CS 598 Data Curation

Anh Nguyen (anhn4)

**Final Project: Responses to Reflection Questions**

1. **Describe your process for canonicalization (i.e., decisions, actions, representation selection, attribute issues, provenance decisions). Report the checksum values after canonicalization.**

Simple pipeline: In this step, we employ a python script to perform the work. The same canonicalization rules are applied against both file A and file B. At the end, we can compare their checksum to confirm whether they are identical.

The following process is to be applied against both documents

*Step 1 - Read document using an XML Parser*: By using an XML parser, we effectively obtain the document intrinsic content. The XML parser has a few built-int features to easily choose (unify) UTF-8 encoding, expand internal DTDs if any (file B), and omit comment elements. We also exclude DTD when loading the document because final DTD will be determined after canonicalization.

|  |
| --- |
| from lxml import etree  parser = etree.XMLParser(  encoding='utf8',   attribute\_defaults=True,   remove\_comments=True,   load\_dtd=False) complaintsRoot = etree.parse(  file\_path, parser=parser).getroot() |

ComplaintRoot is a root node which represents the entire dataset as a tree. From here, we have an easy logical interface to work with. This is advantageous than working directly with content string

*Step 2 – Attribute string values cleanup*: Here we trim off excessive whitespace surrounding a node text. For paragraph-like field such as `consumerNarrative` and `publicResponse`, we normalize line-ends so that long text comparison is not affected by insignificant spaces.

|  |
| --- |
| def normalize\_line\_ends(txt: str) -> str:  pts = txt.split('\n')  pts = [ss.strip() for ss in pts]  return ' '.join(pts)  for elem in complaintsRoot.iter('\*'):  if elem.text:  #print(elem.text)  elem.text = elem.text.strip()  for elem in complaintsRoot.iter('consumerNarrative'):  elem.text = normalize\_line\_ends(elem.text) for elem in complaintsRoot.iter('publicResponse'):  elem.text = normalize\_line\_ends(elem.text) |

*Step 3 – Attributes sort and format*: For all elements, sort their attributes by tag name, and trim attribute values. By doing this, corresponding attributes are lined up appropriately for cross-reference elements comparison.

|  |
| --- |
| def trim\_and\_sort\_attrs(tt: etree) -> None:  for k in tt.attrib:  tt.attrib[k] = tt.attrib[k].strip()  dd = {}  for k in tt.attrib:  v = tt.attrib.pop(k)  dd[k] = v  for k in sorted(dd):  tt.attrib[k] = dd[k]  def sort\_child\_elements(tt: etree, key\_func) -> None:  if not tt[:]:  return  tt[:] = sorted(tt, key=key\_func)  for child in tt[:]:  sort\_child\_elements(child, key\_func)  pass  for anyTag in complaintsRoot.iter('\*'):  trim\_and\_sort\_attrs(anyTag) # sort elements at all levels by tag name alphabetically sort\_child\_elements(  complaintsRoot[:],   lambda x: x.tag) |

*Final indentation* - The last step is to conventionalize indentation. Doing this makes sure parent-child relationships are distinguished deterministically as the content is printed

|  |
| --- |
| # normalize indentation etree.indent(complaintsRoot) |

And obviously, write the XML content as string to an output file

|  |
| --- |
| # step: single encoding: UTF8 str = etree.tostring(  complaintsRoot,   encoding='utf8',   pretty\_print=True,   xml\_declaration=True) # return final xml outcome return str  f = open("final-file.xml", "w") f.write(strA.decode('utf8')) f.close() |

At this point, we may attempt to compare the 2 processed documents. We use <https://www.diffchecker.com/diff>, an online diff tool to check line by line difference between 2 arbitrary texts. The outcome is quite interesting. Here are observations over 2 datasets:

*Data observation 1*: In the old system (file A), a complaint has a child element named `submitted` which tells the channel (Web, Referral) via which customer feedback was sent. This maybe effectively the `submissionType` attribute in the new system (file B).

|  |  |
| --- | --- |
| **File A (canonicalized)** | **File B (canonicalized)** |
| <?xml version='1.0' encoding='utf8'?>  <consumerComplaints>  <complaint id="14038">  <company>  <companyName>U.S. BANCORP</companyName>  <companyState>AZ</companyState>  <companyZip>85008</companyZip>  </company>  <event date="2017-01-22" type="sentToCompany"/>  <event date="2017-01-17" type="received"/>  <issue>  <issueType>Loan servicing, payments, escrow account</issueType>  </issue>  <product>  <productType>Mortgage</productType>  <subproduct>Other mortgage</subproduct>  </product>  <response consumerDisputed="Y" timely="Y">  <responseType>Closed without relief</responseType>  </response>  <submitted via="Referral"/>  </complaint> | <?xml version='1.0' encoding='utf8'?>  <consumerComplaints>  <complaint id="14038" submissionType="Referral">  <company>  <companyName>U.S. BANCORP</companyName>  <companyState>AZ</companyState>  <companyZip>85008</companyZip>  </company>  <event date="2017-01-22" type="sentToCompany"/>  <event date="2017-01-17" type="received"/>  <issue>  <issueType>Loan servicing, payments, escrow account</issueType>  </issue>  <product>  <productType>Mortgage</productType>  <subproduct>Other mortgage</subproduct>  </product>  <response consumerDisputed="Y" timely="Y">  <responseType>Closed without relief</responseType>  </response>  </complaint> |

Basic normalization cannot justify this difference. On one hand, by educational guess, we can reason that `submitted` with `via` = Referral/Web is likely the legacy representation of a submission type, or channel. On the other hand, one could argue that “submitted” child element and “submissionType” attribute simply refer to two different things. Without understanding the origin of data, we can’t move forward. Under strict consideration, we can arguably fail the quality check right here and conclude that, the 2 datasets are not the same.

This is one of classic data issues, with this one in particular related to inconsistent schema. Data issues are expected to occur in any migration or canonicalization process.

Nevertheless, within the scope of this project, let’s hypothesize that we have a data team to consult with regard to any information integrity concerns. With their help, we confirm that `submitted` and `submissionType` in fact refer to the same concept: a channel via which a single customer feedback was received. They also acknowledge `submitted` was a poor chosen name for an identifier. It’s descriptive because it’s in the form of a passive verb. From data curation standpoint, a good identifier should be neutral and non-descriptive. Usually it should be a noun. Indeed, this goes along with the team decision which alters the name to `submissionType` in the new system. As also, because a complaint is practically sent under just 1 channel, they move this child element to becoming an attribute that belong to the <complaint> element itself.

Now we can move forward. We choose to normalize this concept to `submissionType` attribute as opposed to having it as a child element. We also add a validation to make sure all <complaint> elements in the old system should have a single <submitted> child. So, this overall step consists of:

* Removing `submitted` child element if there’s any
* Retain the `submitted` `via` value and set it as `submissionType` attribute on the complaint element

This step is performed just before the final indentation:

|  |
| --- |
| for complaint in complaintsRoot.iter('complaint'):  subs = [sub for sub in complaint.iter('submitted')]  if subs:  assert 1 == len(subs)  if 'via' in subs[0].attrib:  submType = subs[0].attrib['via']  complaint.attrib['submissionType'] = submType  # remove the <submitted /> element  complaint.remove(subs[0]) |

*Data observation 2*: It seemed there’re inconsistent representations of binary answer (N, No for No; Y, Yes for Yes).

|  |  |
| --- | --- |
| <product>  <productType>Debt collection</productType>  <subproduct>Medical</subproduct>  </product>  <response consumerDisputed="N" timely="N">  <responseType>Untimely response</responseType>  </response> | <product>  <productType>Debt collection</productType>  <subproduct>Medical</subproduct>  </product>  <response consumerDisputed="N" timely="no">  <responseType>Untimely response</responseType>  </response> |

This is another data issue. Again, in our hypothesized problem space, we speak to the data team and inform them of this mismatch. We resolve this issue by unifying the attribute values to a single form: Capital “N” for No and “Y” for Yes. This step is also run right before the final indentation:

|  |
| --- |
| def uniform\_yes\_no(tt: etree) -> None:  for k in tt.attrib:  v = tt.attrib[k]  if v.lower() == 'y' or v.lower() == 'yes':  tt.attrib[k] = 'Y'  elif v.lower() == 'n' or v.lower() == 'no':  tt.attrib[k] = 'N'  for complaint in complaintsRoot.iter('complaint'):  resp = [resp for resp in complaint.iter('response')]  assert 1 == len(resp)  resp = resp[0]  uniform\_yes\_no(resp) |

*Data observation 3*: Apparently, some attributes do not carry over to the new system. They are `complaint.submissionType` and `response.timely`

|  |  |
| --- | --- |
| <complaint id="837784" submissionType="Web">  <company>  <companyName>Navient Solutions, LLC</companyName>  <companyState>DE</companyState>  <companyZip>19802</companyZip>  </company>  <event date="2015-05-05" type="received"/>  <event date="2015-05-06" type="sentToCompany"/>  <issue>  <issueType>Dealing with my lender or service</issueType>  <subissue>Need information about my balance/terms</subissue>  </issue>  <product>  <productType>Student loan</productType>  <subproduct>non-federal student loan</subproduct>  </product>  <response consumerDisputed="N" timely="Y"> | <complaint id="837784">  <company>  <companyName>Navient Solutions, LLC</companyName>  <companyState>DE</companyState>  <companyZip>19802</companyZip>  </company>  <event date="2015-05-05" type="received"/>  <event date="2015-05-06" type="sentToCompany"/>  <issue>  <issueType>Dealing with my lender or service</issueType>  <subissue>Need information about my balance/terms</subissue>  </issue>  <product>  <productType>Student loan</productType>  <subproduct>non-federal student loan</subproduct>  </product>  <response consumerDisputed="N"> |

One more time, we have to ask the data team about what to do with this. Putting all the why questions aside, long story shorts, let’s assume they tell us to set default value for `complaint.submissionType` to “Web” and `response.timely` = “Y” if we detect these flags missing for any complaint in the new system. We implement their suggestion by declaring a couple of attribute default values in the DTD section of file B:

|  |
| --- |
| **File B** added DTD declarations |
| <!DOCTYPE consumerComplaints [  <!ENTITY redaction "XXXX">   <!ATTLIST complaint submissionType CDATA "Web">  <!ATTLIST response timely CDATA "Y"> ]> |

*To summarize, the whole workflow employs the following steps*:

* Step 1 - Read document using an XML parser
* Step 2 – Attribute string values cleanup
* Step 3 – Attribute sort and format
* Data observation 1 – Resolve `submissionType` variation
* Data observation 2 – Resolve Yes/No binary answer inconsistency
* Data observation 3 – DTD declaration: add default values for `complaint.submissionType` and `response.timely` in the new dataset (file B)
* Final indentation
* Write result to new XML file

The 3 data issues and observations are built upon the assumption that we understand the source of data. The normalization is carried out accordingly while this holds true. Data resolution is not part of generic canonicalization. They vary from problem to problem.

To test identity, we wrote a utility function to compare checksum of 2 documents after cleansing. We asserted that the outcome checksums are equal

|  |
| --- |
| def get\_checksum(ss: bytearray) -> str:  hash\_object = hashlib.md5(ss)  return hash\_object.hexdigest()  fA\_path = 'Consumer\_Complaints\_FileA.xml' fB\_path = 'Consumer\_Complaints\_FileB.xml'  strA = canonicalize\_utf8\_encoded(fA\_path) strB = canonicalize\_utf8\_encoded(fB\_path)  assert get\_checksum(strA) == get\_checksum(strB) |

We went even further by comparing their binary contents. The results are also confirmed to be exact equal

|  |
| --- |
| def binary\_compare(a: bytearray, b: bytearray) -> bool:  if len(a) != len(b):  return False  idx = 0  buf = 8  while idx < len(a):  ba = str(a[idx: buf])  bb = str(b[idx: buf])  if ba != bb:  return False  idx += buf  return True  is\_same = binary\_compare(strA, strB) assert is\_same |

At this point, we conclude that the 2 intrinsic contents are indeed identical.

1. **How does the way data is represented impact reproducibility?**

The detailed steps in section a showed that not only did we do ordinary canonicalization, we also made decision on how some attributes transform. To quickly summarize, they are: uniformizing representation of submissionType, singleton resolver for Boolean (yes/no) element, and setting default value for missing datum in the new system such as submissionType and response.timely. These data-oriented decisions definitely impact reproducibility of a small percentage of the overall dataset. At a quick glance, about less than 10% of complaints are subject to different presentation due to these fields. If we’d reproduce these customers from the old and new systems, the 2 results would not be exact equal.

To clarify, while the different schemas (old and new) affect reproducibility, the canonicalization step is consistent. Utterly, it always produces an unchanged view for either old or new inputs with respect to the same complaint.

1. **How may your canonicalization support the overarching goals of data curation (revisit objectives and activities of Week 1)?**

Collection/ Organization: With our canonicalization script, new data comes in with new formats is now easily translated to its legal form. This helps support efficient collection and acquisition of data while maintain the appropriate data model and standard.

Preservation/Discovery: As showed in this particular problem, an old system’s product is now processed and work well together with new system through the canonicalization process reached the goal of data curation to ensure data is understandable and will be usable in the future.

Identification/WorkFlow/Modification/Reformatting: the canonicalization script created a standard workflow making sure the data are identical, authentical and validated.

1. **Which additional curation activities would you recommend enhancing the data set for future discovery and use?**

In our memo to the new director, we discussed in-depth the adoption of data preservation, workflow and organization. There’re multitude of other curatorial best practices we can do in order to enhance the dataset.

Reproducibility is an important one to take part in. Via our data cleansing process, we’ve discovered non-equal presentation of some attributes between the old and new systems. We can implement a service to reproduce the same complaint in different formats, as well as document the types of difference. This helps ensure the validity of our data even when they are yielded from different systems.

Reformatting is a nice add-on to reproducibility. Specifically, in the reproduction service, we can have a feature converting a complaint line item from old schema to new one. This feature not only ensures data should agree to the new standard, but also helps recognize new type of inconsistency should they arise.

We can improve discoverability or shareability by allowing stakeholders to search for complaints in both new and legacy databases. Sometimes, it’s beneficial to know how the old data were persisted and what attributes of them were reformatted.

We might as well add an Identification layer on top of all complaints. This master identifier mechanism should successfully map a customer complaint to its various versions or formats. As the organization is big, different subsystems may require different versions to work with. Having a master identifier warehouse allow these subsystems to effectively overlap with each other.

There’re many other applicable activities: Upgrading to better storage for the growing data and versions; Advanced integration to allow constructing conceptual customer complaints from several persistent parts; Communication to allow tagging and improve self-explanatory metadata; Modification pipeline to enable editing a complaint version; Compliance to help track PI and PII user data; and Security to employ hierarchical-level access with regard to different attributes of customer complaints.