Extracted Features Use Cases and Examples

This page lists examples of what can be done with the Extracted Features 2.0 Dataset. For examples of how scholars are using it, see also Extracted Features in the Wild.

- Introduction
- Use Case 1: Using EF to identify poetry and prose volumes
- Use case 2: Using EF to identify poetry within a mixed volume

Introduction

The Extracted Features (EF) 2.0 Dataset contains structured information about each volume in HathiTrust. This page, and the accompanying notebooks (in Jupyter and Google Colab formats) present example use cases for using the files, and making use of the different levels of data contained in the EF Dataset.

For more information about the EF 2.0 Dataset, see the following pages:

- Dataset documentation
- Downloading the dataset
- Basic walk through an EF file

These use cases are also available in interactive Python notebooks, if you’d like to run or edit the code:

- Jupyter notebook
- Google Colaboratory notebook (make a copy of the notebook and add to your Drive or choose "Open in playground" to edit and save)

Use Case 1: Using EF to identify poetry and prose volumes

Before running an algorithm for text analysis, it's important to understand the data being studied. For traditional research that relies on a small number of books, this step is covered when reading the material directly. For computational analysis, where often the focus is sets of hundreds or thousands of books (what HTRC calls "worksets"), this is not achieved so easily. As a result, exploring and understanding our data becomes an explicit step in the research process. This is often the first step to computational analysis, but can present a number of challenges. For instance, if we were interested in studying poetry through traditional means, we could simply visit our local library, go to the poetry section and start reading. If we wanted to see if there is poetry in volumes that are not shelved in this section, we could open them and see if we find poetry or prose. But when working with hundreds or thousands of digital items, we do not have the ability to practically do either of these things. Some volumes we want to study may be under copyright, which means we can’t read them without purchasing. Others may have incorrect metadata records attached to them, which means we may not even know to look for them as poetry. Perhaps most challenging of all, it would take lifetimes to manually read thousands of books.

This is where the Extracted Features 2.0 Dataset can come in handy. Since the dataset is non-consumptive, we can use the data, without restriction, even to study volumes we cannot read due to copyright. Additionally, the dataset is heavily structured, removing the need for users to replicate common data analysis tasks like part-of-speech tagging, or page section separation. This structure also lets us quickly explore thousands of volumes, and do so relatively quickly.

Returning to our example: suppose we are interested in studying large-scale trends in poetry. Our first step would be to assemble a workset of poetry in order to answer our research questions. With a massive digital library like HathiTrust, finding all of the items we are interested in is no easy task. However, we can use the EF 2.0 Dataset to assess if there are structural characteristics of texts that we know are correctly classified as poetry, and potentially develop a generalized method of identifying poetry based on these characteristics. We’ll be doing exactly this using two unidentified volumes by Ursula K. Le Guin, who published both prose and poetry.

To start, we’ll import Pandas, a very common data science library in Python, and the HTRC FeatureReader, a library written by HTRC to ease the use of the Extracted Features Dataset:

```python
from htrc_features import FeatureReader, Volume
import pandas as pd
```

Next, we'll use both volumes' HathiTrust IDs, their permanent, unique identifier in HathiTrust, to download their EF files and create FeatureReader Volume objects—which allow us to take advantage of volume-level data in the files—for each of our volumes of interest. A Volume in FeatureReader has specific properties and built-in methods that let us do a number of common tasks. We can create Volumes simply by using Volume() and adding the HathiTrust ID (HTID) in the parentheses:

```python
leguin_v1 = Volume('mdp.39015000639800')
print(f'\{leguin_v1.author}. \{leguin_v1.pub_date}. \{leguin_v1.id\}"
leguin_v2 = Volume('mdp.39015052467530')
print(f'\{leguin_v2.author}. \{leguin_v2.pub_date}. \{leguin_v2.id\}"

['Le Guin, Ursula K. 1929- '], 1981, mdp.39015000639800
['Le Guin, Ursula K. 1929- '], 1969, mdp.39015052467530
```

After running the above code, the print statements will indicate that we have successfully created Volume objects for each work by Le Guin.

For this example, we are using the FeatureReader to download the files into the memory of our Python session. This works well for small worksets, but we should only use this method for <100 volumes, or else it will become less efficient and practical.
We'll be focusing on one particular field of features in the EF Dataset: begin-line characters. This is because we're working on the assumption that a page of poetry is likely to contain more capitalized letters at the beginning of each line on a page, a common poetry convention that Le Guin is known to have followed. To start, we'll create a Pandas DataFrame with the first character of each line (begin_line_char) by page, for both of our volumes:

```python
1 leguin_v1_blc = leguin_v1.begin_line_chars()
2 leguin_v2_blc = leguin_v2.begin_line_chars()
3
4 leguin_v1_blc.head()
5 # leguin_v2_blc.head()
```

<table>
<thead>
<tr>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>page</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

With this DataFrame, we have every begin-line character (BLC, for brevity) from our volumes, their counts by page, which section of the page they occur in (header, body, footer) and the page sequence number on which they occur. FeatureReader makes getting this data very easy (only 2 lines of code!), and retrieves it in a format that lets us generate counts by page with a simple `for` loop:

```python
1 leguin_v1_blc_up = 0 # a variable that will count uppercase BLCs
2 leguin_v1_blc_low = 0 # a variable that will count lowercase BLCs
3
4 # iterate through rows of our BLC DataFrame
5 for idx, count in leguin_v1_blc.iterrows():
6    blc = idx[3] # assign third index, the character, to variable
7    if blc.isupper(): # check if the BLC is uppercase
8       leguin_v1_blc_up += 1 # if uppercase, add 1 to uppercase BLC
9    else:
10       leguin_v1_blc_low += 1 # if not, add 1 to lowercase BLC
11 # The next section has print statements that will report our results
12 print(f"Found {leguin_v1_blc_up} capitalized begin line characters")
13 print(f"Found {leguin_v1_blc_low} lowercase begin line characters")
14 print(f"Found (leguin_v1_blc_up + leguin_v1_blc_low) total begin line characters")
15
16 # using counts to generate percentage
17 total_char = len(leguin_v1_blc['count'])
18 leguin_v1_caps_pct = leguin_v1_blc_up / total_char
19
20 # printing results, with rounded percentage
21 print(f"Uppercase characters make up {{:.2f}}.format(leguin_v1_caps_pct)) of begin line characters")
```

Found 190 capitalized begin line characters
Found 389 lowercase begin line characters
Found 779 total begin line characters

Uppercase characters make up 50.06% of begin line characters

For our first volume of Le Guin's, over 50% of all BLCs are capitalized. That seems like a substantial amount, but it's hard to know for sure without comparing our other volume. Let's repeat the code for Volume 2 and see what we find:
# code in this cell is identical to the code above, save for changed labels
leguin_v2_blc_up = 0
leguin_v2_blc_low = 0

for idx, count in leguin_v2_blc.iterrows():
    # print(count)
    blc = idx[3]
    # print(blc)
    if blc.isupper() == True:
        leguin_v2_blc_up += 1
    else:
        leguin_v2_blc_low += 1

print("Found \{leguin_v2_blc_up\} capitalized begin line characters")
print("Found \{leguin_v2_blc_low\} lowercase begin line characters")
print("Found \{leguin_v2_blc_up + leguin_v2_blc_low\} total begin line characters")

total_char = len(leguin_v2_blc[\'count\'])
leguin_v2_caps_pct = leguin_v2_blc_up / total_char
print("Uppercase characters make up \{\{1.2\}\}.format(leguin_v2_caps_pct) of begin line characters")

Found 1081 capitalized begin line characters
Found 3520 lowercase begin line characters
Found 4601 total begin line characters

Uppercase characters make up 23.49% of begin line characters

From 50% of BLCs capitalized to 23.5% is a steep reduction. It now seems likely that our first volume is poetry and the second likely to be prose. This process is still part of our exploratory data analysis phase, so we should do our best to verify these results, if possible. Since both volumes came from HathiTrust, we have bibliographic metadata for each volume, and can access certain fields within FeatureReader. Let’s see what we can find out by retrieving the titles of our mystery volumes, along with a link back to the item in HathiTrust, in case we want to take a closer look:

Success! leguin_v1 makes verification easy by including "and other poems" in its title. We could also use the returned URL to examine the volumes' bibliographic record and see if there is a marker for poetry. (If these volumes were in the public domain, we could also get their handle URLs using leguin_v1.handle_url and examine the texts in the HathiTrust Digital Library.)

Use case 2: Using EF to identify poetry within a mixed volume

We've shown above that, given two volumes from a single author, we can use the page-level metadata in Extracted Features (EF) 2.0 Dataset files to identify which is poetry and which is prose. But what about a single volume with mixed poetry and prose included? A similar workflow should be successful, but let's try.

In this example we’ll generate a benchmark value of tokens-per-page, based on the mean number of tokens per page in volume 142 of an anthology of Harper's Magazine covering issues from 1920 and 1921. We’ll then replicate the begin line characters analysis we did for Case Study 1, above, but modified slightly to delineate pages that are suspected poetry from suspected prose. Our results will be pages that appear as suspected poetry by both metrics, which we'll then do our best to manually verify.

But first, we'll import the Python libraries we need, and then we'll load our volume of Harper's as a FeatureReader Volume:

```python
from htrc_features import FeatureReader, Volume
import pandas as pd

# creating a FeatureReader Volume from our issue of Harper's
harpers = Volume('coo.31924054824473')

# printing info about the volume, available via FeatureReader methods
print(f"{harpers.title}, {harpers.enumeration_chronology}, {harpers.id}")

Harper's magazine., v.142 1920/21, coo.31924054824473

We now have a FeatureReader Volume from our anthology of Harper's, and we can take advantage of the methods that the FeatureReader includes, one of which is tokens_per_page(), which will return a sparse DataFrame with (you guessed it) tokens by page. This DataFrame will also allow us to identify the mean number of tokens per page in this volume, which we'll use for our benchmarking later on:
```
Next, we'll use the mean per-page token count to calculate a benchmark figure of one standard deviation below the mean that we will use to identify suspected pages of poetry. Pandas allows you to get this number very easily for a given column (what Pandas calls a "series") of numerical data using `std()`. We'll then subtract the standard deviation from the mean to get a benchmark number of tokens that we will use to compare each page of the volume:

```
242.46642446133332
376.9547876597876
```

The benchmark number of tokens we'll be using is ~377 for a given page. However, this is imprecise and our project is exploratory, which means we don't yet really know what the most accurate benchmark for poetry in this volume is. As a result, we will be considering pages that have a total number of tokens within 100 of this number, above or below, as possible pages of poetry. This will cast a wide net in the hopes of not missing any possible pages of poetry, or maximize recall. This works for our task because (as you'll see later) we are also considering begin line characters in conjunction with tokens per page, which helps reduce the wide net to the most likely candidate pages. For any given project, these types of decisions will need to be made by the researcher, taking into account their preferences, understanding of the data, and chosen methodological approach.

In the next code cell, we'll be using our benchmark token count to gather a preliminary list of suspected poetry pages by iterating through our tokens-per-page DataFrame `h_tokens` and recording rows where the token count is within 100 tokens of our benchmark:

```
"""Found 147 suspected pages of poetry"
"""Found 443 other pages"
```

The code above creates first-pass lists of pages that contain possible poetry and possible prose, and adds the page numbers to the appropriate list. We've identified these pages by using a benchmark that's really just an estimate, and should only be used to identify poetry in conjunction with other analysis or as a first-pass method for weeding out pages that are unlikely to contain traditionally-structured poetry. This method alone is not enough for our use case, so we must combine it with further analysis. We'll do this by looking at the percentage of begin-line characters (letters) that are capitalized, and see if we can find a trend.

We'll be re-using the general code and workflow from Case Study 1, where we looked at begin-line characters (BLCs) with some modifications:
Now we can iterate through our BLC DataFrame and track how many characters are uppercase, lowercase, and non-alphabetical on each page. This is different to our first example use case, as we are interested in both the numbers of BLCs in each category and the page on which they appear. Otherwise, this is a very similar workflow to what we did in the first example. The big difference is that we will be creating and working with Python dictionaries rather than lists. Unlike lists, dictionaries store key:value pairs, which means you can store multiple data points within a single entry in the dictionary. Rather than storing one list of page numbers and one list of BLC counts for each page, we can use a single dictionary to store both. This allows us to track BLC types by page and organize them for easy comparison later on.

```python
h_blc = harpers.begin_line_chars()  # make new DataFrame of BLC for issue of Harper's
h_blc = h_blc.reset_index()  # flatten DataFrame for clarity in code
h_blc.head()  # look at first 5 rows of new DataFrame
```

<table>
<thead>
<tr>
<th>page</th>
<th>section</th>
<th>place</th>
<th>char</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>body</td>
<td>begin</td>
<td>{</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>body</td>
<td>begin</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>body</td>
<td>begin</td>
<td>H</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>body</td>
<td>begin</td>
<td>I</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>body</td>
<td>begin</td>
<td>J</td>
</tr>
</tbody>
</table>

Now we can iterate through our BLC DataFrame and track how many characters are uppercase, lowercase, and non-alphabetical on each page. This is different to our first example use case, as we are interested in both the numbers of BLCs in each category and the page on which they appear. Otherwise, this is a very similar workflow to what we did in the first example. The big difference is that we will be creating and working with Python dictionaries rather than lists. Unlike lists, dictionaries store key:value pairs, which means you can store multiple data points within a single entry in the dictionary. Rather than storing one list of page numbers and one list of BLC counts for each page, we can use a single dictionary to store both. This allows us to track BLC types by page and organize them for easy comparison later on.

```python
h_blc_up = {}  # new dictionary for uppercase BLCs
h_blc_low = {}  # new dictionary for lowercase BLCs
h_blc_nonalpha = {}  # new dictionary for non-alpha BLCs

# iterate through DataFrame, return indices and row
for i, r in h_blc.iterrows():
    page = r[0]  # assign each column of DataFrame to clear variables
    section = r[1]
    token_place = r[2]
    char = r[3]
    count = r[4]
    # use Python method to check if char is alphabetic. If so, continue down this loop:
    if char.isalpha() == True:
        # if BLC is upper, continue down loop:
        if char.isupper() == True:
            # check if dictionary entry for current page exists
            if page not in h_blc_up:
                h_blc_up[page] = 1  # add new dictionary entry for page
            else:
                h_blc_up[page] += 1  # update existing dictionary entry
        # if BLC is not upper, continue down this loop:
        else:
            if page not in h_blc_low:
                h_blc_low[page] = 1
            else:
                h_blc_low[page] += 1
    # if BLC not alphabetic, continue down this loop:
    else:
        if page not in h_blc_nonalpha:
            h_blc_nonalpha[page] = 1
        else:
            h_blc_nonalpha[page] += 1

# printing results
print(f"Found {len(h_blc_up)} uppercase BLC")
print(f"Found {len(h_blc_low)} lowercase BLC")
print(f"Found {len(h_blc_nonalpha)} non-alpha BLC")

Found 977 uppercase BLC
Found 956 lowercase BLC
Found 871 non-alpha BLC
```

We now have three different dictionaries that contain BLC counts by type for each page. Since they share keys, we can make a new DataFrame with each page sequence as a row, and values for each type of begin line character as columns. This will let us compare BLCs of all three types, by page, in a single DataFrame. To do this, we’ll create a DataFrame from the dictionaries, rename the columns, transpose its axes to move pages to rows (which allows for simpler code for iterating through the DataFrame), and then fill all NaN (Not a Number) values, which are a result from pages with no lower /upper/non-alpha begin line characters, with 0 in order to compare them numerically to the other values:
Now we can go through the rows and track which category of begin line characters (BLCs) is the most common on each page, adding the page number to a list of possible pages containing poetry, as appropriate. For this example, we're only interested in pages of possible poetry, so we won't track pages with more lowercase or non-alpha BLCs, only those where uppercase characters are most common. This could easily be adjusted in our code by adding lists for the other two categories of BLCs, and adding .append() statements under each loop.

<table>
<thead>
<tr>
<th>non-alpha</th>
<th>lowercase</th>
<th>uppercase</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.0</td>
<td>5.0</td>
</tr>
<tr>
<td>11.0</td>
<td>15.0</td>
<td>7.0</td>
</tr>
<tr>
<td>2.0</td>
<td>0.0</td>
<td>5.0</td>
</tr>
<tr>
<td>7.0</td>
<td>9.0</td>
<td>16.0</td>
</tr>
<tr>
<td>8.0</td>
<td>6.0</td>
<td>21.0</td>
</tr>
</tbody>
</table>

Now we can go through the rows and track which category of begin line characters (BLCs) is the most common on each page, adding the page number to a list of possible pages containing poetry, as appropriate. For this example, we're only interested in pages of possible poetry, so we won't track pages with more lowercase or non-alpha BLCs, only those where uppercase characters are most common. This could easily be adjusted in our code by adding lists for the other two categories of BLCs, and adding .append() statements under each loop.

With two lists of potential pages of poetry—one based on token counts by page (poetry_pages) and one based on capital letters making up the plurality of begin line characters (blc_poss_poetry)—we can look for page numbers that appear in both lists. These pages are the most likely candidates to be poetry instead of prose, as they've been flagged by their relatively low number of tokens and high number of capitalized BLCs. We can find page numbers in both lists by using one simple piece of code:
From 990 total pages in our volume of *Harper's Magazine*, we now have 17 pages that are strong candidates to contain poetry! However, it's important to be reminded (yes, again!) that this use case is showing an exploratory workflow, so it's best to verify our results, and if applicable, improve our process and code to maximize accuracy based on our data.

This anthology of *Harper's* is in the public domain, so we can directly view the pages we expect to be poetry and see what we find. To make this easier to do in our Jupyter notebook, we'll use the URL to the first page of the volume in HathiTrust's PageTurner as a base (https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=), and then add a sequence number at the end with our possible page numbers in order to print URLs for each page:

```
# assign a root URL to use to generate URLs for each page
harpers_url = 'https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq='

# loop through list of pages and add the page number to the end of our root URL
for page in sorted(poeetry):
    page_url = harpers_url + str(page)
    print(page_url)
```

https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=79
https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=81
https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=82
https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=242
https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=307
https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=350
https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=416
https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=691
https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=884
https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=943
https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=955
https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=958
https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=966
https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=969
https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=977

Our (clickable) final list of pages of suspected poetry are:

- [https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=79](https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=79)
- [https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=81](https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=81)
- [https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=82](https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=82)
- [https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=242](https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=242)
- [https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=307](https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=307)
- [https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=350](https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=350)
- [https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=416](https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=416)
- [https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=691](https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=691)
- [https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=884](https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=884)
- [https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=943](https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=943)
- [https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=955](https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=955)
- [https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=958](https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=958)
- [https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=966](https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=966)
- [https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=969](https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=969)
- [https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=977](https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=977)
The results are pretty accurate! There are some false positives, as there are pictures and ads in Harper's which can skew our method, though these are relatively unique features in comparison with most volumes. But, we did find valid poetry, including the Robert Frost poem “Fire and Ice” on page 79 (https://babel.hathitrust.org/cgi/pt?id=coo.31924054824473&view=1up&seq=79). These results also help to remind us that any general text analysis workflow will require customization based on knowledge of our target data—both what we hope to find and the pool of data in which we hope to find it!