Contents

1 Knowing what we know now

The below documents my rough efforts. There doesn't appear to be any low-hanging fruit which isn't already mostly captured by the OBAPI demonstrator interface. Essentially, below is a hackneyed attempt to analyze the effort to duplicate the OBAPI searching interface, and potentially expand the listed features.

2 Initial sputter

So, after a bit of playing around with trying to get a single xml file parsed ("./sessionsPapers/18370102.xml"), for example working. It seems that BS4 is the best option (xmltodict is convinient, but mis-parses some items if deliniated in a particular manner (ie, a page delineator in xptr will break some other element)).

As per usual, the best approach is to stand on the shoulders of the past, so I've pulled some of the preprocessing scripts from here. (See getfiles.sh, which pulls code and data, and does a bit of local directory cleanup)

3 Narrowing down files of interest

Filenames appear to be in a `YYYYMMDD` format, which is mostly straightforward to manipulate programmatically. The following snippet is a series of functions which defines if a given pathname describes a file in the range of interest

```
import glob
import re
from datetime import datetime, date
from pathlib import Path
sessiondir = "./sessionsPapers/"
files = glob.glob(f"{sessiondir}/*.xml")

# Get files in correct interest period
# Some file stems can end with "A" or "E", so only slurp the number
↪  portion
numonly_pat = re.compile("([0-9]+)")
# Aquire filename without extension
normalize_path = lambda x: numonly_pat.match(Path(x).stem)[0]
# Get datetime object from filename
path_to_date = lambda x: datetime.strptime(normalize_path(x),
↪  "%Y%m%d").date()
# Make datetime object from year
year = lambda y: date(year=y, month=1, day=1)
# Return True if file is in period of interest
is_victorian = lambda x: path_to_date(x) >= year(1837) and
↪  path_to_date(x) <= year(1901)

# Apply functions to our list
victorian_files = sorted([d for d in files if is_victorian(d)])

f"Found {len(victorian_files)} matching (out of {len(files)})"
```

```
Found 769 matching (out of 2163)
```

4 A Rough Attempt at Adding Racial Features

As an incredibly naive approach to see how difficult (very) it would be to associate a given person/persname with some racial identifier, I took the following approach. If we scan the text for the term "negro", how many people can we see are related to this term?

First, we do a rough scan of the texts using `grep`:

```
#-i ignores case, -l only prints matching file names.
# See `man grep' for more details
grep -i -l "negro" ./sessionsPapers/* > foi.txt
```

Passing this list into our `is_victorian` filter:

```
with open("./foi.txt", 'r') as f:
    fns = f.readlines()
vic_sub = [f.strip() for f in fns if is_victorian(f)]
f"{len(vic_sub)} matches out of {len(fns)}"
```

```
27 matches out of 86
```

Editorializing: with only 27 matches, it's probably best to not pursue a programmatic approach further because that's a fairly manageable number on its own, but for it's own sake, let's see what kind of lifting would be involved:

`grep` will tell us the plaintext match, but let's see if we can get a better idea of the surrounding context programmatically:

```
from bs4 import BeautifulSoup
from itertools import chain
# Python's regular expression builder
npat = re.compile(r"negro")
def process_foi(fn):
    with open(fn, "r") as f:
```

```
        soup = BeautifulSoup(f.read(), 'lxml')
    # Get all tags which match npat
    foi_instances = soup.find_all(text=npat)
    return foi_instances
# We have a list of lists, flatten that using chain.
instances = list(chain.from_iterable([process_foi(l) for l in vic_sub]))

# Just looking at the first instance:
instances[0]
```

```
I think it was a week afterwardsthe prisoner was locked up in the
↪  workhouse that timehis father is a negro, and it also in the
↪  workhouse.
```

Hmmm. This is difficult to parse programatically (to whom is the "he" that "his father" refers?). Looking in the surrounding text

```
"\n".join(instances[0].parent.stripped_strings)
```

```
Cross-examined by
MR. PAYNE
.
Q.
How long afterwards did you go before the Justice?
A.
I think it was a week afterwardsthe prisoner was locked up in the
↪  workhouse that timehis father is a negro, and it also in the
↪  workhouse.
```

This is from a cross-examination of a witness, but even then a trivial program likely couldn't assemble reasonable comprehension of this fact.

Let's see if there are any matches which make our lives easy. We'll do this by examining the nested tag structure of each of the aforementioned matches, hoping that one such match

is directly inside a "person" tag, or at least something more descriptive than a blob of plaintext:

```
def parent_chain(res):
    # Collect results here
    parents = []
    el = res.parent
    # div1 is the parent tag of all trial accounts, so it's a cheap
    ↪  stopping place
    while el.name != "div1":
        parents.append(el.name)
        # Traverse upwards through tag chain
        el = el.parent
    return parents

cs = [parent_chain(instance) for instance in instances]
# Filter out results that are direct children of a p tagg/only have one
↪  parent
[c for c in cs if len(c) > 1]
```

hi	rs	p
persname	p	
hi	p	
hi	rs	p
hi	p	

Excellent, we seem to have three possibilities:

- hi: Highlighted/italicized text. Not terribly useful, because this typically indicates that the text is emphasized but doesn't contain any extra metadata
- rs: Referencing string. Pretty useful, it means that this text may have some related metadata
- persname: Yes, this contains a person's name, allowing us to directly identify people. (Good for naive "database search/cross-reference" with known sentencing details)

Let's check the persname

```
# Very lazy way to find: make a list, pick the one element
pni = [i for i in instances if "persname" in parent_chain(i)][0]
pni.parent
```

```
<persname id="def1-162-18900113" type="defendantName">
<interp inst="def1-162-18900113" type="gender" value="male"></interp>
<interp inst="def1-162-18900113" type="age" value="29"></interp>
<interp inst="def1-162-18900113" type="surname" value="HIGGINS"></interp>
<interp inst="def1-162-18900113" type="given" value="CHARLES"></interp>
<hi rend="largeCaps">CHARLES HIGGINS</hi>, a negro (29)</persname>
```

So, we have useful items here, this case can be further analyzed manually.

Let's check the other items, starting with the referencing strings:

```
pni = [i.parent for i in instances if "rs" in parent_chain(i)]
first, second = pni
first.parent
```

```
<rs id="t18860308-369-offence-1" type="offenceDescription">
<interp inst="t18860308-369-offence-1" type="offenceCategory"
↪  value="theft"></interp>
<interp inst="t18860308-369-offence-1" type="offenceSubcategory"
↪  value="simpleLarceny"></interp> (<hi rend="italic">a negro</hi>),
↪  Stealing a walking-stick, the property of <persname
↪  id="t18860308-name-216" type="victimName">
<interp inst="t18860308-name-216" type="gender" value="male"></interp>
<interp inst="t18860308-name-216" type="surname" value="DAVIS"></interp>
<interp inst="t18860308-name-216" type="given" value="HENRY
↪  ALBERT"></interp>
<join result="offenceVictim" targets="t18860308-369-offence-1
↪  t18860308-name-216" targorder="Y"></join>Henry Albert
↪  Davis</persname>.</rs>
```

So, the first "rs" match appears the be in an "offenceDescription". What makes this one tricky to parse for a trivial program is that once again, there needs to be some correlating information of the fragment description to the plaintiff of the current case.

Looking at the second "rs" match

```
second.parent
```

```
<rs id="t18970628-452-offence-1" type="offenceDescription">
<interp inst="t18970628-452-offence-1" type="offenceCategory"
↪  value="theft"></interp>
<interp inst="t18970628-452-offence-1" type="offenceSubcategory"
↪  value="simpleLarceny"></interp> (a <hi rend="italic">negro</hi>),
↪  Stealing a bag, a cigar case, a flask, and a cheque-book, the
↪  property of <persname id="t18970628-name-97" type="victimName">
<interp inst="t18970628-name-97" type="gender" value="male"></interp>
<interp inst="t18970628-name-97" type="surname" value="CLOWES"></interp>
<interp inst="t18970628-name-97" type="given" value="EDWARD
↪  ARNOTT"></interp>
<join result="offenceVictim" targets="t18970628-452-offence-1
↪  t18970628-name-97" targorder="Y"></join>Edward Arnott
↪  Clowes</persname>.</rs>
```

Looks to be the same case here.

5 Conclusion

As stated earlier, it's non-trivial (read: requires some proper Natural-Language-Processing) to "extract" additional features from the corpus and even then associate said features with named entities in the trial account.