Defactoring Pace of Change

Reviewing computational research in the digital humanities

Matt Burton
mcburton@pitt.edu
@mcburton

School of Computing and Information
University of Pittsburgh

Joris J. van Zundert
joris.van.zundert@huygens.knaw.nl

Huygens Institute for the History of the Netherlands
Royal Netherlands Academy of Arts and Sciences
Whoami?

- PhD in *Information*
- Visiting Assistant Professor
- Department of Information Culture and Data Stewards
- School of Computing and Information
- University of Pittsburgh
- The *iSchool!*
The Digital Humanities - Two sides of the same coin

Study Culture Digitally

Study Digital Culture
Ben Schmidt - Visualizing historical shipping logs

http://sappingattention.blogspot.com/2014/03/shipping-maps-and-how-states-see.html
Orbis - Google maps for ancient Rome

According to the Fastest routes from Antiochia to the rest of the Roman world in July, sites are this far away.

The most distant major sites are:
- Londinium (51 Days)
- Corduba (41 Days)
- Lugdunum (41 Days)
- Mediolanum (36 Days)

http://orbis.stanford.edu
Digital studies, coding as a literacy

CODING LITERACY
How Computer Programming Is Changing Writing

ANNETTE VEE
Digital Humanities & Jupyter - Great for teaching

```python
import pandas as pd
derp = pd.read_csv("http://toob")
```

# Heading 1
## Heading 2
* List item 1
* List item 2
Digital Humanists are Unicorns!

- Digital humanists can:
  - Write *good* prose
  - Communicate effectively
  - Talk to humans
  - Talk to machines
  - Cut code
  - Build systems
  - *Think critically!*

- Yack + Hack!
Digital humanists have written lots of code...

But where is it?
The invisible material labor of coding

- Code and coding are often not seen as scholarship, but theory speaks against this...
  - A ‘philosophy’ of ‘purification’, (Bruno Latour, Tanya Clement)
  - an artificial opposition between ‘intellectual’ and ‘material’ labour, (Burgess & Hamming)
  - obscure what is in essence scholarship through different literacies (Annette Vee, Joris van Zundert)

- Credit & Attribution
  - Who does most of the coding in your lab? Are they authors on your papers?
Code as Scholarship?

DATA

Analysis & Interpretation

Publication
Code (along with data) should be a part of the scholarly product.
But the cultures and infrastructures of knowledge production in the academy ignore or even occlude the *code-work* of scholarship.
Social + Technical = Sociotechnical
Defactoring - A road to peer reviewing code...

- Peer review of digital scholarship is...
  - mostly focused on promotion & tenure assessment (digital humanities)
  - difficult because of varied technical literacies (few literary historians who code)
  - caught in a catch 22 (are software engineering best practices appropriate?)
  - very much in need of some practical methods and techniques for reviewing actual code
Layers of Scholarly Software

- Konrad Hinsen’s software stack for computational science (2017)
- We focus on the *bespoke* code
  - NOT general purpose
  - The glue code
  - Notebook code
- Different set of standards from lower layers
  - Not software engineering
- This code should be peer reviewed as *scholarship*
Example - Pace of Change
Pace of Change

DATA
HathiTrust

CODE
Github

Preprint
Figshare

Publication
MLQ
Pace of Change

- **DATA**
  - HathiTrust

- **CODE**
  - Github

- **Preprint**
  - Figshare

- **Publication**
  - MLQ
Defactoring Pace of Change

- We took Underwood and Sellers’ code from *Pace of Change*
- Converted into a Jupyter Notebook
- Conducted a *close reading* of the code
- Created an linear, executable, computational narrative
- Human & machine readable
Defactoring - Annotating bespoke code

### DEFACTOERING FUNCTION

```python
# This function is used to get metadata from a given file.

get_metadata(file):
    ... This function will return a dictionary containing metadata.
```

The function `get_metadata` is used to retrieve metadata from a file. This metadata is then used to generate a dictionary that can be used to annotate code.

### Sorting Training Data

In [9]:
```python
all_instances = set([x for x in metadata['keys']])

# The next block of code filters the instances.

all_positives = set()

for key, value in metadata['items']:
    if value[category2sort[0]] == positive_class:
        all_positives.add(key)
```

This block of code sorts the instances in the metadata dictionary based on a given category and assigns them to a set of positive instances.

### Defactoring

In [10]:
```python
all_negatives = all_instances - all_positives

for item in iterator:
    if item in all_negatives:
        del all_negatives[item]
```

The negative labels are assigned to all instances that are not in the set of positive instances. This code removes any negative instances from the dictionary.

### Preprocessing

In [11]:
```python
if sizecap > 0 and len(all_positives) < sizecap:
    positives = random.sample(all_positives, sizecap)
else:
    positives = list(all_positives)
```

This block of code selects a random sample of positive instances if the size is greater than zero. Otherwise, it uses the list of all positive instances.

### Model Training

In [30]:
```python
def model_one_volume(data_tuple):
    ... This function trains a model on the given data.
```

This function trains a model on the given data. It first prepares the data, then splits it into training and test sets, and finally trains a logistic regression model on the training set.

In many respects, this is the most important block of code in the entire document. The code above actually runs the logistic regression and does the magical, algorithmic work that generates a prediction about each individual volume. This function builds upon the previous two functions to assemble a normalized set of training data (`trainset`) distinct from the single volume to be predicted (`testset`).

There are three lines of code involved in the computational modeling of the data. First, Underwood an Sellers instantiate a model object with the regularization parameter (more on that below):

```python
newmodel = LogisticRegression(C = regularization)
```

Then they “fit” the model using the normalized training data:

```python
newmodel.fit(trainset, yvals)
```

Once a model has been “fit” to the data it can use that model to make predictions about unseen or held-out data. This is what they do with the `predict_proba` function in this line:

```python
prediction = newmodel.predict_proba(testset.reshape(1, -1))[0][1]
```

Those three lines are all it takes to do the computational part of the analysis, the rest of the code up until this point has all been data preparation, cleaning, normalization, and re-shaping. This ratio of analytical to preparatory code is interesting and indicates that machines are taking our jobs are greatly exaggerated.

### Regularization and Logistic Regression

In this section, we’ll discuss the principles behind regularization and how they apply to the Logistic Regression model.

---

The code provided above demonstrates the process of defactoring code by annotating it with metadata. This approach helps in understanding and managing complex codebases more effectively.
Diving into the Code

The code expressed below has nine steps:

- **Setting Parameters** - Specifies parameters for the loading, cleaning, and labeling of data as well as sets conditions for the logistic regression.
- **Preparing Metadata** - Generates a list of *.tsv files from the poems/ directory.
  - **Cleaning Metadata** - Loads the metadata file, poemmetadata.csv and performs some cleaning of the metadata to make labeling easier.
  - **Sorting Data** - Sort the volumes into two bins, reviewed and not reviewed using the cleaned metadata.
- **Transforming Words into Features** - Identifies the 3,200 most common words in the corpus. Those most common words will be the features for the regression.
  - **Filtering Authors** - Removes poems by authors who have been reviewed.
  - **Filtering Words** - Remove any words from the poem data that are not in the most-common feature list.
- **Training Predictive Models** - Run a separate logistic regression for each volume, using a single volume as held-out data and measure each model's predictive power.
- **Modeling Coefficients** - Run a single logistic regression over all the data to inspect the salient coefficients.
- **Plotting Results** - Generate a plot showing the accuracy of the predictive models.
Interrogate the computational narrative

The authors are selecting their list of features (stored in the vocablist variable) by selecting words ranked by the number of documents in which they appear. The total number of documents we are working with is 720, so the table we generated above tells us that the top ten words appear in all of the documents. If we look at more than just the top ten, we can start to see the distribution of words in documents.

```
In [17]: # DEFACOndefalcong INSPECTION
plt.style.use('ggplot')
pd.Series([x[1] for x in wordcounts.most_common(n=3200)]).hist(bins=72, figsize=(12,2))
```

The plot above shows a histogram of the top 3,200 words and how they are expressed across corpus. The spike on the right end of this chart shows the 18 words that appear in all 720 documents (as we can see in the text table above). As a whole, most of the higher bars are on the left side of the chart indicating most of the words appear in fewer documents. In the course of data preparation it is important to inspect the shape of our data, in this case to visualize the distribution of our features. While it may not have a direct impact upon our analysis, it does provide a deeper perspective on the various transformations of our data. The chart above looks a lot different than the word frequency files of the poetry volumes at the start (or even of the original poems themselves).
Bespoke code is the engine of intermediate representations

In [14]:

# make a vocabulary list and a volsize dict
wordcounts = Counter()

volspresent = list()
orderedIDs = list()

positivecounts = dict()
negativecounts = dict()

for valid, volpath in zip(volumeIDs, volumepaths):
    if valid not in IDsToUse:
        continue
    else:
        volspresent.append((valid, volpath))
        orderedIDs.append(valid)

date = infer_date(metadata[valid], datatype)
if date < pastthresh or date > futurethresh:
    continue
else:
    with open(volpath, encoding = 'utf-8') as f:
        for line in f:
            fields = line.strip().split(' \'"
            if len(fields) > 2 or len(fields) < 2:
                # print(line)
                continue
            word = fields[0]
            if len(word) > 0 and word[0].isalpha():
                count = int(fields[1])
                wordcounts[word] += 1
                # for initial feature selection we use the number of
                # *documents* that contain a given word,
                # so it's just +=1.

vocabulary = [x[0] for x in wordcounts.most_common(numfeatures)]
Decisions made in the code

```python
if nation == 'ca':
    nation = 'us'
elif nation == 'ir':
    nation = 'uk'

# I hope none of my Canadian or Irish friends notice this.
```
Defactoring ‘Pace of Change’

Exploring Code Review Methods for Textual Scholarship and Literary Studies

Introduction

Code, the symbolic representation of computer instructions that drives software, has always been part of research methods in literary scholarship. While it is a tried and tested way to point to the work of Bana and his competitors at IBM (Janes 2016, Rhyner 2014) as foundational work in this regard, it was a very early and important application of computational methods as a means of analyzing literature. In no examples we first retrieved & Tangherlini (2012) who calculated genealogies for a large corpora of data, and Pager (2011) who sought linguistic identifiers for “compared” reading. In no tests were linguistic witnesses of the shifting continuities. When are we lucky we find them the enabling such feats of computational literary analysis, like Scott Enderlin (2018) who crucially contributed to a methodological discussion about non-linear “fundamental narrative arcs” from sentiment data in works of fiction or Ted Underwood and Jordan Sanders” “How Quickly Do Ubers Standards Change?” (2015), which used logistic regression to examine 25th-century poetry texts.

Unfortunately, much of the code used in the long history of humanities computing or the recent abundance in digital humanities has not been adequately recognized for its importance in the production of knowledge.

In this paper we argue code cannot be simply regarded as the intermi-stages of research must be treated as a first-order element of scholarly products. The increased application of code as a means to create digital cultural artefacts and as a mechanism for analysis within the human warrants the necessity to elevate code out of the invisible research process and into visible research outputs. Coding and code represent scholarly labour that forms part of the production of knowledge. However, the current system of scholarly publishing and communication all but ignores the code of computational work—although it is a component of the discourse, the code itself is not presented as part of the discourse. We posit its argumentative role in the evolving epistemology of literary studies (and the humanities in general) warrants a more overt inclusion of code in the scholarly discourse.

The current systems of scholarly communication are not tailored very well to such inclusion. In

https://github.com/interedition/paceofchange/
Discussion

Narratives