CA Markdown

John McLevey
Design plan revised on July 13, 2015

General

Almost all software for content analysis and qualitative analysis of unstructured data **sucks**. The applications are slow, bloated, expensive, don't work on all platforms, have a hard time with multiple coders, etc. My idea is to simplify everything by making it about marking up plain text files, and them parsing them to produce simple tables and other output. After all, most of the work of content analysis and qualitative analysis is just categorizing text. For many researchers, the table produced by parsing the text files will be enough. For others, the tables can be easily read into R, Python, Stata, etc. for further analysis and visualization.

There are many advantages to doing it this way instead of the way it is currently done:

- because text files are everything, you are not tied to a specific application. you can work in your favorite text editor, on your tablet or phone, etc. you do the 'heavy' lifting on the command line, but the actual work of marking up text can be done anywhere. it doesn't matter because it's just plain text.
- text files are future proof, so you don't have to worry about data preservation or proprietary file formats. just look after your text files and you will never have to worry about losing access.
- text files and project folders can be easily encrypted to protect participant confidentiality in interview or field studies.
- you have access to the powerful capabilities of good text editors and command line tools, not the least of which is very fast multi-file searching and the ability to use regex when necessary. this is an important part of most (all?) commercial software for content analysis and qualitative data analysis, but they have been available in text editors for a long time. the tools in text editors are much better than the ones in the commercial applications.
- because we are working with simple plain text files, it is easy(ish) to work with multiple coders. if you end up with file conflicts somehow, you can use tools like diff to resolve them easily.
- you can use version control systems like git to track changes to your text files, which you cannot do with NVivo, ATLAS.ti, etc.
- you can easily produce output for further analysis in R, Python, Stata, etc.

  - you have access to better visualization tools than the ones that are included in the commercial applications
  - you can produce descriptive summaries and perform quantitative analyses on your code tables when appropriate
  - you can use the output tables for systematic qualitative case comparisons using Charles Ragin's fsQCA and variants. this is especially useful for projects that are focused on causal relationships or constructing ideal types.
  - you can use text mining and topic modeling packages alongside your content analysis

- You are not stuck if you don't want to use R, Python, Stata, or some other language. You can use the graphing tools in programs like LibreOffice, Google Spreadsheets, Numbers, Excel, etc. While they don't always look great, they are often on par with the graphs produced by applications like NVivo.

The main limitation of this approach is that you cannot code data that is not in plain text. No PDFs, no docx files, and no media files. If you are working with PDF files, there are some decent tools for extracting text. I think this is a reasonable trade.

Design wise, the goal is to be simple enough that someone who is not comfortable on the command line can do what they want to do given a very clear set of instructions. However because we are working with marked up plain text, anyone who is comfortable on the command line can access all kinds of powerful tools, and can easily get tables from their content analysis into R, Python, Stata, etc. So it's simple and minimal, but you have a lot of power just a few commands away.

My design ideas are strongly influenced by markdown and pandoc. Pandoc in particular has influenced the way I think about how the command line tools should be, because it is fundamentally about parsing an input file and creating output files. That said, I'm sure my *specific* ideas of what the ideal commands are could be much better. So the thing to take away are the objectives, and the general ideas and design principles. If you have better ideas, by all means **let me know**.

By the way, I'm not totally sold on the idea of calling this **"CA Markdown"** (for Content Analysis Markdown). I short of like it because it reminds me of RMarkdown and other variants, and because of the shared design principles. But it is not a markdown processor, and calling it CA Markdown might imply that it is. I don't really know what is best here. If you can think of something better, let me know. We can use CA Markdown, or *'cam'*, in the meantime.

I've got a bunch of pseudo commands listed below. Things like 'fileSetOne' correspond to the mock files I created to help show you what I mean: example.md, codebook.md, and attributes.csv.

Project Files

CA Markdown will work with (1) data files like interview transcripts or news stories; (2) a codebook that defines 'context' codes and 'content' codes, groups codes into code sets, and groups files into file sets; and finally (3) a case attributes file for storing data about study participants (e.g. demographic data, pseudonyms for publication if necessary) if the project is a field study. See the mock project files I attached for further clarification.

The codebook should have 4 sections:

1. **Context Codes**: Used to structure textual data. In an interview transcript, it would specify when one person starts talking and when they they stop. With newspaper data or abstracts from research articles, it marks where an article or abstract starts and stops.
2. **Content Codes**: Used for categorizing textual data. More below.
3. **Content Code Sets**: Used to group codes together into more general sets / categories / families.
4. **File Sets**: Used to group sets of related files. For example, all of the transcripts for interviews with women, or people of the same occupation, etc.

Grouping codes and files into sets is an important part of content analysis and qualitative analysis, especially in more inductive studies, or in studies concerned with identifying causal mechanisms. *Codes and files can belong to multiple sets at once.*

The case attributes file will be a csv file where the first column is an ID for a research participant (or organization, etc.). All other columns are determined by the specific project. Most likely this file will contain demographic data on the participants. If the study does not require this kind of data, the relevant commands (discussed below) should be able to work without it.

The data files will have meta-data at the top, similar to the types of YAML meta-data that is used in Pandoc Markdown. See the example.md file to see what I mean.

In the YAML-like meta-data section, the codebook and attributes fields specify the path to the codebook file and the attributes file. This is important because some of the commands need to know where all of those files are. To keep the command line operation minimal, it's better to put that information here.

Project Set Up

The primary idea here is that we are parsing marked up text files, which means that the work can be done anywhere in any text editor. But obviously, some

good command line tools are what makes this useful, and the first one should be to set up a project directory with the required files in sensible places. I'm thinking that a simple command like

```
cam init "my_new_project"
```

will grab a project template directory from Github that looks like this:

```
my_new_project/
    codebook.md
    attributes.csv
    data_coding/
        example_transcript.md
    data_raw/
    memos/
        extracted_memos.md
        other_notes.md
    output/
    readme.md
```

This is an ideal project directory structure from my perspective, but not everyone will use this structure, so the commands should probably not depend on this exact directory structure. But if it makes it easier, we can.

The data_raw folder is for unprocessed data, obviously. The extracted_memos.md file is special; I describe it below. The output folder is for storing results and tables produced on the command line (more below). The readme.md file will provide the necessary help / documentation.

Creating and Applying Codes

Again, the idea here is to get away from the slow and bloated way that commercial applications use for assigning codes, and instead to simply mark up plain text files, and then parse those text files to produce useful summaries and comparisons. As with markdown, the goal should be to keep it as simple and clean as possible. In some cases I have borrowed ideas from LaTeX instead of HTML because markdown processors will process HTML code as HTML code.

The basic syntax I have in mind for assigning a code is this:

```
[firstCode]{The actual text you want to code is here...}
```

It's similar to the syntax for linking in markdown except that it uses { } instead of ( ).

Grouping codes into more general sets / categories is an important part of much content analysis or qualitative data analysis. There should be two ways to do this. The first is to create code sets in a dedicated section of the codebook, which I describe below. You should also be able to do it in the data files themselves like this:

```
[setThree/secondCode]{The actual text you want to code is here...}
```

Obviously the first code "setThree" is the more general category and "secondCode" is the original code.

Syncing, Renaming, and Merging Codes and Code Sets

It is absolutely essential that the codes in the data files and the codes in the codebook are in sync. If people are lazy, they won't do this and will forget what the codes mean when they come back to them later. This results in bad research, and is an absolute nightmare for researchers.

The solution I have in mind is to parse the codebook and the data files, identify codes that are in the data files but not in the codebook, and append them to a list so that the researcher can define them easily. An ideal command would be something like:

```
cam sync
```

By default, the number of codes that are undefined would be printed to the screen, and the codes would be appended to codebook in the proper format under the right section. If it is a content code, then an empty code will be appended to the list of content codes using the structure laid out in the example codebook.md. Same goes for context codes.

It should also check the code sets in the data files vs. the codebook. If there are differences, it should print them to the command line and ask if you want to (a) update the data files *and* the codebook by adding codes to code sets where there are differences (but *never* removing code sets), or (b) do nothing so that you can fix them manually.

In some cases it will be necessary to rename codes. This could just be a simple search and replace operation, but I think it would be better to be able to type

```
cam rename --code-firstCode --code-newCode
```

on the command line, which would find and replace "firstCode" with "newCode" in the codebook and *all* of the datafiles.

In other cases you will want to merge codes. This happens a **lot** in more inductive studies. The ideal command would be something like

```
cam merge --code-firstCode --code-secondCode --code-finalCode
```

Where the codebook and the data files are parsed, and every code listed on the command line is merged into the very last code listed. It should not matter if the merged code is in the codebook or not. Perhaps the entries for the codes that were merged could be commented out in the codebook so that it is clear that they are not in use anymore, but a record of them is still kept.

Searching Files, Printing Code Content and Counts

Most of the searching that people will do when they work can be done using the tools built into their text editor. Researchers that are comfortable on the command line can use tools like grep. But it would be good to have a simple way of printing the content of specific codes and code sets. If I want to see all of the text that has been coded with **'firstCode'** in any data file, then I could type something like:

```
cam extract --code-firstCode
```

and it will print the content to screen.

Most people will want this in a text file, which they can easily do since this is all on the command line...

```
cam extract --code-firstCode > summaries/firstCode.txt
```

Like grep, each snippet of text that prints should be preceded by the filename.

You should also be able to do this for specific files, as well as for code sets and file sets.

For example, for a specific file, something like:

```
cam extract --code-firstCode --file-example_transcript.md
```

would print the content of firstCode applied in that file.

If you wanted to get the the content of the codes in that are included in setOne, you could type something like:

```
cam extract --cset-setOne
```

If you want to see all of the codes from a code set in a specific file, something like:

```
cam extract --cset-setOne --file-example_transcript.md
```

What if you just want to see counts of codes? Or counts of codes in each code set? Something like:

```
cam count
```

should print a table of all the codes in the project in order of most commonly applied to least commonly applied. The number of times the code was applied will be reported, obviously. The results might look something like:

```
thirdCode   23
firstCode   11
fourthCode  10
secondCode  10
```

If you want to be a little more specific, a command like

```
cam count --cset-setTwo
```

would print a similar table, but *only* for the codes that are part of setTwo.

```
cam count --cset-setTwo --fset-fileSetOne
```

would print a similar table, but *only* for the codes that are part of setTwo and assigned in files that are part of fileSetOne.

Similarly, something like

```
cam count --fset-setOne
```

would print a list of the codes in the *fileset*, ranked by most common to least common, with the counts reported.

Something like

```
cam count --file-example_transcript.md
```

would produce a table reporting on all of the codes applied in the file, and how many times they appear.

Comparing Counts

Finally, what if you want to compare the counts of codes from two file sets? Let's imagine that the researcher has created a file set for women called fileSetThree and a file set for men called fileSetFour and wants to compare them. Something like:

```
cam table --fset-fileSetThree-fileSetFour --cset-setTwo
```

would print a table comparing *counts* of codes in the setTwo code set for the files in fileSetThree (women) and fileSetFour (men). Since there is more than one column of counts in the table, the rows should be sorted alphabetically.

I think it makes the most sense if the command assumes that the first thing specified (in this case specifying the file sets to include) will be the rows in the table, and the second thing specified (in this case the code set to include) will be columns.

This would be more intuitive if this was a real project, because you would just call those file sets 'men' and 'women' instead of my genetic fileSetOne etc.

Who Coded What?

What if you want to see all of the files that have been coded by a specific person? Or the instances of **'firstCode'** and **'thirdCode'** that were applied by a specific member of the research team?

One of the meta-data fields at the top of each data file is **'coderID'**. So you should be able to use that to select which content to print and what not to print. The command might look something like:

```
cam --cid-JVPM --code-firstCode-thirdCode
```

or if you want to look at a whole code set

```
cam --cid-JVPM --cset-setThree
```

Extracting Memos

When people are doing content analysis or qualitative analysis of interview transcripts, they write memos as they go. These could be any random notes, very rough preliminary analysis, concept formation, ideas for the next article, etc.

Typically these come up when people are working through text, so it is useful to be able to start writing a memo in the middle of a data file. This also takes removes the friction of starting a new file, or having another text file open that you have to toggle over to. Writing memos in the middle of a data file also makes it easier to go back and see the context that might have triggered an idea, etc.

Because these are markdown files, you can trigger the HTML code for a comment comment quickly and easily. It makes sense to use these for memos.

It might look something like this:

```
< !--
Memo: July 13, 2015

I have intentionally made an error so that this prints... ;)
 -- >
```

Obviously we want to be able to extract these memos and concatenate them into one file. If we use the init command I suggested earlier, there will be a subdirectory called memos, which will include a file call extracted_memos.md. A simple command like:

```
cam extract memos
```

would parse the data files and put all of the memos into extracted_memos.md. Obviously **'Memo: July 13, 2015'** would be a H1. If there are multiple memos like this, we could just group them together.

I would prefer it not to re-write the entire file every time, but instead to append new memos that were not there before. If someone edits a memo, it will show up twice, but that's not really a big deal. It's not like duplicating codes, it's mostly just a messy research journal with thoughts as they develop.

Being able to print memos from a specific file or file set would also be useful. Maybe:

```
cam extract memos --fset-fileSetTwo
```

Other Issues Related to Writing Data

The section above covers a lot of the printing needs. This section just includes a little more details on the other tables that are typically used to in content analysis, and for getting data into a format that can be easily used for more quantitative work or visualization in something like R, Python, or Stata.

Some of those other tables include:

in the rows	in the columns	in the cells
participant	code	counts
codeset	code	counts
fileset	code	counts
participant	codeset	counts
fileset	codeset	counts
code	code	count of co-occurrence

There are likely others that I don't realize that I have left out, but once these are created, it would (presumably) be easy enough to add new ones.

The commands to produce these tables should be similar to those I suggested earlier (i.e. cam table...).

As with the other commands, the default should be to print to screen. This is especially useful for work in progress because it doesn't produce a lot of temporary results cluttering up the directory. When you are ready to print something to a file, we can simply use pipes. When I write up the documentation, I will explain how that works for people who are not used to CLIs.

Writing Raw Data for Text Mining or New Coders

There are some circumstances where it will be necessary to work with raw data, for example if the researchers wants to use text mining and topic modeling packages in R, Python, Java, Stata, etc. Another use would be introducing a new coder to the project to determine inter-coder reliability rates. This requires coding the same files that another coder worked on while not being able to see their codes as you work. You need the raw files for this.

Hopefully the researcher will have kept raw data in a backup (as the subdirectory created by the init command suggests), but just in case, it would be helpful to have a command like:

```
cam raw
```

that would look for files in the *'data_raw/'* subdirectory. If there are data files from the coding directory that are not in there, it will add a copy of the file with all of the codes and memos stripped out.

In cases where you are doing text mining or topic modeling, you likely want to strip out the YAML-style meta-data as well. So adding an option for that would be very useful.

Scripts and Making it a Little Easier

I think this is all very simple and intuitive, and if the documentation is good enough, people will be able to do this just fine even if they have never seen a terminal in their life.

Still, there are a few little things that would *definitely* make things go a little smoother. My main idea here is to create a Sublime Text 3 package that would have syntax highlighting for the data files and could list and execute commands via the command pallet. My *main* interest in the ST3 package, though, is hooking into the codebook. The LaTeXing package hooks into a .bib file. When you start to type cite commands into your tex file, it opens up a search box where you can retrieve specific entries from your .bib file. When you search, find, and select an entry, LaTeXing completes the cite command with the appropriate cite key. Being able to do that with codes from the codebook would be *AMAZING*.

Timeline, etc.

That's all I've got for now. I know this is a lot, but I've been thinking about this on and off for about 4 years. I have a *lot* of grievances about the state of software for content analysis and qualitative analysis of unstructured data... ;)

The command suggestions are just that: suggestions. I don't know what the best options are just yet. The main thing is that they are simple and as minimal as possible.

If you are good with this and want to give it a shot, or at least get started on it, then I will setup a private repo on Github under my account. I'll work on documentation on and off as we go. Early in 2016 I will submit a short paper describing the design principles behind the tool and how to use it to a methods journal, or a journal like *Social Science Computer Review*. I'll add you as a co-author if you are OK with the manuscript, but I don't expect you to contribute writing unless you want to.

I don't know how long this will take you, but I think it makes sense to get started. If we can get it working OK by the end of the fall then I will use it in my Research Methods class instead of PyQDA or a trial version of ATLAS.ti. If you think you can get it done before then, by all means do so.

Looking forward to continuing to work with you on this.

John
Austin, TX
July 13, 2015