs in the mapping table column involved the use of a temporary DataFrame, loops, `split`, and `append`. The process was identical for the `variant_alias` DataFrame, except the assumption was also made that the mapping table should not include variants with missing information in the `variant_alias`, since those variants have nothing to contribute to the table.

In order to create a new `aaPos` column, regular expressions as well as the `extract` function (to capture the group) retrieved position information from the variant descriptions in the `variant` field. This information was then placed into a new `aaPos` column.

Loading this information in the sqlite3 database tables required the `sqlalchemy` library, as well as the `pandas to_sql` function. The database file was named assessment.db. Because `pandas` automatically made certain columns `floats`, there were also last-minute conversions into integers so that the information would be displayed in the appropriate format.

The bonus task was also achieved successfully through the `sqllite3` command line utility.