Final Project: Computational Approaches to Fanfiction

Edward Hernández
College of William & Mary

Final Project: Computational Approaches to Fanfiction

Contents

Questions

Answers

The Program

This document is a literate Python program inset within a LaTeX document, woven and executed using Pweave and typeset using pdfTeX This code is written in/for Python 3, and is intended for and tested under only version 3.5.1. It uses nltk to analyze corpora built using Scrapy (≥1.1.0rc1).[1]

Resolving the dependencies for this code is non-trivial. Ideally, pip should be able to handle the dependencies:

```
pip3 install               \
        bs4                \
        chatterbot         \
        python-Levenshtein \
        nltk               \
        vaderSentiment     \
        scrapy==1.1.0rc1
```

There are some issues with using pip, however. If you are using an operating system with a package manager, you may wish to allow that package manager to handle the packages which constitute your Python environment. If this is your preference, this code presents an unusual complication. It relies on both bleeding-edge versions of several Python packages (likely to be ahead of the stable versions packaged for your OS) and on some Python modules that are not packaged for most operating systems.[2] As a result, you may not want to attempt to alter your Python installation to accomodate this program. Additionally, if you are running OS X, pip is likely to fail building lxml (a dependency of Scrapy), since OS X does not ship libxml by default.

[1] Scrapy's stable release (1.0.5) does not support Python 3, so this program currently runs against only development versions after 1.1.0rc1. All code is tested to run against 1.1.0rc1 but is primarily run against 1.2.0dev2, the current development branch on github.

[2] We have created builds for all the necessary packages for Arch Linux.

This can often be solved by (re)installing the Xcode command line tools: `xcode-select install`. However, this is not always successful.

In either case, the best solution is probably virtualization. The following Dockerfile is minimally sufficient to a suitable environments in Ubuntu 16.04 Xerial Xerus (which must still rely on pip).[3]

```
FROM ubuntu:xenial
ENV LC_ALL C.UTF-8
RUN apt-get update &&                             \
        apt-get install -y                        \
                python3                           \
                python3-pip                       \
                python3-boto                      \
                python3-cookies                   \
                python3-cssselect                 \
                python3-bs4                       \
                python3-future                    \
                python3-fuzzywuzzy                \
                python3-levenshtein               \
                python3-lxml                      \
                python3-nltk                      \
                python3-responses                 \
                python3-requests-oauthlib         \
                python3-pydispatch                \
                python3-pymongo                   \
                python3-queuelib                  \
                python3-twisted                   \
                python3-w3lib
RUN pip3 install          \
```

[3] Virtualenv is another good solution to avoid altering your Python installation, but cannot be used to satisfy the dependence on libxml, since it manages only Python-internal dependencies.

```
        chatterbot       \
        scrapy==1.1.0rc1 \
        vaderSentiment
```

This Dockerfile installs the following dependencies, against which the code has been tested to run on Ubuntu 16.04:

```
attrs==15.2.0
beautifulsoup4==4.4.1
blinker==1.3
boto==2.38.0
chardet==2.3.0
ChatterBot==0.3.6
cookies==2.2.1
cryptography==1.2.3
cssselect==0.9.1
funcsigs==0.4
future==0.15.2
fuzzywuzzy==0.10.0
html5lib==0.999
idna==2.0
jsondatabase==0.1.1
lxml==3.5.0
mock==1.3.0
nltk==3.1
numpy==1.11.0
oauthlib==1.0.3
PAM==0.4.2
parsel==1.0.1
pbr==1.8.0
pyasn1==0.1.9
```

```
pyasn1-modules==0.0.7
PyDispatcher==2.0.5
PyJWT==1.3.0
pymongo==3.2.2
pyOpenSSL==0.15.1
pyserial==3.0.1
python-forecastio==1.3.4
python-Levenshtein==0.12.0
python-twitter==3.0rc1
queuelib==1.1.1
requests==2.9.1
requests-oauthlib==0.4.0
responses==0.3.0
Scrapy==1.1.0rc1
service-identity==16.0.0
six==1.10.0
textblob==0.11.1
Twisted==16.0.0
urllib3==1.13.1
vaderSentiment==0.5
w3lib==1.11.0
zope.interface==4.1.3
```

Building Corpora with Scrapy

Import stuff:

```python
from chatterbot import ChatBot
from datetime import datetime
from glob import glob
from json import loads
from nltk import ConditionalFreqDist
from nltk import FreqDist
from nltk.corpus import stopwords
from nltk.data import load
from nltk.tokenize import word_tokenize
from scrapy import Field
from scrapy import Item
from scrapy import Request
from scrapy import Spider
from scrapy import signals
from scrapy.crawler import Crawler
from scrapy.exporters import JsonLinesItemExporter
from scrapy.loader import ItemLoader
from scrapy.loader.processors import Join
from scrapy.loader.processors import MapCompose
from scrapy.loader.processors import TakeFirst
from scrapy.settings import Settings
from scrapy.utils import log
from twisted.internet import reactor
from w3lib.html import remove_tags
import json
import re

try:
    from string import strip
except:
    strip = str.strip
```

Scrapy provides an `Item` class which is used to collect scraped data. The following code defines an `StoryItem` class, as an expansion of `Item`, to hold information about stories. Each chapter of each story in the corpus will be stored as an object of class `StoryItem`, containing its full text (`body`), its `title`, `author`, a description or summary (`desc`), the `category` of the story, its `chapter` number within a larger work, the `site` it was scraped from, and the `url` from which it was scraped.

```
class StoryItem(Item):
    author   = Field()
    body     = Field()
    category = Field()
    chapter  = Field()
    desc     = Field()
    site     = Field()
    title    = Field()
    url      = Field()
```

`literotica.com` only provides the above information, so there is no need to extend the `StoryItem` class to capture literotica stories.

`fanfiction.net` provides slightly more, so we ought expand the class to contain more information:[4]

```
class FFItem(StoryItem):
    id        = Field()
    rating    = Field()
    language  = Field()
    words     = Field()
    chapters  = Field()
    complete  = Field()
    comments  = Field()
    likes     = Field()
    marks     = Field()
    published = Field()
    updated   = Field()
```

[4] This object currently specifies more data than we have been able to satisfactorily scrape from fanfiction.net. It was my intent to collect all of this information, but it would have taken too much additional time, due to the extraordinarily poor and inconsistent html of fanfiction.net.

Archive of Our Own provides all the same information as fanfiction.net, with some additions:

```
class AOItem(FFItem):
    fandom     = Field()
    characters = Field()
    ships      = Field()
    tags       = Field()
    warnings   = Field()
    hits       = Field()
```

Scrapy provides an API by which data can be loaded into an `Item` via an `ItemLoader`. The code below specifies a `StoryItemLoader` (which inherits the `ItemLoader` class defined by Scrapy) to load story data generally and two classes which inherit it, to load particular data from specific websites.

In cases where both the `StoryItemLoader` and one of the child classes specify a field (e.g. `body_out`), the child class supersedes the parent.

These loaders cannot be written before the `Spiders` defined below, as they operate on the output of those Spiders (and thus depend necessarily for their design on the design of those Spiders). They are defined here only because they must exist to be called by the Spiders.

```
class StoryItemLoader(ItemLoader):
    default_input_processor  = MapCompose(str.strip)
    default_output_processor = TakeFirst()

    body_in  = MapCompose(remove_tags)
    body_out = Join()

class FFItemLoader(StoryItemLoader):
    # desc_out = Join()
    desc_in  = MapCompose(remove_tags)
    desc_out = Join()
```

```
class AOItemLoader(StoryItemLoader):
    body_out       = Join()
    category_out   = Join(', ')
    warnings_out   = Join(', ')
    fandom_out     = Join(', ')
    characters_out = Join(', ')
    ships_out      = Join(', ')
    tags_out       = Join(', ')
```

Pipe the scraped text into Json items on the lines of a file (one file per spider).

```
class JsonLinesExportPipeline(object):
    def __init__(self):
        self.files = {}

    @classmethod
    def from_crawler(cls, crawler):
        pipeline = cls()
        crawler.signals.connect(
            pipeline.spider_opened,
            signals.spider_opened
        )
        crawler.signals.connect(
            pipeline.spider_closed,
            signals.spider_closed
        )
        return pipeline

    def spider_opened(self, spider):
        file = open('%s_stories.jl' % spider.name, 'w+b')
        self.files[spider] = file
        self.exporter = JsonLinesItemExporter(file)
        self.exporter.start_exporting()
    def spider_closed(self, spider):
        self.exporter.finish_exporting()
        file = self.files.pop(spider)
        file.close()

    def process_item(self, item, spider):
        self.exporter.export_item(item)
        return item
```

A spider to scrape `fanfiction.net`:

```
class FFSpider(Spider):
    name = "ff"
    allowed_domains = ["fanfiction.net"]
    start_urls = [
        "https://www.fanfiction.net/%s/" % c for c in
        (
            'anime',
            'book',
            'cartoon',
            'comic',
            'game',
            'misc',
            'movie',
            'play',
            'tv'
        )
    ]

    def parse(self, response):
        tags = response.xpath(
            '//td[@valign="TOP"]/div/a/@href'
        )
        for href in tags:
            url = response.urljoin(href.extract())
            yield Request(url, callback = self.parse_tag)

    def parse_tag(self, response):
        next = response.xpath(
            '//center[1]/a[contains(text(), "Next")]/@href'
        )
        for href in next:
            url = response.urljoin(href.extract())
            yield Request(url, callback = self.parse_tag)

        stories = response.xpath(
            '//div[contains(@class,"z-list")]/a[1]/@href'
        )
        for href in stories:
            long_url = response.urljoin(href.extract())
            url      = '/'.join(long_url.split('/')[0:-1])
            yield Request(url, callback = self.parse_story)

    def parse_story(self, response):
        header  = response.xpath(
            '//span[@class="xgray xcontrast_txt"]/text()'
        )
        head    = header.extract()
        chapter = int(response.url.split('/')[-1])
        more    = re.search('Chapters: [0-9]*', head[1])

        if more and chapter == 1:
            chapters = int(more.group(0).split()[1])
            base_url = '/'.join(response.url.split('/')[0:-1])
```

```python
    urls = [
        base_url + '/' +  str(x) for x in range(2, chapters+1)
    ]
    for url in urls:
        yield Request(url, callback = self.parse_story)

time_xpath  = response.xpath(
    '//span[@data-xutime]/@data-xutime'
)
times       = time_xpath.extract()
u_published = float(times[0])
d_published = datetime.fromtimestamp(u_published)
published   = d_published.strftime('%Y-%m-%d')
u_updated   = float(next(reversed(times)))
d_updated   = datetime.fromtimestamp(u_updated)
updated     = d_updated.strftime('%Y-%m-%d')

# info    = head[1].split(' - ')
# language = info[1]
# category = info[2]
# characters = info[3]
# words = ''.join([s for s in info[5] if s.isdigit()])

complete = str(bool('Complete' in ' '.join(head)))

loader = FFItemLoader(FFItem(), response=response)
loader.add_xpath(
    'title', '//*[@id="profile_top"]/b/text()'
)
loader.add_xpath(
    'author', '//*[@id="profile_top"]/a[1]/text()'
)
```

```
        loader.add_xpath(
            'desc', '//*[@id="profile_top"]/div'
        )
        loader.add_xpath(
            'body', '//*[@id="storytext"]'
        )
        loader.add_value(
            'url', response.url
        )
        loader.add_value(
            'site', 'fanfiction.net'
        )
        loader.add_value(
            'chapter', str(chapter)
        )
        loader.add_xpath(
            'rating', '//span[@class="xgray xcontrast_txt"]/a/text()'
        )
        loader.add_value(
            'published', published
        )
        loader.add_value(
            'updated', updated
        )
        yield loader.load_item()
```

Spider for Literotica:

```
class LESpider(Spider):
    name = "le"
    allowed_domains = ["literotica.com"]
    start_urls = [
        "https://www.literotica.com/c/%s/1-page" % c for c in
```

```
(
    'adult-how-to',
    'adult-humor',
    'adult-romance',
    'anal-sex-stories',
    'bdsm-stories',
    'bdsm-stories',
    'celebrity-stories',
    'chain-stories',
    'erotic-couplings',
    'erotic-horror',
    'erotic-letters',
    'erotic-novels',
    'erotic-poetry',
    'exhibitionist-voyeur',
    'fetish-stories',
    'first-time-sex-stories',
    'gay-sex-stories',
    'group-sex-stories',
    'illustrated-erotic-fiction',
    'interracial-erotic-fiction',
    'lesbian-sex-stories',
    'loving-wives',
    'masturbation-stories',
    'mature-sex',
    'mind-control',
    'non-consent-stories',
    'non-erotic-poetry',
    'non-erotic-stories',
    'non-human-stories',
    'reviews-and-essays',
    'science-fiction-fantasy',
            'taboo-sex-stories',
            'transsexuals-crossdressers'
        )
    ]

    def parse(self, response):
        next = response.xpath(
            '//*[@class="b-pager-next"]/@href'
        )
        for href in next:
            url = response.urljoin(href.extract())
            yield Request(url, callback = self.parse)

        stories = response.xpath(
            '//*[@id="content"]/div/div/h3/a/@href'
        )
        for href in stories:
            url = response.urljoin(href.extract())
            yield Request(url, callback = self.parse_story)

    def parse_story(self, response):
        next = response.xpath(
            '//*[@class="b-pager-next"]/@href'
        )
        for href in next:
            url = response.urljoin(href.extract())
            yield Request(url, callback = self.parse_story)

        loader = LEItemLoader(LEItem(), response = response)
        loader.add_xpath(
            'title', '//h1/text()'
        )
        loader.add_xpath(
            'author', '//*[@id="content"]/div[2]/span[1]/a/text()'
        )
        loader.add_value(
            'desc', ''
        )
        loader.add_xpath(
            'category', ('//*[@id="content"]/div[1]/a/text()')
        )
        loader.add_xpath(
            'body', '//*[@id="content"]/div[3]/div'
        )
        loader.add_value(
            'url', response.url
        )
        loader.add_value(
            'site', 'literotica.com'
        )
        loader.add_xpath(
            'page', '//*[@class="b-pager-active"]/text()'
        )
        yield loader.load_item()
```

Spider for AO3:

```
class AOSpider(Spider):
    name = "ao"
    allowed_domains = ["archiveofourown.org"]
    start_urls = [
        "https://archiveofourown.org/media"
    ]

    def parse(self, response):
        genres = response.xpath(
            '//h3/a/@href'
        )
        for href in genres:
            url = response.urljoin(href.extract())
            yield Request(url, callback = self.parse_genre)

    def parse_genre(self, response):
        tags = response.xpath(
            '//li/ul/li/a/@href'
        )
        for href in tags:
            url = response.urljoin(href.extract())
            yield Request(url, callback = self.parse_tag)



    def parse_tag(self, response):
        next = response.xpath(
            '(//ol[@role="navigation"])[1]/li[last()]/a/@href'
        )
        for href in next:
            url = response.urljoin(href.extract())
            yield Request(url, callback = self.parse_tag)

        stories = response.xpath('//h4/a[1]/@href')
        for href in stories:
            extension = '?view_adult=true&style=disable'
            url = response.urljoin(href.extract()) + extension
            yield Request(url, callback = self.parse_story)

    def parse_story(self, response):
        next = response.xpath(
            'a[contains(text(), "Next Chapter ")]/@href'
        )
        for href in next:
            extension = '?view_adult=true&style=disable'
            url = response.urljoin(href.extract()) + extension
            yield Request(url, callback = self.parse_story)

        chapter_path    =
response.xpath('//dd[@class="chapters"]/text()')
        chapters        = tuple(chapter_path.extract()[0].split('/'))
        current, total = chapters
        complete        = str(bool(current == total))

        loader = AOItemLoader(AOItem(), response = response)
        loader.add_xpath(
            'title', '//h2/text()'
        )
        loader.add_xpath(
            'author', '//a[@rel="author"]/text()'
        )
        loader.add_xpath(
            'desc', '(//*[@class="summary module"])[1]//p/text()'
        )
        loader.add_xpath(
            'body', '//*[@id="chapters"]//div/p/text()'
        )
        loader.add_value(
            'url', response.url
        )
        loader.add_value(
            'site', 'archiveofourown.org'
        )
```

```
loader.add_xpath(
    'category', '//dd[@class="category tags"]//a/text()'
)
loader.add_xpath(
    'language', '//dd[@class="language"]/text()'
)
loader.add_xpath(
    'rating', '//dd[@class="rating tags"]//a/text()'
)
loader.add_xpath(
    'warnings', '//dd[@class="warning tags"]//a/text()'
)
loader.add_xpath(
    'fandom', '//dd[@class="fandom tags"]//a/text()'
)
loader.add_xpath(
    'characters', '//dd[@class="character tags"]//a/text()'
)
loader.add_xpath(
    'ships', '//dd[@class="relationship tags"]//a/text()'
)
loader.add_xpath(
    'tags', '//dd[@class="freeform tags"]//a/text()'
)
loader.add_xpath(
    'hits', '//dd[@class="hits"]/text()'
)
loader.add_xpath(
    'published', '//dd[@class="published"]/text()'
)
loader.add_xpath(
    'updated', '//dd[@class="status"]/text()'
```

```
        )
        loader.add_xpath(
            'words', '//dd[@class="words"]/text()'
        )
        loader.add_xpath(
            'comments', '//dd[@class="comments"]/text()'
        )
        loader.add_xpath(
            'likes', '//dd[@class="kudos"]/text()'
        )
        loader.add_xpath(
            'marks', '//dd[@class="bookmarks"]/a/text()'
        )
        loader.add_xpath(
            'hits', '//dd[@class="hits"]/text()'
        )
        loader.add_value(
            'chapter', current
        )
        loader.add_value(
            'complete', complete
        )
        yield loader.load_item()
```

We haven't yet figured out what signal the spiders should send to avoid shutting down the reactor (which cannot be restarted) before all three are finished (if they are all run together).

```
# callback fired when the spider is closed
def callback(spider, reason):
    stats = spider.crawler.stats.get_stats()  # collect/log stats?

    # stop the reactor
    reactor.stop()
```

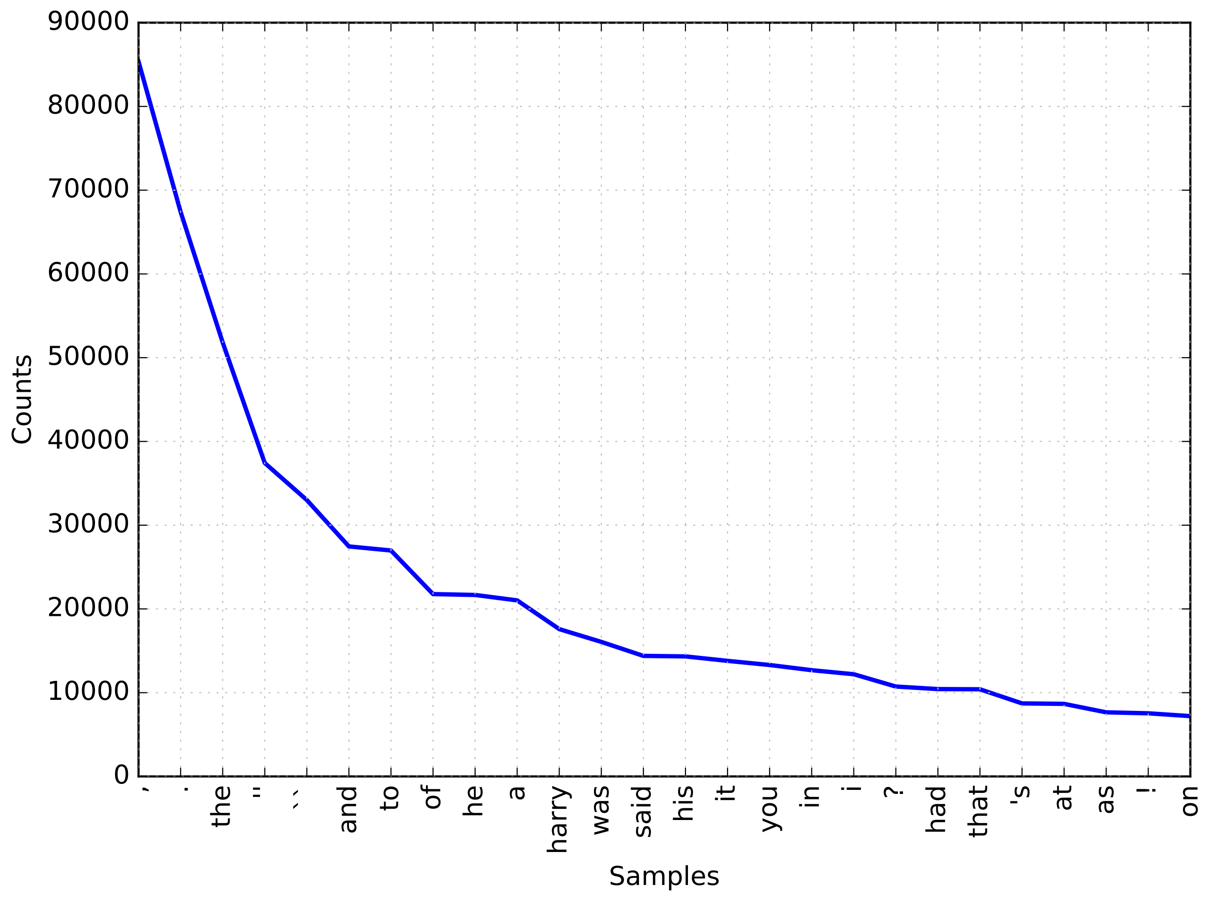
```
# instantiate settings and provide a custom configuration
settings = Settings()
settings.set(
    'ITEM_PIPELINES', {
        '__main__.JsonLinesExportPipeline': 100,
    }
)
settings.set(
    'USER_AGENT', 'Mozilla/5.0 (Windows NT 6.3; Win64; x64)'
)

for Spider in [ff_spider, ao_spider]:
    spider = Spider()
    crawler = Crawler(spider, settings)
    crawler.signals.connect(
        callback,
        signal = signals.spider_closed
    )
    crawler.crawl()
    log.configure_logging()
    reactor.run()
```

Analysis with NLTK

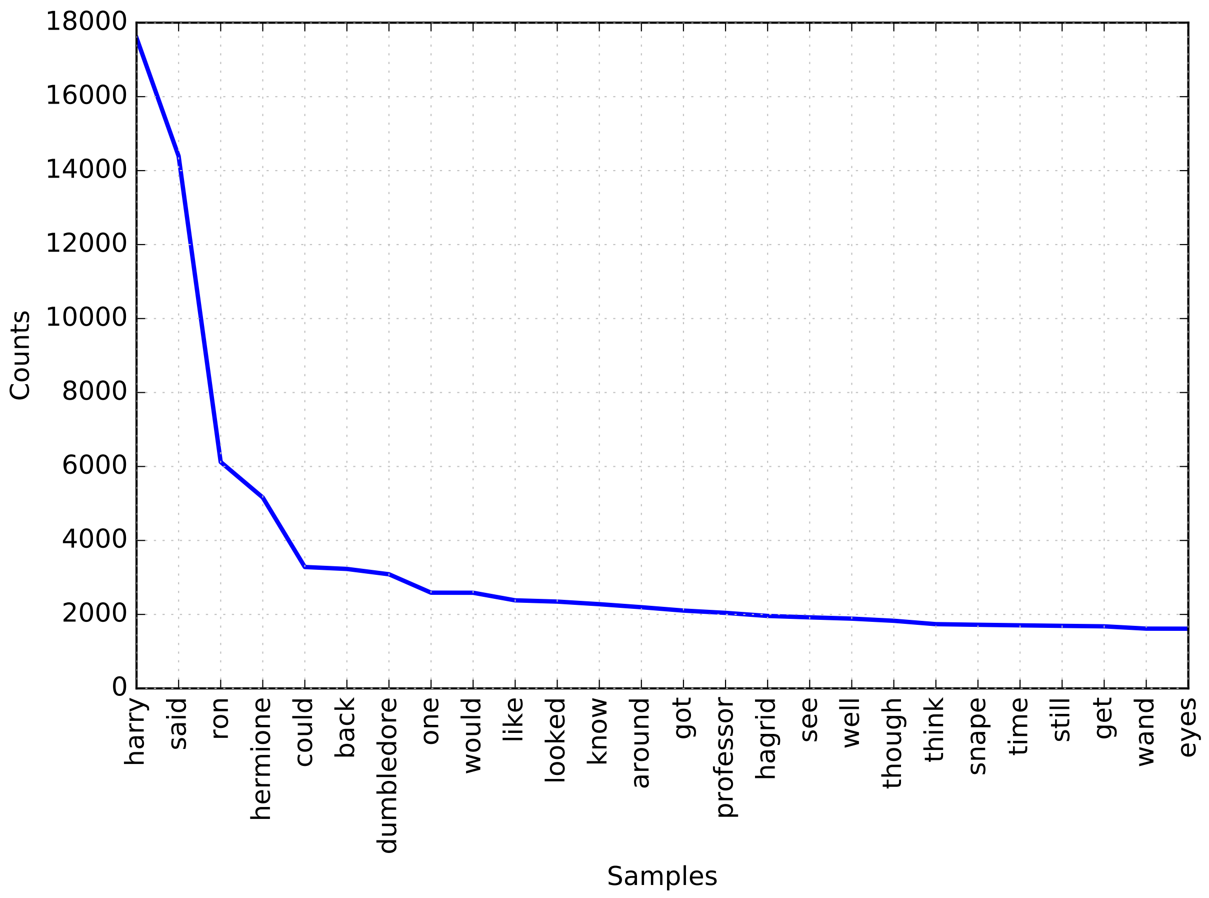
class StoryException(Exception): pass

```
stop = stopwords.words('english')
tokenizer = load('tokenizers/punkt/english.pickle')

# real

real_chapters = []
real_words    = []
dir = 'hp'
```

```
for path in glob("hp/*.txt"):
        file = open(path)
        chapter = file.read()
        chapter_tuple = (chapter, 'real')

        words = [ w.lower() for w in word_tokenize(chapter) ]

        real_chapters.append(chapter_tuple)
        real_words.extend(words)

word_total  = len(real_words)

fd = FreqDist(real_words)
fd.plot(26)
```



```
# filtered_real_words = [ w.lower() for w in real_words if w.isalpha()
]
filtered_real_words = [ w for w in real_words if w.isalpha() and w not
in stop ]

Rowling = filtered_real_words

fd  = FreqDist(filtered_real_words)
fd.plot(26)
```

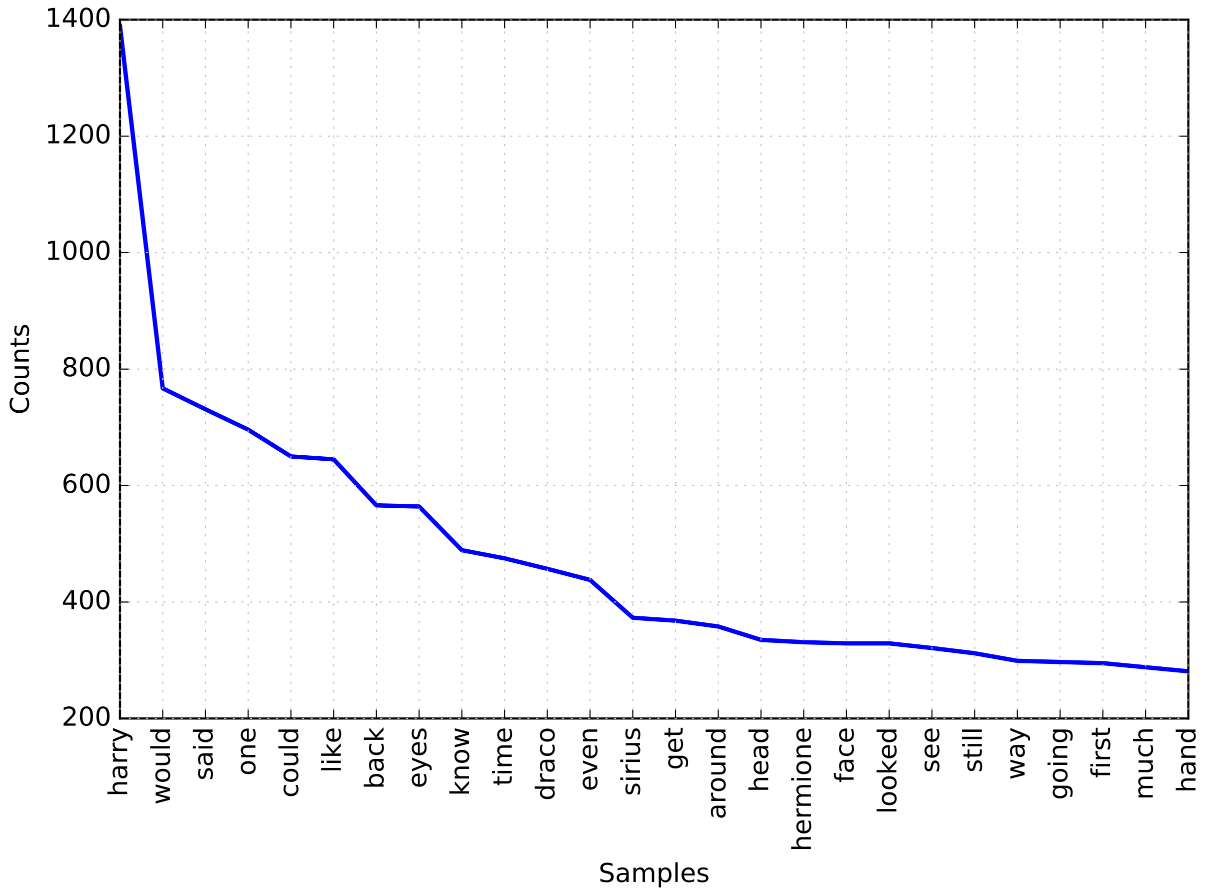


```
file = open('ao_hp_stories.jl')


ao_chapters = []
ao_words    = []
AO3         = []
AO3_normed  = []


for line in file.readlines():
    chapter_obj = loads(line)
    if chapter_obj['body'] and chapter_obj['language'] == 'English':
        chapter = chapter_obj['body']
        chapter_tuple = (chapter, 'fake')
        words = [ w.lower() for w in word_tokenize(chapter) ]
        ao_words.extend(words)
```

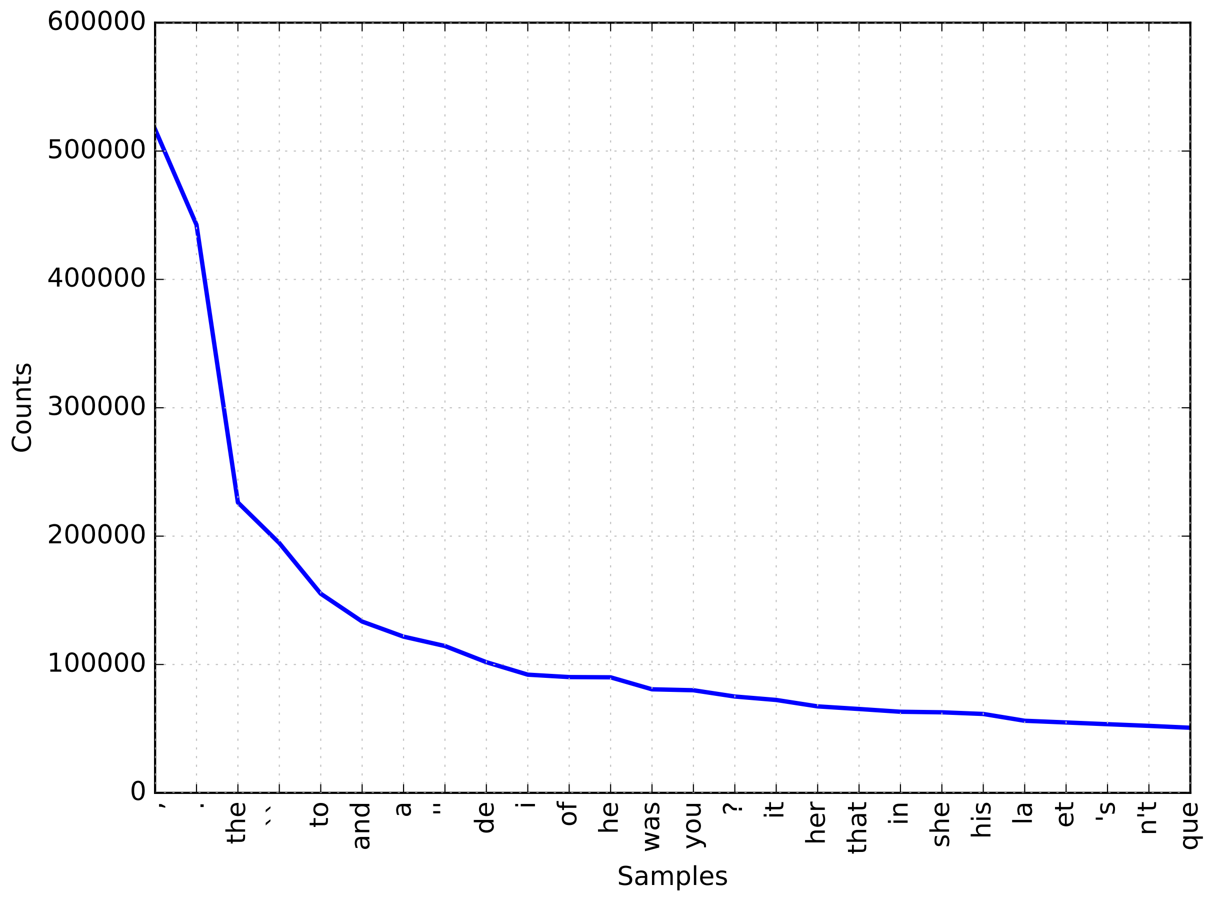
```
            ao_chapters.append(chapter_tuple)
            if len(AO3) < word_total:
                AO3.extend(words)

fd = FreqDist(ao_words)
fd.plot(26)
```

```
<class 'KeyError'>
'body'
```

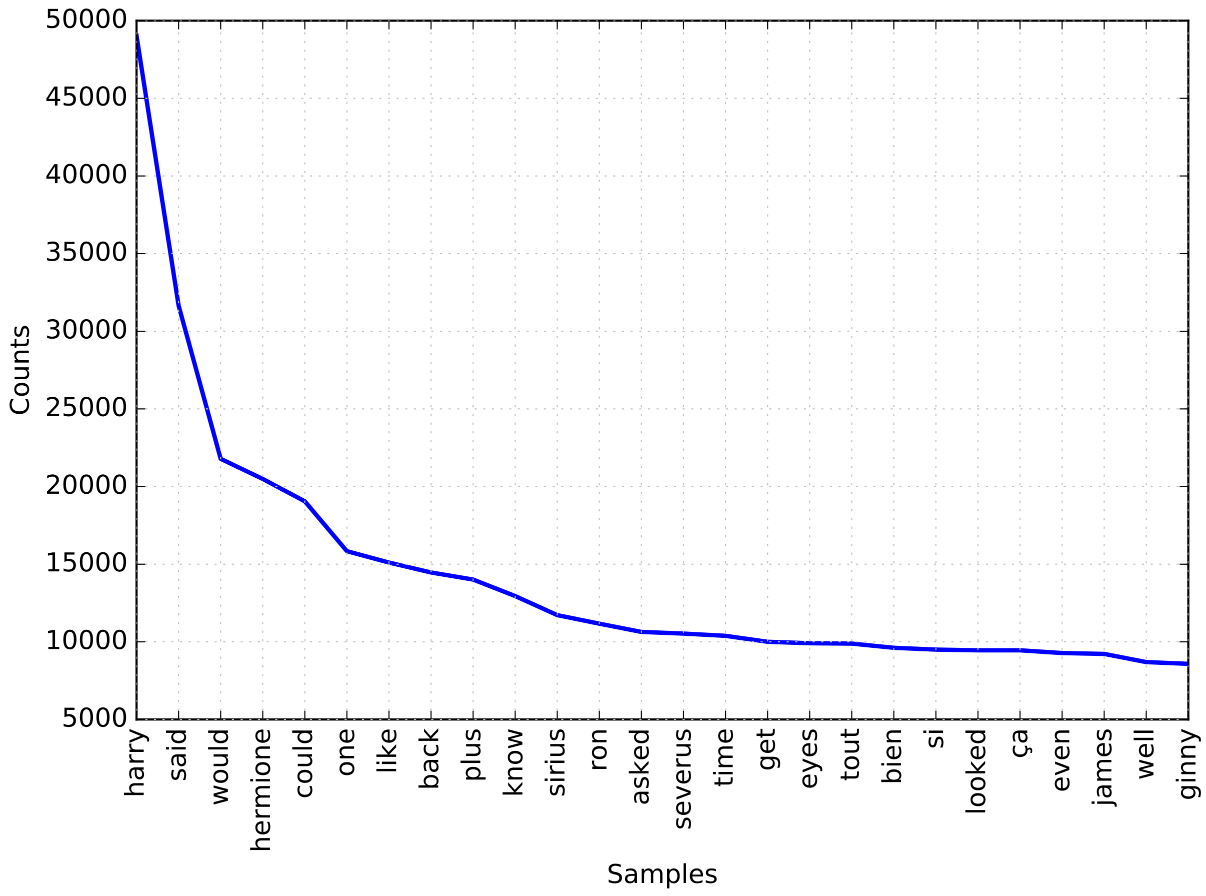
```
# filtered_ao_words = [ w.lower() for w in ao_words if w.isalpha() ]
filtered_ao_words = [ w for w in ao_words if w.isalpha() and w not in
stop ]

fd = FreqDist(filtered_ao_words)
fd.plot(26)
```



```
file = open('ff_hp_stories.jl')

ff_chapters        = []
ff_words           = []
fanfiction         = []
fanfiction_normed = []

for line in file.readlines():
    chapter_obj = loads(line)
    if chapter_obj['body']:
        chapter = chapter_obj['body']
        chapter_tuple = (chapter, 'fake')
        words = [ w.lower() for w in word_tokenize(chapter) ]
        ff_words.extend(words)
        ff_chapters.append(chapter_tuple)
        if len(fanfiction) < word_total:
            fanfiction.extend(words)

fd = FreqDist(ff_words)
fd.plot(26)
```

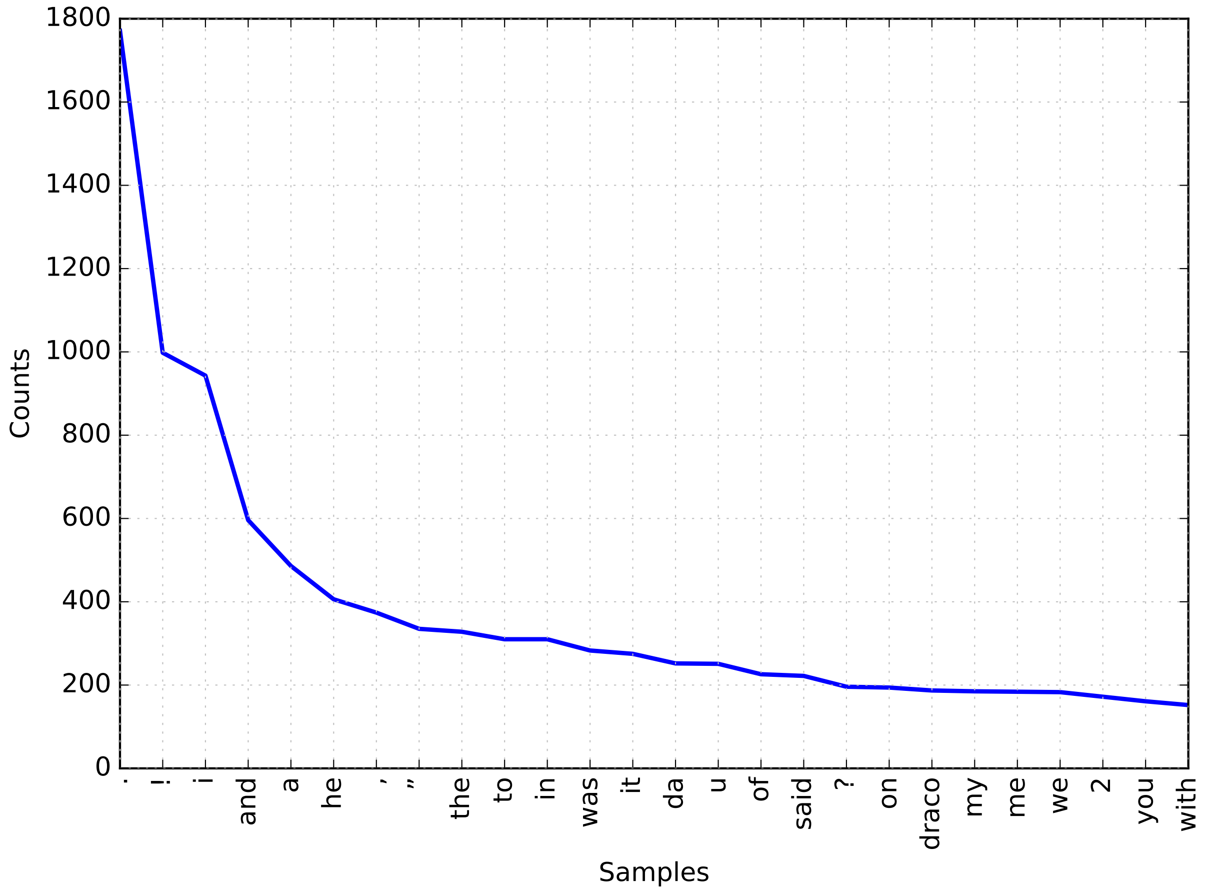


```
ff_stop = stop
for language in ('spanish', 'portuguese', 'french'):
    ff_stop.extend(
        stopwords.words(language)
    )

# filtered_ff_words = [ w.lower() for w in ff_words if w.isalpha() ]
filtered_ff_words = [ w for w in ff_words if w.isalpha() and w not in
ff_stop ]

fd = FreqDist(filtered_ff_words)
fd.plot(26)
```

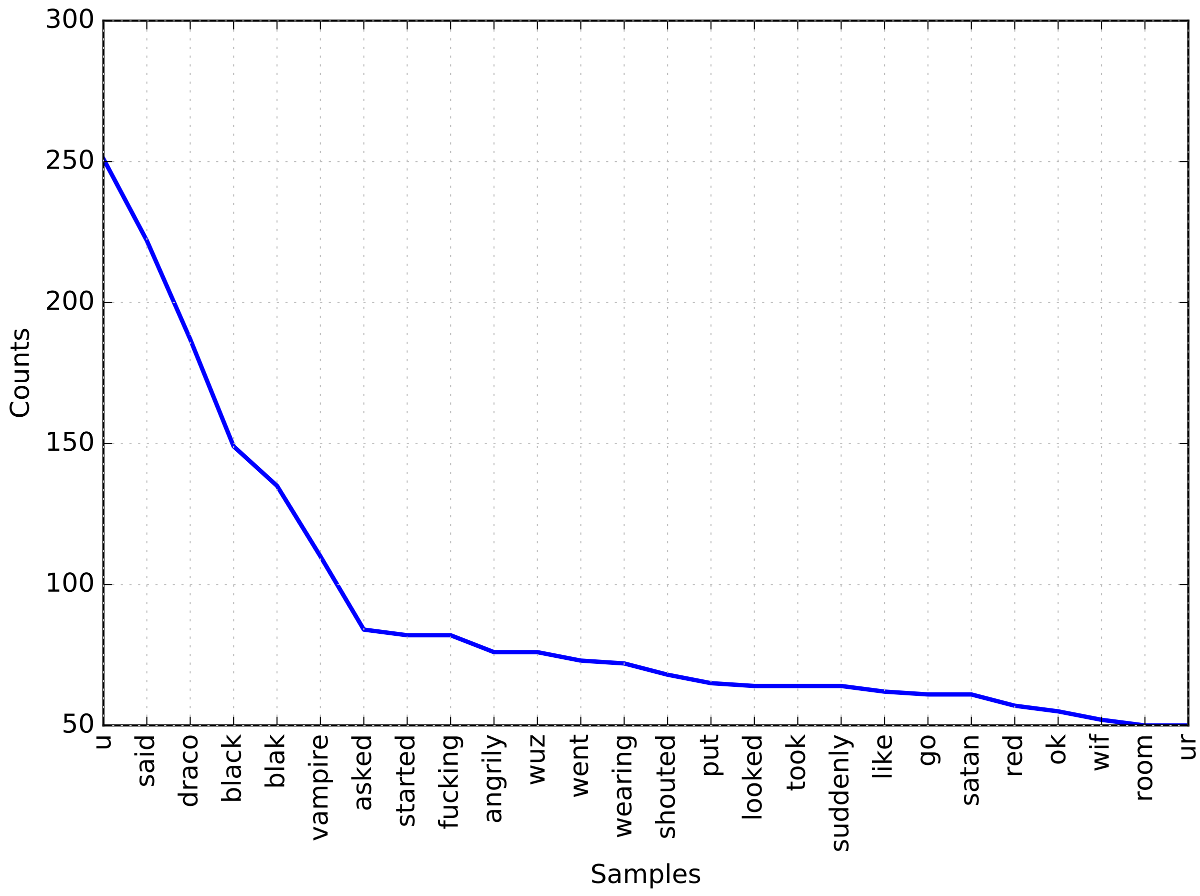


```
immortal_chapters = []
immortal_words    = []

for path in glob("immortal/*.txt"):
        file = open(path)
        chapter = file.read()
        chapter_tuple = (chapter, 'real')

        words = [ w.lower() for w in word_tokenize(chapter) ]

        immortal_chapters.append(chapter_tuple)
        immortal_words.extend(words)

fd = FreqDist(immortal_words)
```

```
fd.plot(26)
```

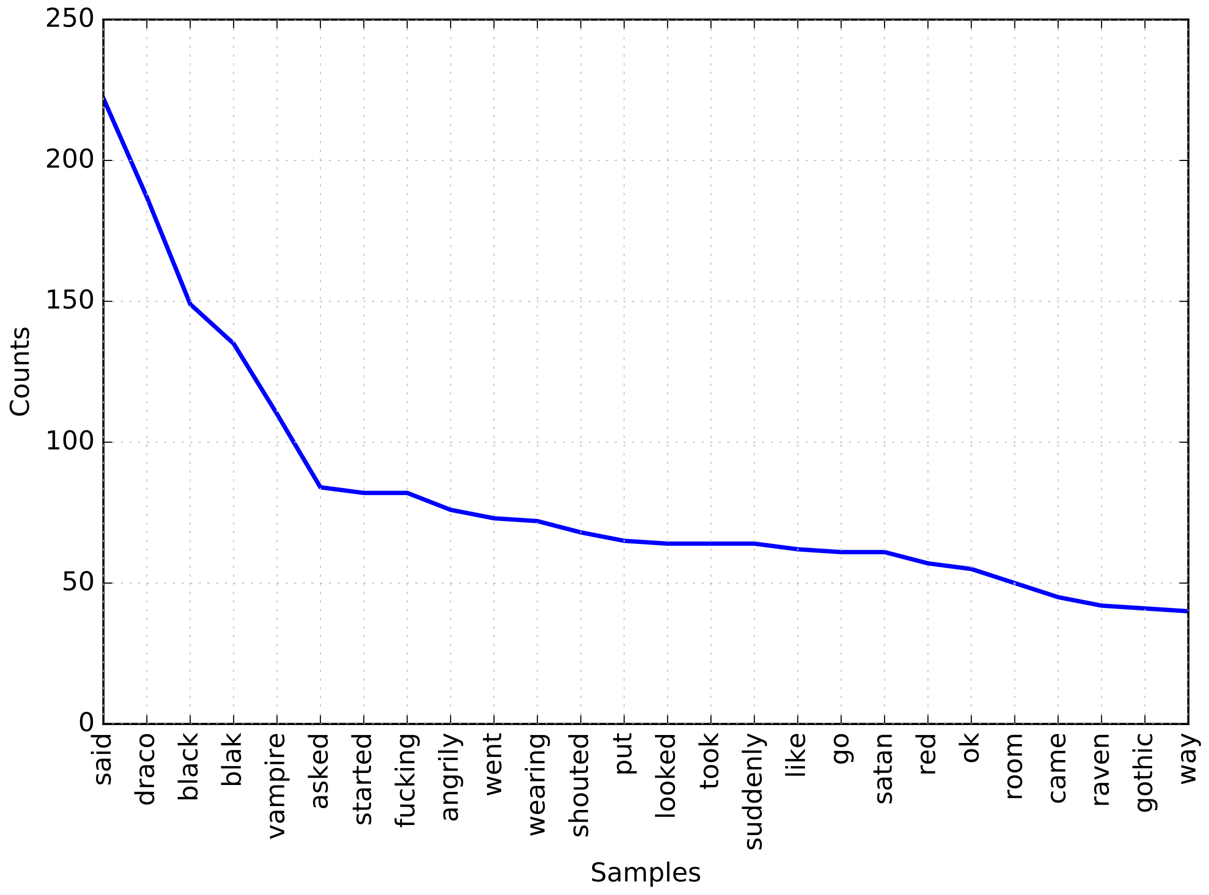


```
# filtered_immortal_words = [ w.lower() for w in immortal_words if
w.isalpha() ]
filtered_immortal_words = [ w for w in immortal_words if w.isalpha()
and w not in stop ]


fd  = FreqDist(filtered_immortal_words)
fd.plot(26)
```

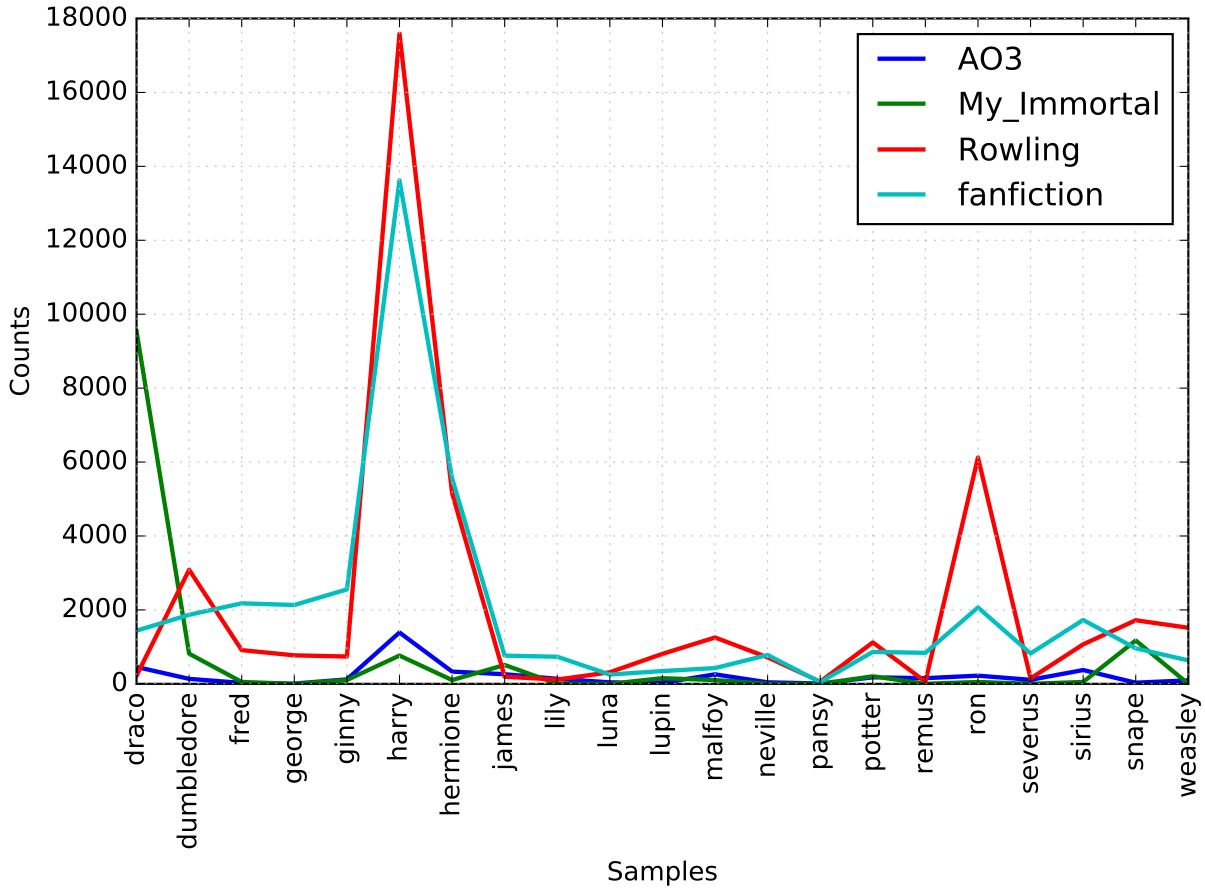


```
i_stop = [
    'da',
    'dat',
    'u',
    'ur',
    'wif',
    'wuz',
    'chapter'
]

# filtered_immortal_words = [ w.lower() for w in immortal_words if
w.isalpha() ]
filtered_immortal_words = [ w for w in immortal_words if w.isalpha()
and w not in stop and w not in i_stop ]
```

```
immortal_scale = int(len(real_words) / len(immortal_words))


My_Immortal = immortal_words * immortal_scale


fd  = FreqDist(filtered_immortal_words)
fd.plot(26)
```



```
names = (
    'neville',
    'draco',
    'dumbledore',
    'fred',
    'george',
    'ginny',
    'harry',
    'hermione',
    'james',
    'lily',
    'luna',
    'lupin',
    'malfoy',
    'pansy',
    'potter',
    'remus',
    'ron',
    'severus',
    'sirius',
    'snape',
    'weasley'
)

AO3 = [ w.lower() for w in AO3 if w.isalpha() ]
AO3 = [ w for w in AO3 if w not in stop ]

fanfiction = [ w.lower() for w in fanfiction if w.isalpha() ]
fanfiction = [ w for w in fanfiction if w not in stop ]

My_Immortal = [ w.lower() for w in My_Immortal if w.isalpha() ]
My_Immortal = [ w for w in My_Immortal if w not in stop and w not in
i_stop]

unified_words = (
    (source, word)
    for source in (
        'Rowling',
        'AO3',
        'fanfiction',
        'My_Immortal'
    )
    for word in eval(source)
    if word in names
)

cfd = ConditionalFreqDist(unified_words)

cfd.plot()
```



Appendices

Training a ChatterBot

For fun, we tried to implement a chatbot that drew its responses from reported speech in the stories from the corpus.

Here we create an untrained chatbot and trian it on the (tiny) corpus that ships with the chatterbot package.

```
chatbot = ChatBot(
    "Mary Sue",
    io_adapter = "chatterbot.adapters.io.NoOutputAdapter"
)
chatbot.train("chatterbot.corpus.english")
```

We then split stories at double quotes and then train the bot on all odd-numbered indices (which should all between quotes). This will take quite a long time, and create an absolutely huge database.

```
for path in glob("*_stories.txt"):
    file = open(path)

    for line in file.readlines():
        conversation = []
        story = json.loads(line)
        body = story['body']
        graphs = body.split('\n')
        for graph in graphs:
            speech = graph.split('"')[1::2]
            utterance = ' '.join(speech)
            if utterance:
                conversation.append(utterance)
        if conversation:
            chatbot.train(conversation)
```

```
print(
     chatbot.get_response("Hello")
)
```

Unfortunately, once the chatbot has been trained on a decent amount of data, it becomes unusably slow (even with better Levenshtein distance algorithms). This is almost certainly a result of its storage method (a flat, plain-text database stored to disk) and the nature of its implementation (in Python, not compiled).

If we were to seek to implement this in a useful way, we would have to implement a much cleverer logic, inferring categories, hypernyms, etc.

References

Black, R. W. (2008). *Adolescents and online fan fiction* (1st ed.) (No. 23). Peter Lang Publishing.

Chander, A., & Sunder, M. (2007). Everyone's a superhero: A cultural theory of "Mary Sue" fan fiction as fair use. *California Law Review*, *95*(2). Retrieved from http://www.jstor.org/stable/20439103

de Certeau, M. (1984). *L'invention du quotidion* [The practice of everyday life] (Vol. 1; S. Rendall, Trans.). Berkeley, CA: University of Californial Press. (Original work published 1980)

Gilespie, T. (2007). *My immortal* (Raven, Ed.).

destinationtoast. (2014a). *Everybody loves Hogwarts.* Retrieved from http://destinationtoast.tumblr.com/post/77453118197/

destinationtoast. (2014b). *Relationships: FFNet vs AO3.* Retrieved from http://destinationtoast.tumblr.com/post/98680961704/

Jenkins, H. (1992). *Textual poachers: Television fans and participatory culture*. New York, NY: Routledge.

Jung, S. (2004). Queering popular culture: Female spectators and the appeal of writing slash fan fiction. *Gender Forum: An Internet Journal for Gender Studies*, *8*. Retrieved from http://www.genderforum.org/issues/gender-queeries/queering-popular-culture-female-spectators-and-the-appeal-of-writing-slash-fan-fiction/

Katyal, S. (2006). Performance, property, and the slashing of gender in fan fiction. *Journal of Gender, Social Policy, and the Law*, *14*(103). Retrieved from http://ssrn.com/abstract=869742

MacDonald, M. (2006). Harry Potter and the fan fiction phenom. *The Gay & Lesbian Review Worldwide*, *XIII*(1).

Rowling, J. K. (1997). *Harry Potter and the Sorcerer's Stone*. Scholastic.

Rowling, J. K. (1998). *Harry Potter and the Chamber of Secrets*. Scholastic.

Rowling, J. K. (1999). *Harry Potter and the prisoner of Azkaban*. Scholastic.

Rowling, J. K. (2000). *Harry Potter and the Goblet of Fire*. Scholastic.

Rowling, J. K. (2003). *Harry Potter and the Order of the Phoenix*. Scholastic.

Rowling, J. K. (2005). *Harry Potter and the Half-Blood Prince*. Scholastic.

Rowling, J. K. (2007). *Harry Potter and the Deathly Hallows*. Scholastic.

Schwabach, A. (2009). The Harry Potter lexicon and the world of fandom: Fan fiction, outsider works, and copyright. *University of Pittsburgh Law Review*, *70*. Retrieved from http://ssrn.com/abstract=1274293

Steenhuyse, V. V. (2011). The writing and reading of fan fiction and transformation theory. *CLCWeb: Comparative Literature and Culture*, *13*(4). doi: 10.7771/1481-437.1691

Thomas, B. (2011). What is fanfiction and why are people saying such nice things about it? *StoryWorlds: A Journal of Narrative Studies*, *3*(1). Retrieved from https://muse.jhu.edu/article/432689 doi: 10.1353/stw.2011.0001

Tosenberger, C. (2008). Homosexuality at the online Hogwarts: Harry Potter slash fanfiction. *Children's Literature*, *36*, 185-207. doi: 10.1353/chl.0.0017