CMPT 473 Spring

Assignment 2: Testing with Input Space Partitioning

Bonnie Ng
Kylie Loo
Sonny Myet

1. Specification of the Program Under Test

The program under test is CSVkit 1.0.2. (*https://CSVkit.readthedocs.io/en/749/)*

It is a suite of command-line tools that allows the user to work with CSV files. It includes a few features including converting to and from CSV format, modifying and displaying CSV columns and interacting with databases with CSV format. The tools can be categorized into 3 major categories: Input, Processing and Output & Analysis.

The particular portion of the program we will be testing is the CSV to JSON functionality of the output and analysis category command, "csvjson". This function takes in a csv file along with some configuration options and converts the files content into JSON format.

Reference - *http://CSVkit.readthedocs.io/en/1.0.1/scripts/CSVjson.html*

2. Input Space Partitioning

Usage

Below is the usage of csvjson as described in the document page. For example, a sample usage with default flags would be:

$ csvjson data.csv

```
usage: csvjson [-h] [-d DELIMITER] [-t] [-q QUOTECHAR] [-u {0,1,2,3}] [-b]
               [-p ESCAPECHAR] [-z FIELD_SIZE_LIMIT] [-e ENCODING] [-S] [-v]
               [-l] [--zero] [-V] [-i INDENT] [-k KEY] [--lat LAT] [--lon LON]
               [--crs CRS] [--stream] [-y SNIFF_LIMIT] [-I]
               [FILE]
```

Note: for this assignment only we did not consider the GeoJSON functionality the program offers, just to save us from building another test suite with a whole new set of flags.

Arguments

The tool of CSVkit is the first argument; in this case, csvjson. Options (overrides) can also be added onto the command via arguments. The options available for this function are like configuration flags. Some example arguments would be:

-t	specifies that the input CSV file is delimited using tabs. Considerations: - If tab character exists in CSV - If tab character doesn't exist in CSV
-q QUOTECHAR	specifies that character used to quote strings in the input CSV file. Our considerations: - If character exists in CSV - If character doesn't exist in CSV - If quote isn't done properly (i.e. open)

The last argument is the input file which must be in the format .csv.

Input Domain Model and Characteristics

We used an interface-based approach to identify our characteristics because the options themselves hold the semantic meanings of the program. Also, my choosing the parameters based on the options,

our input domain would almost be trivial to define yet still effective in capturing the program's behavior..

A lot of the properties we chose (see below) are input parameters that bring up environmental issues with the CSV file itself. This is because the output is influenced by any discrepancies between the flags and the file since the flags are to be used more like configuration options based on the file content characteristics. Therefore, we have split the classification of characteristics into "parameters" and "environment" where parameters are the characteristics that deal with the command interface itself, and "environment" characteristics deal with the characteristics of the input file contents itself. While this would further increase the number of possible test files needed, some of the flags depend on the same property, and thus the test artifacts themselves may be reused.

To illustrate why this would be needed, let's say the user includes the -t flag in the command which sets the delimiter to a tab character. If there are no tabs in the CSV file, we need to consider what the resulting behavior of the program would be in both cases in which tabs do and do not exist in the file. Conversely, the function of tags like -l (--linenumbers) are independent of the file's characteristics so we did not need to include an environment property for it. Some flags also require an argument value after it, and because we are excluding test cases that create an error due to improper usage, we assume that the when the value of a parameter characteristic is TRUE and requires an argument, an appropriate argument will be provided.

Parameters

isFlagged	Boolean	[**true**,false]
isValidFlag	Boolean	[**true**,false]
flaggedDelimiter	Boolean	[true,false]
flaggedTabbed	Boolean	[true,false]
flaggedDoubleQuote	Boolean	[true,false]
flaggedEscapeChar	Boolean	[true,false]
flaggedFieldSizeLimit	Boolean	[true,false]
flaggedEncoding	Boolean	[true,false]
flaggedSkipSpace	Boolean	[true,false]
flaggedVerbose	Boolean	[true,false]
flaggedLineNum	Boolean	[true,false]
flaggedZeroBase	Boolean	[true,false]
flaggedVersion	Boolean	[true,false]
flaggedIndent	Boolean	[true,false]
flaggedKey	Boolean	[true,false]
flaggedStream	Boolean	[true,false]
flaggedInference	Boolean	[true,false]
flaggedQuoteStyle	Boolean	[true,false]
isDefault	Boolean	[true,**false**]

Environment

isValidPath	Boolean	[**true**,false]
isValidCsv	Boolean	[**true**,false]

isCsvEmpty	Boolean	[true,false]
containsDelimiter	Boolean	[true,false]
delimiterEscaped	Boolean	[true,false]
stringsQuoted	Boolean	[true,false]
uniqueColumnNames	Boolean	[true,false]
keyExists	Boolean	[true,false]

Note that we eliminated the flags that *do not* change the possible output (for example, versioning).

Constraints

While most of the option flags are independent of each other, there are a few that should or must be used together. We have represented these as constraints to ensure proper flag usage. Additionally, there are some obvious dependencies of characteristics, which are also identified as constraints.

isValidPath = false => isValidCsv = false
If the CSV path is invalid, isValidCSV cannot be true
isValidCsv = false => isFlagged = false
If the input is not a valid CSV, there cannot be a valid Key comparison with its column names
isFlagged = false => (isValidFlag = false && isDefault = true)
If nothing is flagged, default options are used and validity of flags need not be checked
isDefault = true => isFlagged = false isValidFlag = false => (flaggedDelimiter = false && flaggedTabbed = false && flaggedDoubleQuote = false && flaggedEscapeChar = false && flaggedFieldSizeLimit = false && flaggedEncoding = false && flaggedSkipSpace = false && flaggedVerbose = false && flaggedLineNum = false && flaggedZeroBase = false && flaggedVersion = false && flaggedIndent = false && flaggedKey = false && flaggedStream = false && flaggedInference = false && flaggedQuoteStyle = false)
If any of the flags are set, the default fallback is not used. Conversely, if user is using the default command then all the flags are turned off *This doesn't mean that the same behaviour will not occur in the two scenarios. This is only concerned with the user's input*
isCsvEmpty = true => (containsDelimiter = false && delimiterEscaped = false && stringsQuoted =

false && uniqueColumnNames = false && keyExists = false)
If the file is empty, none of the environmental blocks about the file are true.
delimiterEscaped = true => (flaggedEscapeChar = true || flaggedDoubleQuote = true || flaggedQuoteStyle = true)

3. Combinatorial Test Generation using ACTS

After determining input domain model, we started a new test suite using the default algorithm from scratch on strength 2. A total of 14 test cases were generated after providing the characteristics above as 27 parameters and providing the constraints. This resulted in a truth table containing the possible test cases that we could model our test suites after.

We created our test system according to the assignment suggestion. A test harness was was made using C++ which runs each test frame located in a folder containing test data. Since the program we are testing is run on the command line, we organized our test frames as a command in a text file that the test harness would run. The commands were formed by referring to the values in the test cases generated by ACT. The necessary CSV input files were also contained in the same folder under a test data folder. All the test files for the entire test suite is contained there because as mentioned earlier, some test frames/test cases may use the same test data, reducing the amount of effort needed to create test artifacts. Additionally, two folders containing expected output files and expected error files were also created along with the appropriate files to be used to check the results from running the test.

Since we were testing a utility that converts CSV to json files, we generated CSV files using a spreadsheet program that could export properly formed CSV files.

Separate folders contained the output and error messages resulting from running the test script. The results were then compared using the ‘diff’ command to check whether the program ran as expected.

4. Report

Although ACTS only explicitly supports enums, booleans and integers, the characteristics that we identified in our program were easily adapted to fit into one of the types. It was easy to spot the possible characteristics to test when we made the distinction between the program’s options and the input file’s content characteristics. Originally we tried to use an *enum* type to hold the option flags, but that would only allow us to testone flag at a time. Thus, we changed the option flags to independent boolean characteristics that represented their presence with the correct usage and their

absence. To identify characteristics of the input file, we started by looking at the possible flags and determining what characteristics would need to be present in the file to use the flags.

We used an interface-based approach to determine the constraints on the characteristics; because we understand the format of typical CSV files, we reduced the number of tests by eliminating the impossible. This helped us reduce the number of characteristics to test.

We also wondered what the expected output would look like according to certain interpretive nuances. For example, would a CSV file looking like ",,," generate its own headers? What would the headers say? This could determine whether the test suite finds bugs with the program properly. We could not find a way to express these test cases via ACTS, but these are special test cases that we could manually think of and add to our test suite. Similarly, we were unable to test specific combinations of some flags due to the way ACTS generated the test cases. There were particularly interesting cases between particular characteristics, for example, like the ones regarding quotation handling. flaggedEscapeChar depends on the values of flaggedDoubleQuote or quoteStyle, and there are multiple combinations of different values that may result in some interesting interactions and thus warrant some extra test cases. As demonstrated, though we could have created extra parameters and more constraints to make sure these cases are represented by the table generated by ACTS, it would add another layer of work on top of having a boolean to represent each option type.

The ACTS tool itself was not hard to use; entering in the parameters and constraints was well-documented in the user guide. One of the early problems we faced was the many number of flags, as we had represented each flag as its own boolean value. So to make the number of tests manageable, we chose the more significant flags to include in the test case generation. More specifically, we chose the flags that would have a significant impact on how the program was to run. The flags we chose and their format was further refined as we were building our test cases. We eliminated some flags, like -lon and -lat which were concerned with GeoJSONs because we wanted to focus on normal JSON files as mentioned in the beginning.

If we were to have used all-pairs testing, we would have ended up with 2^26 * 4 = 268435456 test configurations because we have 27 boolean parameters. Using the pairwise testing technique aided by ACTS, we ended up with only 14 test configurations; a significant reduction. However, using this automatic test generation technique, we were left with many test cases that did not test the interactions of the flags themselves because there were only a few test cases that were a combination of a valid and non-empty CSV that was specified from a valid path name. We were able to set relations on our parameters such that there were significantly fewer test cases that involve an invalid path, an invalid csv file and an empty csv file, allowing us to test the interactions between the options themselves.

Having to approach testing a real system in a pairwise fashion forced us to think about the impact of each possible option for the function. However, because the individual options were more or less independent of each other, combining the parameters in the way that ACTS does it produces test cases that are not necessarily valuable unless we wrap some of them in relations and impose many

constraints. For a small command-line program with options such as csvjson, we think that it would be adequate and more effective to reason about the test cases manually since the options are so independent and are actually required to just make the program run as intended. To have ACTS generate a meaningful test suite for csvjson would mean more work and analysis than if we were to do it manually. However, we do see the potential that pairwise testing would provide if the SUT parameters were not like configuration options and more like actual values that create meaningful interactions.