COW usage

Auke Rijpma, Ruben Schalk, and Richard Zijdeman

23 February 2017

Installation and activation

- Install either via pip or git+virtualenv.
- I recommend virtualenv because numpy
- If virtualenv, these would be the first steps to get COW running.

```
cd /users/auke/repos/wp4-converters/
source bin/activate
cd cow
```

- If using pip, the csvw-tool.py command should be available everywhere so life is easier
- Tradeoffs!

Cattle

- Web service:
- http://cattle.datalegend.net
- Upload csv to get json schema file.
- Modify json.
- Upload csv and json, get rdf!
- If you use this, ignore all the command line instruction below.

Build schema

- First time, build the schema
- Note the usage of the full path because we have to be in cow/cow to access the python script (referring to script using full path from another directory gives unexpected results).

```
python csvw-tool.py build /users/auke/repos/dataday/test.csv
```

- test.csv-metadata.json should now also exist!

Convert

- Use metadata to convert the csv into nquads

```
python csvw-tool.py convert /users/auke/repos/dataday/test.csv
```

- A wild nquads file appears!

```
ls /users/auke/repos/dataday
```

The output

- The data triples

```
head -3 /users/auke/repos/dataday/test.csv.nq
```

- The metadata triples

```
tail -4 /users/auke/repos/dataday/test.csv.nq
```

Base URI specification

```
python csvw-tool.py build /users/auke/repos/dataday/test.csv \\
--base=https://data.iisg.amsterdam/resource/test/
python csvw-tool.py convert /users/auke/repos/dataday/test.csv
```

- note: first specify in schema building, then conversion
- in future allow you to specify predicate prefixes besides base, currently bit inconsistent.
- note also that old schema has been backed up: -metadata.json.datespecification.
- This is nice and nothing to worry about. Useful if you accidentally build schema and overwrite your work.

Speed

- final note before we continue: everything we do here should happen relatively quickly because we're working with a very small file
- scales linearly with number of columnsXrows
- So on files larger than a few thousand lines, it starts to take a little while.
- When protyping use e.g. head to make a sample

```
head -2 /users/auke/repos/dataday/test.csv > /users/auke/repos/dataday/test2lines.csv
```

Speed

- but mind the fact that the metadata and the data have to have same name (except metadata and extension addition)
- easy fix is to first copy the original data to elsewhere, then copy a few lines back to the original folder with the same file name
- or better yet, create a custom sample in your stats program of choice, making sure all interesting cases are in there, and prototype json meta
- then use this on full file, keeping in mind stuff about file names

Modifying the json file.

- Overall idea is that you modify the json file to describe the csv-file and the rdf-representation you would like to achieve.
- The -metadata.json file consists of a number of blocks to do this.
- First few blocks are actual metadata:
    - *file encoding, delimiters*
    - *keywords*
    - *publisher (us)*
    - *base uri*
    - *rdf namespaces*
    - *tableSchema*
- Look at base first, then tableSchema, then rest of metadata

Base specification in the json file.

- The base is one of those things we can change in the json file.
- Alternative to using the –base parameter.
- Avoids all those backups.
- Done by changing

```
"@base": "https://data.iisg.amsterdam/resource/test/",
```

into ↓

```
"@base": "https://data.iisg.amsterdam/resource/supertest/",
```

- And convert again using csvw-tool (this step omitted from instructions from now on)

```
python csvw-tool.py convert /users/auke/repos/dataday/test.csv
```

overall aboutUrl

- The aboutUrl corresponds to the subject in RDF's subject-predicate-object representation of data.
- The metadata contains a statement about the global aboutUrl, specifying how the subject for each row if formed.
- Means same subject for each observation in one row.
- Data thus represented in RDF as $subject_{row1} - predicate_{col1} - object_{row1,col1}$ $subject_{row1} - predicate_{col2} - object_{row1,col2}$ $subject_{row1} - predicate_{col3} - object_{row1,col3}$
- This is a fairly efficient way of representing tabular data
- (Albert sent me a paper that hub-and-spoke representation fastest to query).
- That said, sometimes there are more direct links in the data (personID inHousehold housholdID) that you might want to represent.
- In short: efficient, if the table itself was an efficient representation of the data.

overall aboutUrl

- Overall aboutUrl is first line in tableSchema
- By default the row number.
- Sensible, because subject needs to uniquely identify the row.
- Bit dangerous, because row number and poorly chosen (identical to other dataset) base can cause subject clash.
- Take some time to consider base uri and subject construction.
- Here's how to change it so that we use Country as the subject.

```
"aboutUrl": "{_row}",
```

into ↓

```
"aboutUrl": "country/{Country}",
```

overall aboutUrl

```
"aboutUrl": "country/{Country}",
```

- Let’s break this down.
- We take the global base URI (if you say nothing, you get the global base specified earlier), add `country` and add to that the value from the column `Country` for this row.
- Use column content “as is” using `{}` and the column name.
- Subject now looks like this: `<https://iisg.amsterdam/resource/country/Ireland>`.
- Note that we can only do this safely because in this dataset country uniquely identifies observations (rows). (see above)

overall aboutUrl

- If countries did not uniquely identify the rows/observations, we'd have to make a more complex ID.
- This might be the case in data where we have annual observations for each country.
- Row numbers are pretty safe and mean you don't have to worry about uniqueness (with proper base URI).
- More complex one gives semi-interpretable subject names (identifying the unit of observation) which might be nice to have.

overall aboutUrl

- Here we paste together the `Country` and `Rank` variable.

```
"aboutUrl": "country/{{Country + Rank|string()}}",
```

- Breakdown: take base, add `/country/` then take Country column and concatenate with Rank cast as a string (string concatentation in python done with +).
- The transformation requires double `{{}}`. Will revisit in more detail below.
- String cast probably not necessary, but just to be sure. If you want to use column values as numbers, use usually have to cast to numeric using `float()` or `int()`.
- Will return to data transformations in-depth below.

The table columns

- Moving on to the rest of tableSchema, where each of the columns is specified.
- First choice is whether object (columns) should be a literal (default) or a URI.
- Rule of thumb: if something else also refers to this object, or if it in turn will refer to something else, a URI is appropriate (joins are faster on URIs than Literals).
- Or: finite collections (something of which there are not endless variants).
- Examples: IDs in relational databases, countries, municipalities, but not: surnames, first names, notes, etc.
- Or: things that have an obvious datatype: numbers, dates.
- Break these rules of thumb for compatibility with othet dataset. If for example a useful geographic dataset refers to country names as strings, you should too (or do both!).

Datatype

- If you choose the column values (objects) to be Literals, you'll have to specify the datatype.
- Default is `xsd:string`.
- Main alternatives are numbers
  - *`xsd:int` for integer that are always below 64k*
  - *`xsd:integer` for all integer*
  - *`xsd:float` for decimals*
- And dates:
  - *`xsd:date` for full dates (YYYY-MM-DD)*
  - *`xsd:gYear` for years (YYYY)*
- Many other options (search for "xsd datatypes"), but these are frequently used.
- xsd-prefix is optional, datatype is always assumed xsd.

Datatype

- Let's set the rank variable to be an int.

```
"datatype": "string",
```

into ↓

```
"datatype": "xsd:int",
```

propertyUrl

- `propertyUrl` maps to the predicates in the RDF s-p-o system.
- Important step: for cross-dataset querying to be easy, predicates need to be shared between datasets when possible.
- And this needs to happen consistently (if one dataset uses `prefix:age` and the other `prefix:Age`), we're not one step closer.
- If the values in the column need any work to be compatible (e.g. remove -99999 for missing values, change capitalisation), it is usually good to create a dataset-specific propertyUrl (just leave the default in place) and to create a new one at the same time in a "virtual" column (more about that below).

propertyUrl

- First the propertyUrl itself.
- By default the base followed by the colun name.
- Modify by adding a `propertyUrl` element to the column description.

propertyUrl

- So let’s change the propertyUrl for Country into one that’s not capitalised.

```
"propertyUrl": "country",
```

- Would use the global base specified earlier.

propertyUrl

- If you do not want to use the global base, add a prefix.
- Prefixes come from https://github.com/CLARIAH/COW/blob/master/cow/converter/util/namespaces.yaml.
- They're also in the basic json-file.
- Feel free to add namespaces to this file.
- Here we use the clio-infra one for country.
- For this we use the clio-predicate (from the predicate block) just like we did for the xsd-datatypes.

```
"propertyUrl": "clio:country",
```

The predicates should now look like `<http://iisg.amsterdam/clio/country>`.

valueUrl

- If the columns (objects in the s-p-o) system are not to be Literals, you need to turn them into URIs.
- Important that these are well-formed, because choosing them to be URIs usually means you'll be referring to them (in another dataset or the rdf-representation of the codebook).
  - *Usually we convert the dataset and the codebook separately (there should probably be a separate slide about this).*

valueUrl

- Done by adding a valueUrl element to the column description.
- You can only do this if you have specified the propertyUrl.
  - *Maybe a bug, but typically if you care this much about the valueUrl, you should also care enough about the propertyUrl.*
- Note that you have to refer to the column by the column-name and `{}`. Otherwise COW just thinks it's a word.
- Here we use the clio country prefix to (again not sure if this is how clio-infra exactly refers to countries).

```
"valueUrl": "clioctr:{Country}",
```

The objects now look like `<http://iisg.amsterdam/clio/country/Macau>`.

virtual columns

- Sometimes you want to have additional variables that are not a column.
- For example a combination of information from two columns to add extra information for querying convenience, such as birthyear from the year of observation and the age.
- Or you want to keep the original data as it is in the table, but also want to present transformed data, for example the original data with missing value-codes, but also new triples that can be used directly (provided you're happy with omitting missing data).

virtual columns

- Done by adding a full new column description with the additional `virtual` element.

```
{
 "virtual": true,
 "propertyUrl": "urirank",
 "valueUrl": "rank/{Rank}"
},
```

- Would add a new "column" (triples representing this column, anway) where the rank is not just an integer, but also URI.

column-specific aboutUrl

- In virtual columns you can also specify the aboutUrl (subject).
- This is not possible in regular columns (bug or feature: generally not wise to change the global aboutUrl).
- Virtual columns deal with special cases such as connecting the values of two columns, in which case this is useful.
- Done simply by adding an `aboutUrl` statement to a virtual column.
- So a row-number aboutUrl:

```
{
 "virtual": true,
 "aboutUrl": "rownumber/{_row}",
 ...
},
```

Would get you subjects like `<https://iisg.amsterdam/resource/rownumber/1>`.

Data transformations

- Often data in csv not ready to turn into RDF.
- Missing value codes, cases, number representations, etc.
- If possible, try to solve this in metadata-json to have provenance.
- COW allows you to do this with python functions and jinja2 templating.
- Double curly brace notation `{{}}` to tell COW that you want to take column name and do something special with it.
- Searching for "your problem" + "jinja2" will often get you an answer. Bit of trial and error also useful.
- See github and readthedocs for some commonly used functions.

Data transformations

- Example: string slice.
- Take first three characters of string with python string slices.

```
"valueUrl": "clioctr:{{Country[0:3]}}",
```

- You can chain these functions using |.

```
"valueUrl": "clioctr:{{Country[0:3]|upper}}",
```

Data transformations: literals

- Transforms in valueUrl create URIs.
- To transforms literals, use csvw:value.
- Example, replace the comma , (thousand separator) with nothing in the numbers.

```
"csvw:value": "{{Int|replace(',', '')}}",
```

Null

- Null allows you to exclude cells (not rows) from the rdf output.
- Simply specify the value(s) you want to exlude (in a list).
- Refers to the column in name/titles. Cannot refer to other column, that should be done with ifelse statement.
- These should all work, first two should give identical results.

```
"null": "Macau"
"null": ["Macau"]
"null": ["Macau", "Qatar"]
```

- So this would only work in the description of the column `Country`. If you want to refer to `Country` for another column, you'd use a conditional: {%if% …

Null

- COW automatically skips empty cells.
- Usually desired behaviour, but maybe you'd like to do something with the empty value.
- Use `csvw:parseOnEmpty` (default is false).

```
"csvw:parseOnEmpty": true
```

Language

- For string literals it can be good to add a language tag.
- Is this occupation in French, Dutch, English, etc.
- Simply add a `lang` element to a column block where the `datatype` is `string`.

```
"lang": "en",
```

- `"string"^^<http://wwww.w3...` now `"string"@en`.
- en for English, fr for French, nl for Dutch, etc.

More options.

- `collectionUrl` to place items as skos:concept in a skos:collection.
- `schemeUrl` to place items as skos:concept in a skos:scheme.
- Useful to do, but not essential. Makes data structure more complete.

Metadata

- Two things should be added to the metadata blocks (not the tableSchema):
  - *Who converted the data (you).*
  - *Where the original data comes from.*
- Easy to make mistakes here, but very generic.
- Just copy-paste from a complete one.

The author

- COW takes the converter to be the author.
- After `publisher` bit, add (very minimal):

```
"dc:author": {
     "rdf:type": [
       {
         "@id": "foaf:Person"
       },
       {
         "@id": "prov:Person"
       }
     ],
     "foaf:name": ["Auke Rijpma"],
       "foaf:mbox": {
           "@id": "mailto:auke@example.com"
       },
   },
```

Original dataset

- Done with `prov:wasDerivedFrom`.
- Again after publisher:

```
"prov:wasDerivedFrom": [{
    "@id": "http://www.imf.org/external/datamapper/PPPPC@WEO/THA"
},
```

- This is case of website, paper is more difficult.

Original dataset


```
"prov:wasDerivedFrom": [{
  "rdf:type": {
      "@id": "bibo:Article"
  },
  "dc:title": {
      "@value": "Building life course datasets from population registers by the Historical Sample of the Netherlands (HSN)",
      "@lang": "en"
  },
  "dc:author": ["Mandemakers, K."],
  "dc:publisher": "Edinburg UP",
  "dc:date": {"@value":"2006", "@type":"xsd:gYear"},
  "dc:isPartOf": ["http://www.euppublishing.com/toc/hac/14/1-2"]
 }],
```