Python for Linguists

February 20, 2018

Contents

1 Python and NLP for Linguists

I'm going to try something novel: giving this talk from a Jupyter notebook so I can run code on the fly.

While I was writing up this notebook, this talk turned into a broad introduction to NLP as well as Python. Oh well.

1.1 Who is this guy?

- Henry Anderson (henry.anderson@uta.edu)
- Data Scientist in the University Analytics department
- Specialist in unstructured data (i.e., text), machine learning, and Natural Language Processing
- First year Linguistics masters student, with interests in computational social science, digital language use, and the language of online communities and networks.

1.2 Contents

- Who stands to gain the most
- Why consider *programming*, generally?
- Why consider *Python*, specifically?
- Some demos:
  - Custom concordance code, with massive flexibility
  - Quick n-gram analysis
  - Automatic dependency parsing, POS-tagging, lemmatization, tokenization, etc. (i.e., text preprocessing)
  - Topic models
  - Word vectors
  - Classification and regression tasks with text

1.3 Goals of this talk

- Put some basic computational/programming tools on your radar.
- Give a sense of what *type* of work can be done with these tools.
- Make you generally aware of the scope and nature of computational tools.
- Give some *very* basic exposure to Python.
  - We'll walk through some very basic code examples, but we'll skim over most of them.

1.4 This talk is *not*...

- A tutorial on Python, the dataset, or the libraries.
  - That can come later, if people are interested.
- A tutorial in natural language processing, text processing, or big data.
- Really a tutorial in anything.

1.5 Who stands to gain

- Anyone who deals with *data:* people interested in corpus work, sociolinguistics, natural language processing, digital/online language, etc.
- Anyone interested in *computational social science* (CSS): i.e. general social science approaches leveraging large datasets and computational horsepower.
  - CSS is currently exploding, and is a hugely important avenue for applied social science research.
  - CSS is also massively interdisciplinary: programming, statistics, machine learning, AI, network analysis, linguistics, sociology, psychology, etc all combine to make CSS happen.
- If you deal mostly with theory, or are primarily an experimentalist, you may stand to gain less from this talk. (But you're still welcome!)

1.6 What does programming offer?

- (Quite literally) infinite control over your data processing: you're not limited by the features someone else decided to code into their program–you can change your code up to do anything you want.
- Scalability and automation of your data work
  - Work with literally millions of documents and billions of words with relative ease.
  - Automate steps from data collection through final analysis.
- Some types of analysis simply are infeasible to do by hand–network analysis, topic modeling, word embeddings, etc–and *have* to be done computationally.
- Marketable skills: even a little bit of Python, Java, or any other language can open doors in the job market.
- You'll feel like a badass.

1.7 What does *Python* offer?

- Free (as in speech, not beer. But also as in beer), open-source, royalty-free. No licenses to sign, no royalties to pay, and *essentially no restrictions* on what you can and can't do with it. (the Python Software Foundation license is an extremely permissive BSD-type license)
- Easy-to-learn language.
  - Very user-friendly language with very friendly users.
  - Great documentation and stupid amounts of free, high-quality learning resources.
  - Among its core ideas:
    - Code is read far more than it is written, so the language should be *human-readable.*
    - "There should be one, and preferably only one, obvious way to do it." I.e., the most straightforward approach is *usually* the best. (This results in a lot of people writing straightforward, fairly easy-to-follow code).

  - Commonly taught as a first programming language, so there are LOTS of materials for eveyone from beginning programmers to seasoned professionals; the Python community is also very welcoming of newcomers.

- General purpose language: can do (almost) everything you want to make it to.
  - Compare to R, which is great for statistics, and a pain for a lot of other stuff.
  - Or Matlab, which is great for being a broken, slow, difficult software environment, and isn't so good at being, well, good.
    * (this has been your mandatory "Matlab is bad" comment)

- Rapidly becoming *the* language for data science, displacing even R in most applications. (R is still dominant for raw statistics, though Python has plenty of packages that implement common statistical tests).
  - Though, keep an eye on a different language–Julia–over the next few years. It is truly a worthy contender, but has yet to hit version 1.0 as of this talk.

- **For linguists**: a *huge* array of language processing functionality and libraries.
  - spaCy, basically a Python version of Stanford's CoreNLP toolkit (lemmatization, tokenization, dependency parsing, POS tagging, and more).
  - Gensim, full of topic models and pretty bleeding-edge NLP tools.
  - Natural Language Toolkit (NLTK), a *massive* library that's designed to teach a lot of NLP concepts (but can be used for some serious production work too).
  - Pandas for R-like dataframes, statistics, and general tabular data management.
  - Matplotlib (and others like Seaborn, PyGal, Bokeh, ...) for high-quality, powerful data visualization.
  - scikit-learn for non-neural machine learning (support vector machines, random forests, and a few text features like basic preprocessing)
    * Side note, the scikit-learn User Guides are an *excellent* technical crash course in machine learning, even if you're not too interested in Python.
  - Networkx for performing network analysis.
  - Tensorflow+Keras, for quickly and easily building neural networks.
  - PyTorch, an up-and-coming (but extremely exciting) neural network library.
  - And dozens more.

2 Some links

- Python.org, the official Python website.
  - Download Python 3.6.4, the latest releast as of writing this.
  - *Or,* download the Anaconda distribution, which comes pre-packaged with common data science libraries and can make library management easier (at the expense of being a larger installation).
  - Python Standard Library reference
  - Python's official, basic tutorial. It's not the best out there, but it's pretty good, and pretty quick. Highly recommended if you've already got a bit of programming experience.

–

- PyCharm, an excellent Python editor (download the community edition, not the professional edition–it's free and still has more features than most people need)
- Automate the Boring Stuff with Python, a free (Creative Commons-licensed) eBook that introduces Python via concrete, hands-on examples and projects.
- r/learnpython, a subreddit dedicated to people learning Python and asking questions about the language.
- Project Euler, a collection of math problems designed to be used as programming practice (in any language).
- Rosalind, another collection of practice problems, but geared at genetics and bioinformatics.
- PyCon's YouTube channels (2017, 2016, etc) have a lot of good videos, including some long-form tutorials and workshops.
- PyData, a data science themed Python conference/convention, posts almost all of their talks to YouTube.

3 Running this notebook

To run this notebook, you will need to install the following Python libraries: * Jupyter (to run/view the notebook itself) * Gensim * spaCy * You'll need to download two of spaCy's language models: `en_core_web_sm` and `en_core_web_lg`. Installation instructions are here. * Numpy * Scikit-learn (goes by sklearn when installing) * Matplotlib * Natural Language Toolkit (NLTK) * tqdm * Pandas

Open Jupyter and navigate to this .ipynb file, then open it. Every major section is designed to be able to run independently, minus the "Setup Code" section, which should always be run first.

For the first part of the talk, you'll also need to download `glen carrig.txt` from the Github repository and have it in the same folder as this notebook.

For the second part of the talk, you'll need to download the Blog Authorship Corpus and unzip its files into a folder named "blogs" (case-sensitive) in the same folder as this notebook.

4 Setup Code

First, some necessary setup–we just need to have this program create some folders to save stuff into.

```
In [1]: import os

        if not os.path.isdir("corpus data files"):
            os.mkdir("corpus data files")
        if not os.path.isdir("model files"):
            os.mkdir("model files")
```

5 Some basic demos

5.1 Concordances

Concordances can be done with regular expressions and a teeny tiny bit of legwork. (By the way: if you're working with text, you have no excuse to not learn regular expressions. But that would

be another talk all unto itself). We'll work with the text of William Hope Hodgeson's novel *The Boats of the "Glen Carrig"*, a 1907 horror novel. The text was taken from Project Gutenberg, with the site's boilerplate text removed from the front and back.

```
In [2]: %%time

        import re

        def concordance(text, token, window=50):
            pattern = re.compile(r"\b{}\b".format(token.strip()), re.IGNORECASE)
            # convert all whitespaces to single space characters
            text = re.sub(r"\s+", " ", text)
            for i in pattern.finditer(text):
                print(
                    "...",
                    text[i.start() - window:i.start()].rjust(window, " "),
                    text[i.start():i.start() + window].ljust(window, " "),
                    "...",
                    sep=""
                )

        glen_carrig = open("glen carrig.txt", "r", encoding="utf8").read()
        concordance(glen_carrig, "think", window=30)
```

```
...that land. For, indeed, now I think of it, I can remember th...
..., indeed, we had ever need to think more of such. And then, ...
...ces on my throat he seemed to think but little, suggesting t...
...e; so that I knew not what to think, being near to doubting ...
...rawling in the valley. Yet, I think the silences tried us th...
...her, and further than this, I think with truth I may say, we...
...fter so long upon the sand. I think, even thus early, I had ...
...me thus armed, they seemed to think that I intended a jest, ...
...ch that it seemed hopeless to think of success; but, for all...
...red in me; nor, could I, do I think I would; for were I succ...
...d so big and unwieldy. Now, I think that Jessop gathered som...
...were both of us young, and, I think, even thus early we attr...
...at no decent-minded man could think the worse of her; but th...
...ars, and this little thing, I think, brought back more clear...
...thing; for I had not dared to think upon that which already ...
...our love one for the other, I think yet, and ponder how that...
Wall time: 52 ms
```

And if we want to get *really* clever, we can have our concordance function search by stemmed forms. We'll revisit stemming in a bit more detail shortly; for now, just know that stemming is the process of determining an uninflected form of words, but it's based purely on character patterns–so each word is treated completely in isolation, with no information about parts of speech.

We need to stem the original text, then search for concordances of any tokens that get stemmed to the same value as our input. Then we run the previous concordance function on those tokens:

In [3]: %%time

```
import re
from gensim.parsing.preprocessing import stem_text

def stem_concordance(text, token, window=50):
    text = re.sub(r"\s+", " ", text)
    # get a unique list of all word-like tokens using a basic regex
    word_finder = re.compile(r"[A-z0-9]+", re.MULTILINE)
    vocab_to_stem = {
        i.lower():stem_text(i)
        for i in set(word_finder.findall(text))
    }
    if token.lower().strip() not in vocab_to_stem:
        print("Token is not in the vocabulary.  Please try again.")
        return
    # now flip the dict to {stemmed_form:{set of unstemmed form}}
    stem_to_vocab = {i:set() for i in vocab_to_stem.values()}
    for i in vocab_to_stem:
        stem_to_vocab[vocab_to_stem[i]].add(i.lower())
    # look up other tokens that have same stem as input token
    stemmed_token = vocab_to_stem[token]
    possible_forms = stem_to_vocab[stemmed_token]

    # and now run the previous concordance function.
    for i in possible_forms:
        concordance(text, i, window=window)

glen_carrig = open("glen carrig.txt", "r", encoding="utf8").read()
stem_concordance(glen_carrig, "think", window=30)
```

```
c:\users\andersonh\appdata\local\programs\python\python36\lib\site-packages\gensim\utils.py:1167
  warnings.warn("detected Windows; aliasing chunkize to chunkize_serial")


...bo'sun bade him keep silence, thinking it was but a piece of...
...ithering sounds. And at that, thinking a host of evil things...
...nation, we turned to descend, thinking that this would be th...
...h I spent much of my watch in thinking over the details of m...
... men had caught their fish--, thinking that, if Tompkins had...
...he proceeded to explain that, thinking they were cut off fro...
...that land. For, indeed, now I think of it, I can remember th...
..., indeed, we had ever need to think more of such. And then, ...
...ces on my throat he seemed to think but little, suggesting t...
...e; so that I knew not what to think, being near to doubting ...
...rawling in the valley. Yet, I think the silences tried us th...
...her, and further than this, I think with truth I may say, we...
...fter so long upon the sand. I think, even thus early, I had ...
```

```
...me thus armed, they seemed to think that I intended a jest, ...
...ch that it seemed hopeless to think of success; but, for all...
...red in me; nor, could I, do I think I would; for were I succ...
...d so big and unwieldy. Now, I think that Jessop gathered som...
...were both of us young, and, I think, even thus early we attr...
...at no decent-minded man could think the worse of her; but th...
...ars, and this little thing, I think, brought back more clear...
...thing; for I had not dared to think upon that which already ...
...our love one for the other, I think yet, and ponder how that...
Wall time: 1.93 s
```

That added less than a second to the total runtime. Nice.

Now let's do it again, but with spaCy instead of Gensim. spaCy has built-in tokenization, lemmatization, and more that are all based on large, pre-trained machine learning models. This will give us much better accuracy–both with tokenizing and lemmatizing–but at the cost of higher runtime. spaCy also has multiple models to choose from–for English, there's small, medium, and large. The bigger the model, the better its accuracy, but also the slower it runs. But since the interface is exactly the same, we'll use the small model for speed. (the small model is pretty darn good, anyways)

We'll also revisit lemmatization in a bit more detail shortly. The short version: it's like stemming, but it returns a valid, real word corresponding to the uninflected form of a token. (unlike stemming, which might map "today" to the root form "todai"–lemmatization would correctly map this to "today").

In [4]:
```
%%time

import re
import spacy

def lemma_concordance(text, token, window=50):
    nlp = spacy.load("en_core_web_sm", disable=["tagger", "parser", "ner"])

    # directly get the mapping of raw form to lemma from spaCy's
    # tokenization/stemming
    word_finder = re.compile(r"[A-z0-9]+", re.MULTILINE)
    vocab_to_stem = {
        i.lower_:i.lemma_
        for i in nlp(text)
    }
    if token.lower().strip() not in vocab_to_stem:
        print("Token is not in the vocabulary.  Please try again.")
        return
    # now flip the dict to {stemmed_form:{set of unstemmed form}}
    stem_to_vocab = {i:set() for i in vocab_to_stem.values()}
    for i in vocab_to_stem:
        stem_to_vocab[vocab_to_stem[i]].add(i.lower())
```

```
            # Now get the stemmed form of the input token and look up
            # the list of possible unstemmed forms--this approximates
            # finding other inflected forms of the same word.
            stemmed_token = vocab_to_stem[token]
            possible_forms = stem_to_vocab[stemmed_token]

            # and now run the previous concordance function.
            for i in possible_forms:
                concordance(text, i, window=window)

        glen_carrig = open("glen carrig.txt", "r", encoding="utf8").read()
        lemma_concordance(glen_carrig, "think", window=30)
```

```
... went hither and thither, the thought that IT--for that is h...
...ry of the spring, it might be thought that we should set up ...
...upon the trunk. With a sudden thought that it would make me ...
... our leaving, he had given no thought to take them with him;...
...n here, how that I had little thought all this while for the...
...aving no tides; but I had not thought to come upon such an o...
...estion, I grew full of solemn thought; for it seemed to me t...
...e like for the moment to have thought he had seen a second d...
...ght by the weed; and then the thought came to me of the end ...
...quito will make; but I had no thought to blame any mosquito...
...rooted them. At least, so the thought came to me. And so we ...
...er while than hitherto he had thought needful. Having conclu...
...s further ahead: but I had no thought for these when I perce...
...de. Then, having by this time thought a little upon the matt...
...is liking. But it must not be thought that he did naught but...
...y an enormous crab. Now I had thought the crab we had tried ...
...ll breakers--which we had not thought needful to carry to th...
...zon, and there would come the thought to me of the terror of...
...hen, even as I fell upon this thought, the bo'sun clapped me...
...n in cooler minds, we had not thought strange, seeing that s...
...ed too sincerely to have much thought to watch the hulk, whi...
...were become so excited at the thought of fellow creatures al...
...he rope to us, and at this we thought more upon his saying; ...
... happy to tell that we had no thought at this juncture but f...
...to a method of rescue. Then a thought came to me (waked perc...
... me a short spell of disquiet thought. It was in this wise:-...
...n such a sight, and indeed, I thought nothing more of it tha...
...ls, and further, I could have thought I perceived a flicker ...
...ed it at all such parts as he thought in any way doubtful, a...
...tion, and then, suddenly, the thought came to me that the sc...
...w I was made to understand my thought of the previous night,...
...the first, and then, a sudden thought coming to me, I thrust...
...t remained hid. Then a sudden thought came into my brain, an...
...y, as a result of some little thought, he brought out from t...
...fore the evening. And, at the thought of this, we experience...
```

```
... the lesser rope; for that he thought they in the ship were ...
...t break it, and then a second thought that something might b...
...ted within myself at this new thought, as, indeed, was the b...
...ly officer remaining to them, thought there might be good ch...
...ened the rope so much as they thought proper, they left it t...
...rface, and, at that, a sudden thought came to me which sent ...
...her; but that I, for my part, thought rather the better, see...
...d which had beset them at the thought that they should all o...
...such interest since, that the thought of food had escaped me...
...r to have stayed, the which I thought, for a moment, had not...
...as very terrible, this sudden thought of failure (though it ...
...ing gear about it, so that he thought it would be so safe as...
...of us in the ship that he had thought to go at that moment; ...
...bo'sun bade him keep silence, thinking it was but a piece of...
...ithering sounds. And at that, thinking a host of evil things...
...nation, we turned to descend, thinking that this would be th...
...h I spent much of my watch in thinking over the details of m...
... men had caught their fish--, thinking that, if Tompkins had...
...he proceeded to explain that, thinking they were cut off fro...
...that land. For, indeed, now I think of it, I can remember th...
..., indeed, we had ever need to think more of such. And then, ...
...ces on my throat he seemed to think but little, suggesting t...
...e; so that I knew not what to think, being near to doubting ...
...rawling in the valley. Yet, I think the silences tried us th...
...her, and further than this, I think with truth I may say, we...
...fter so long upon the sand. I think, even thus early, I had ...
...me thus armed, they seemed to think that I intended a jest, ...
...ch that it seemed hopeless to think of success; but, for all...
...red in me; nor, could I, do I think I would; for were I succ...
...d so big and unwieldy. Now, I think that Jessop gathered som...
...were both of us young, and, I think, even thus early we attr...
...at no decent-minded man could think the worse of her; but th...
...ars, and this little thing, I think, brought back more clear...
...thing; for I had not dared to think upon that which already ...
...our love one for the other, I think yet, and ponder how that...
Wall time: 3.76 s
```

5.2 N-gram frequencies

Python has a number of ways we could find n-grams. The first is using a pre-built tool, like NLTK's ngrams() function, or a Phrases()/Phraser() combination from Gensim, which are actually used to find multi-word phrases. Or, we could hack it together ourselves with a few lines of code.

First, let's hack it together ourselves. Then we'll print out the top most common N-grams and plot the frequencies by rank (on a logarithmic scale, naturally. We're not monsters, after all). We'll use spaCy's tokenization for maximum accuracy.

First, we'll tokenize our document and convert everything to lowercase using spaCy. We'll also remove punctuation and stopwords while we're at it.

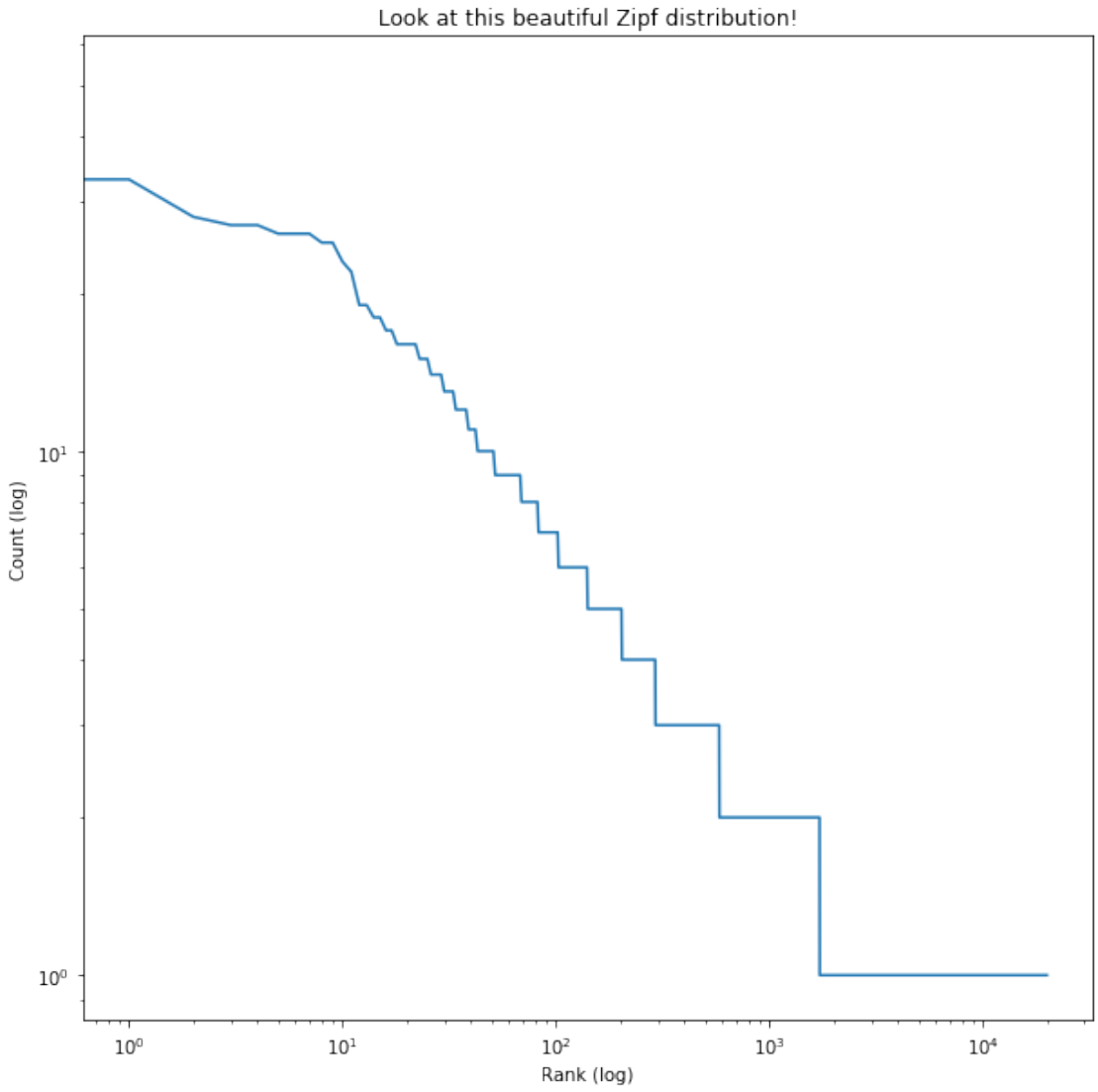
```
In [5]: %%time
        %matplotlib inline

        from collections import Counter
        from pprint import pprint
        import re

        import matplotlib as mpl
        import matplotlib.pyplot as plt
        import spacy

        # Change this number manually to change the N in the ngrams
        NGRAM_N = 2

        # Preprocess the document
        nlp = spacy.load("en_core_web_sm", disable=["tagger", "parser", "ner"])
        doc = open("glen carrig.txt", "r", encoding="utf8").read()
        doc = re.sub(r"\s+", " ", doc)
        doc = [
            i.lower_
            for i in nlp(doc)
            if not i.is_punct
            and not i.is_stop
        ]

Wall time: 3.15 s
```

Now, we'll find n-grams by explicitly coding an n-gram finding bit of Python.

```
In [6]: %%time
        %matplotlib inline

        # change the default matplotlib figure size for Jupyter's sake
        mpl.rcParams["figure.figsize"] = (10,10)

        # Find the n-grams with some manual code
        n_grams = [
            "_".join(doc[i:i+NGRAM_N])
            for i in range(0, len(doc) - NGRAM_N)
        ]
        n_grams = Counter(n_grams)

        # do some prettier formatting than the default printing
        print(f"{'NGRAM':<30s}\tCOUNT")
        for i in n_grams.most_common(20):
            print(f"{i[0]:<30s}\t{i[1]}")
```

```
        # Now, let's just plot the counts by rank.
        counts = sorted(n_grams.values(), reverse=True)
        plt.plot(counts)
        plt.yscale("log")
        plt.xscale("log")
        plt.title("Look at this beautiful Zipf distribution!")
        plt.xlabel("Rank (log)")
        plt.ylabel("Count (log)")
        plt.show()
```

NGRAM	COUNT
i_saw	51
and_i	33
i_discovered	28
i_perceived	27
weed_continent	27
now_i	26
yet_i	26
mistress_madison	26
bo'sun_bade	25
then_i	25
at_i	23
i_found	22
i_knew	19
second_mate	19
bo'sun_'s	18
i_heard	18
i_went	17
and_presently	17
i_come	16
captain_'s	16



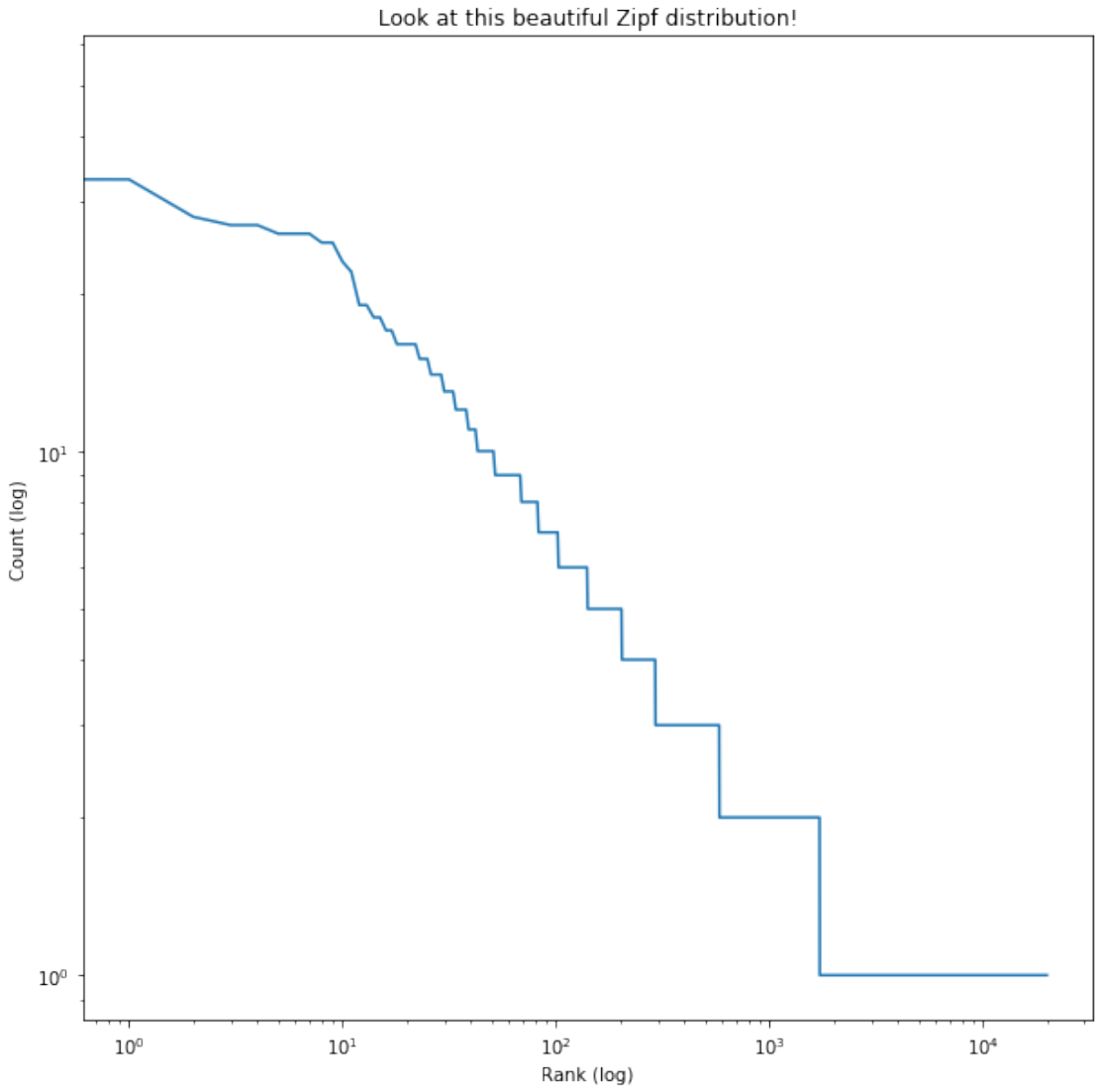

```
Wall time: 729 ms
```

NLTK's ngrams() function will find ngrams for us, like we just did by hand. It'll be more readable, but function exactly the same, and run almost exactly as far–so in general, this method might be preferable for most people.

```
In [7]: %%time
        %matplotlib inline

        from nltk import ngrams

        # change the default matplotlib figure size for Jupyter's sake
        mpl.rcParams["figure.figsize"] = (10,10)
```

```
        # Find n-grams
        n_grams = ngrams(doc, NGRAM_N)
        n_grams = Counter(n_grams)

        # do some prettier formatting than the default printing
        print(f"{'NGRAM':<30s}\tCOUNT")
        for i in n_grams.most_common(20):
            # okay, so the format string changes a little too
            print(f"{'_'.join(i[0]):<30s}\t{i[1]}")

        # Now, let's just plot the counts by rank.
        counts = sorted(n_grams.values(), reverse=True)
        plt.plot(counts)
        plt.yscale("log")
        plt.xscale("log")
        plt.title("Look at this beautiful Zipf distribution!")
        plt.xlabel("Rank (log)")
        plt.ylabel("Count (log)")
        plt.show()
```

```
NGRAM                           COUNT
i_saw                           51
and_i                           33
i_discovered                    28
i_perceived                     27
weed_continent                  27
now_i                           26
yet_i                           26
mistress_madison                26
bo'sun_bade                     25
then_i                          25
at_i                            23
i_found                         22
i_knew                          19
second_mate                     19
bo'sun_'s                       18
i_heard                         18
i_went                          17
and_presently                   17
i_come                          16
captain_'s                      16
```




```
Wall time: 1.98 s
```

5.3 Phrase-finding/collocation analysis

Ngrams are fun and all, but what if you want to find multi-word phrases that appear more often than they should by random chance alone (i.e., collocates)? Well, as before, we could hack a bit of code together, or we could use a pre-built tool from the amazing Gensin library: the Phrasing tools! These tools let you scan a(n already processed) corpus of texts, and finds bigrams that are collocated more than the raw prior distributions would indicate. Then, these tools let you transform your original corpus, replacing these bigrams with a single token. You can repeat this process all you want to find arbitrarily long phrases.

Before doing this, we should run our text through a basic preprocessing pipeline in Gensim. We'll revisit this in a bit more detail later to talk about what it does; for now, just know that it

automates a lot of the basic preprocessing steps for us, like lowercasing, removing stopwords, stemming, etc.

We'll use all the default values for our phrasing models except for the threshold (to guarnatee we find at least *some* phrases for this demo), but they'll be provided explicitly to show how much customization there is. Note that the phraser expectes a *list of sentences*, i.e. a *list of lists of words.* We don't strictly need to make them the actual sentences; the only reason Gensim says to use sentences is to avoid collocation across sentence boundaries.

```
In [8]: import re

        from gensim.models.phrases import Phrases
        from gensim.parsing.preprocessing import preprocess_string

        doc = open("glen carrig.txt", "r", encoding="utf8").read()
        # clean up line breaks and other whitespace issues
        doc = re.sub(r"\s+", " ", doc)
        doc = preprocess_string(doc)
        phrasing = Phrases(
            [doc], # phrases() expects a list of tokenized sentences/documents
            min_count=5,
            threshold=10,
            max_vocab_size=40000000,
            delimiter=b"_", # this has to be a byte string--just a quirk of this model
            progress_per=10000,
            scoring="default",
        )
```

Now, we can look at some of the phrases that our phrase model discovered. This will print out the phrases in the order they're found in the text, so we might see duplicates.

```
In [9]: from pprint import pprint

        found_phrases = list(phrasing.export_phrases([doc]))
        pprint(found_phrases[:15])
```

```
[(b'shook head', 51.86585365853659),
 (b'shook head', 51.86585365853659),
 (b'sun bade', 21.133969389783342),
 (b'sun bade', 21.133969389783342),
 (b'main cabin', 139.67159277504103),
 (b'main cabin', 139.67159277504103),
 (b'captain cabin', 77.1869328493648),
 (b'big cabin', 33.3307210031348),
 (b'aboard hulk', 36.35042735042735),
 (b'main cabin', 139.67159277504103),
 (b'main cabin', 139.67159277504103),
 (b'tot rum', 253.1547619047619),
 (b'captain cabin', 77.1869328493648),
 (b'captain cabin', 77.1869328493648),
```

```
 (b'captain cabin', 77.1869328493648)]
```

And, we can transform our original document(s), replacing all of the discovered bigrams with a single token (e.g., `["the", boat"] -> "the_boat"`). Gensim likes to use the indexing syntax to do transformations–it's a bit weird but you get used to it.

Note that we'll get a warning from Gensim (warnings are not errors–they're more of a "heads up, something looks weird here" sort of notice). Gensim also has Phraser() objects, which are initialized from a Phrases() object, and are much faster at transforming a corpus. This only really matters when you're dealing with *massive* corpora and datasets; for our single book, we don't really need to bother, but I'll show how it would be done anyways.

```
In [10]: from gensim.models.phrases import Phraser

         # transform the text with the original phrases object...
         phrased_doc = phrasing[doc]
         print(phrased_doc[500:550])

         # ...or by creating a new Phraser() from it first.
         phraser = Phraser(phrasing)
         phrased_doc = phraser[doc]
         print('\n\n', phrased_doc[500:550])
```

```
c:\users\andersonh\appdata\local\programs\python\python36\lib\site-packages\gensim\models\phrase
  warnings.warn("For a faster implementation, use the gensim.models.phrases.Phraser class")
```

```
['befal', 'georg', 'youngest', 'prentic', 'boi', 'seat', 'pluck', 'sleev', 'inquir', 'troubl', '
```

```
 ['befal', 'georg', 'youngest', 'prentic', 'boi', 'seat', 'pluck', 'sleev', 'inquir', 'troubl',
```

(as you can see, the results of Phrases() and Phraser() are the same–Phraser() will just be *much* faster, and use much less memory, for very large phrasing passes).

6 Some more fun demos: Natural Language Processing 101

Now, let's do some more interesting demos. These will focus on more sophisticated (but still "entry-level") NLP techniques and tools.

We will: * Go from raw data to cleaned, workable text. * Look at some of spaCy and Gensim's text (pre)processing tools * Do some basic topic modeling with Latent Dirichlet Allocation * Do some basic classification and regression tasks with some text.

6.1 Getting our data

The previous demos were very simple (even simplistic) and don't really leverage all the cool stuff Python–or programming in general–can do for you. Let's work with a non-trivial dataset now and

do some more NLP-like work. We'll use the Blog Authorship Corpus. Download it to the same folder as this notebook, then unzip it into a folder names "blogs" (case-sensitive!).

Each author's blog posts are stored as a .xml file, named in the following format:
`[ID number].[gender].[age].[industry of employment].[astrological sign].xml`
E.g., `11253.male.26.Technology.Aquarius.xml`
And they contain blog data that looks like this:
"' 20,July,2004

```
About to go t bed late (again) got sucked into (another) late night film. Tonight was  urlLink
```

"'

First thing's first: we need to deal with the XML formatting. Fortunately, Python has some excellent tools for that, e.g. `xml.etree.ElementTree`. We'll also want to use a better data structure to represent this text. There's an absolutely indispensible library called Pandas, which gives you R-style dataframes to work with (and Pandas is probably the single biggest reason that Python has taken over the data science world, demoting R to second-place).

```
In [11]: import os

         files = [i.path for i in os.scandir("blogs")]
         print(files[:5])
```

```
['blogs\\1000331.female.37.indUnk.Leo.xml', 'blogs\\1000866.female.17.Student.Libra.xml', 'blogs
```

Let's do a first pass where we just try to parse each file, to make sure there are no problems. (Data validations is a step you *absolutely do not skip* if you're doing any real data work, after all!)

```
In [12]: from xml.etree import ElementTree

         # sorted() just makes sure the files are always in the same order
         # for the demos
         for i in sorted(files):
             try:
                 ElementTree.parse(i)
             except Exception as e:
                 print("\nEXCEPTION ON FILE:", i)
                 print("EXCEPTION:", e)
                 break
```

```
EXCEPTION ON FILE: blogs\1000866.female.17.Student.Libra.xml
EXCEPTION: not well-formed (invalid token): line 103, column 225
```

Uh-oh. Let's take a look at the file that broke and go see where the problem is. Fortunately, the error above gave us a line *and* a column number, so we can go to the exact location where an issue was encountered.

```
<Blog>
[...]

<date>14,January,2003</date>
<post>


      Hehe, just finished dinner! Yum! I'm so happy right now. I don't even know why, I just...a

</post>
```

Column numbers aren't displayed in this notebook, but the error is at the ampersand in `Law & Order`. As it turns out, ampersands are special characters in XML code, and need to be escaped specially (in this case, as `&`). We could do some manual replacement of characters with the appropriate XML escapes, but that sounds like a lot of work and a lot of room for error.

Fortunately, our document structure is so simple that we can just hack this together with regular expressions. This will be fast, but **is not** how we should in general deal with problematic XML or other markup files–we'd want to use some of Python's more rudimentary tools, like the base HTML or XML parsers, and overwrite their functionality (e.g. by subclassing).

We'll create a list of dictionaries (think JSONs), which we can easily pass into Pandas to make a big, beautiful, glorious Dataframe object (which we'll save to a .csv so we can re-open it directly later). We'll work with this Dataframe for most the rest of this demo.

We'll also create a second dataframe, where we only have one entry per author, and concatenate all of their posts together. We'll use this at the end of the talk for some regression and classification tasks where we're predicting author-level variables, rather than working at the document level.

In [13]:
```
%%time

import os
import re

from tqdm import tqdm_notebook as tqdm
import pandas as pd

# pre-compile patterns we'll use a lot--for speed
date_finder = re.compile(r"^<date>(.*?)</date>$", re.MULTILINE)
post_finder = re.compile(r"^<post>$(.*?)^</post>$", re.MULTILINE|re.DOTALL)
whitespace_cleaner = re.compile(r"\s+", re.MULTILINE)

def process_file(infile, by_author=False):
    # get the user metadata from the filename
    metadata = os.path.split(infile)[-1].split(".")
    text = open(infile, "r", encoding="ISO-8859-1").read()

    # Extract date and post content from the file
    dates = date_finder.findall(text)
    posts = post_finder.findall(text)
```

```
            # assert check will crash our program if it fails--
            # this make sure our approach works.
            assert len(dates) == len(posts)
            if by_author == False:
                blog_data = [
                    {
                         "Author ID" : metadata[0],
                         "Gender"    : metadata[1],
                         "Age"       : int(metadata[2]),
                         "Industry"  : metadata[3],
                         "Sign"      : metadata[4],
                         "Date"      : dates[i],
                         "Post"      : whitespace_cleaner.sub(" ", posts[i])
                    }
                    for i in range(len(dates))
                ]
            elif by_author == True:
                posts = " ".join(
                    whitespace_cleaner.sub(" ", i)
                    for i in posts
                )
                blog_data = {
                     "Author ID" : metadata[0],
                     "Gender"    : metadata[1],
                     "Age"       : int(metadata[2]),
                     "Industry"  : metadata[3],
                     "Sign"      : metadata[4],
                     "Posts"     : posts
                }

            return blog_data

        blog_dataframe = pd.concat(
            pd.DataFrame(process_file(i, by_author=False))
            for i in tqdm(files, desc="Generating Dataframe")
        )
        print("Casting Date column to datetime format.")
        print("Saving blog dataframe to Blog Data.csv.")
        blog_dataframe.to_csv("corpus data files/Blog Data.csv", index=False)
        print(blog_dataframe)
```

```
HBox(children=(IntProgress(value=0, description='Generating Dataframe', max=19320), HTML(value='
```

```
Casting Date column to datetime format.
Saving blog dataframe to Blog Data.csv.
```

	Age	Author ID	Date	Gender	Industry	\
0	37	1000331	31,May,2004	female	indUnk	
1	37	1000331	29,May,2004	female	indUnk	
2	37	1000331	28,May,2004	female	indUnk	
3	37	1000331	28,May,2004	female	indUnk	
4	37	1000331	28,May,2004	female	indUnk	
5	37	1000331	28,May,2004	female	indUnk	
6	37	1000331	21,June,2004	female	indUnk	
7	37	1000331	18,June,2004	female	indUnk	
8	37	1000331	15,June,2004	female	indUnk	
9	37	1000331	08,June,2004	female	indUnk	
10	37	1000331	03,June,2004	female	indUnk	
11	37	1000331	02,June,2004	female	indUnk	
12	37	1000331	03,July,2004	female	indUnk	
0	17	1000866	23,November,2002	female	Student	
1	17	1000866	20,November,2002	female	Student	
2	17	1000866	19,November,2002	female	Student	
3	17	1000866	18,November,2002	female	Student	
4	17	1000866	18,November,2002	female	Student	
5	17	1000866	18,November,2002	female	Student	
6	17	1000866	18,November,2002	female	Student	
7	17	1000866	20,December,2002	female	Student	
8	17	1000866	18,December,2002	female	Student	
9	17	1000866	18,December,2002	female	Student	
10	17	1000866	15,December,2002	female	Student	
11	17	1000866	15,December,2002	female	Student	
12	17	1000866	14,January,2003	female	Student	
13	17	1000866	13,January,2003	female	Student	
14	17	1000866	13,January,2003	female	Student	
15	17	1000866	12,January,2003	female	Student	
16	17	1000866	12,January,2003	female	Student	
..	...	...	...	...	...	
28	27	998966	11,July,2004	male	indUnk	
0	25	999503	30,June,2004	male	Internet	
1	25	999503	28,June,2004	male	Internet	
2	25	999503	28,June,2004	male	Internet	
3	25	999503	27,June,2004	male	Internet	
4	25	999503	21,June,2004	male	Internet	
5	25	999503	19,June,2004	male	Internet	
6	25	999503	19,June,2004	male	Internet	
7	25	999503	15,June,2004	male	Internet	
8	25	999503	15,June,2004	male	Internet	
9	25	999503	14,June,2004	male	Internet	
10	25	999503	10,June,2004	male	Internet	
11	25	999503	09,June,2004	male	Internet	
12	25	999503	08,June,2004	male	Internet	
13	25	999503	08,June,2004	male	Internet	
14	25	999503	07,June,2004	male	Internet	

15	25	999503	06,June,2004	male	Internet
16	25	999503	06,June,2004	male	Internet
17	25	999503	06,June,2004	male	Internet
18	25	999503	17,July,2004	male	Internet
19	25	999503	15,July,2004	male	Internet
20	25	999503	14,July,2004	male	Internet
21	25	999503	11,July,2004	male	Internet
22	25	999503	06,July,2004	male	Internet
23	25	999503	05,July,2004	male	Internet
24	25	999503	04,July,2004	male	Internet
25	25	999503	03,July,2004	male	Internet
26	25	999503	02,July,2004	male	Internet
27	25	999503	01,July,2004	male	Internet
28	25	999503	01,July,2004	male	Internet

	Post	Sign
0	Well, everyone got up and going this morning...	Leo
1	My four-year old never stops talking. She'll ...	Leo
2	Actually it's not raining yet, but I bought 1...	Leo
3	Ha! Just set up my RSS feed - that is so easy...	Leo
4	Oh, which just reminded me, we were talking a...	Leo
5	I've tried starting blog after blog and it ju...	Leo
6	My 20th high school urlLink reunion is this w...	Leo
7	We always have pizza on Friday nights. It tak...	Leo
8	Okay, I saw it this past weekend. Not as good...	Leo
9	I've been cataloguing film scripts at work. W...	Leo
10	Paul Martin promised today that if he is elec...	Leo
11	No, I still haven't seen it, but apparently i...	Leo
12	Well, it's over! It was good to see so many p...	Leo
0	Yeah, sorry for not writing for a whole there...	Libra
1	Yeah, so today was ok, late arrival. I'm not ...	Libra
2	Yay, Tuesday...no longer Monday! Whoopie! Plu...	Libra
3	RAR!	Libra
4	Thought- OK...so, I'm all for midgets and wha...	Libra
5	Yeah, so it's later. My parents found somethi...	Libra
6	Yeah, so this is just a place for me to vent ...	Libra
7	Eventful day...well, sort of. Eventful to me ...	Libra
8	Eventful day...well, sort of. Eventful to me ...	Libra
9	:o) Hey...it looks like Alex. You know, the n...	Libra
10	I'm baaack, well, sort of. I have a lesson so...	Libra
11	Wow, haven't written in a long time. Been pre...	Libra
12	Hehe, just finished dinner! Yum! I'm so happy...	Libra
13	I got so many compliments on my hair! It was ...	Libra
14	Avoiding homework...yet again, shock shock. p...	Libra
15	Welp...haha, what a funny word, perhaps becau...	Libra
16	Yeah, so avoiding homework right now and talk...	Libra
..	...	...
28	urlLink Awwww... isnt Amy a cutie? Posted by ...	Taurus

```
0    I have done such a good job for so long pusin...  Cancer
1    I never said this before, but my Dad is in Ja...  Cancer
2    Happy Birthday to me! I will be using urlLink...  Cancer
3    Tomorrow I turn 25. I have made some great pr...  Cancer
4    A bit of lyrical magic for all of you out in ...  Cancer
5    urlLink Are all voluntary acts selfish? Whene...  Cancer
6    "How do you know the chosen ones? No greater ...  Cancer
7                    All my friends are make believe.  Cancer
8    Last couple of days have been hell at work, l...  Cancer
9    Wow, what a weekend. Aside from one very down...  Cancer
10   The realization Today on my way to lunch I fo...  Cancer
11   Bad workout today, I dunno I guess my mind ju...  Cancer
12   My new obsession. I realized that I have been...  Cancer
13   Nice busy day at work, lots of long boring me...  Cancer
14   If at first you dont succeed... a) Destroy a...  Cancer
15   Have you ever stopped to think that maybe you...  Cancer
16   Feeling a bit lonely tonight, I will try and ...  Cancer
17   This is the start of my blog. Its a place for...  Cancer
18   You shouldnt expect to learn about someones l...  Cancer
19   When I was a kid I used to pray every night f...  Cancer
20   Another day older, hair thinning, aches and b...  Cancer
21   I don't want to start any blasphemous rumours...  Cancer
22   urlLink something positive - the comic equive...  Cancer
23   Chillin to some groove salad, studying BGP co...  Cancer
24   Today we celebrate our independence day. In h...  Cancer
25   Ugh, I think I have allergies... My nose has ...  Cancer
26   "Science is like sex; occasionally something ...  Cancer
27   urlLink Dog toy or marital aid I managed 10/1...  Cancer
28   I had a dream last night about a fight when I...  Cancer

[681288 rows x 7 columns]
Wall time: 3min 33s
```

And now, the by-author dataframe.

```
In [14]: author_dataframe = pd.DataFrame([
             process_file(i, by_author=True)
             for i in tqdm(files, desc="Generating Dataframe")
         ])
         print("Saving author dataframe to Author Data.csv.")
         author_dataframe.to_csv("corpus data files/Author Data.csv", index=False)
         print(author_dataframe)
         del author_dataframe
```

```
HBox(children=(IntProgress(value=0, description='Generating Dataframe', max=19320), HTML(value='
```

```
Saving author dataframe to Author Data.csv.
```

	Age	Author ID	Gender	Industry	\
0	37	1000331	female	indUnk	
1	17	1000866	female	Student	
2	23	1004904	male	Arts	
3	25	1005076	female	Arts	
4	25	1005545	male	Engineering	
5	48	1007188	male	Religion	
6	26	100812	female	Architecture	
7	16	1008329	female	Student	
8	25	1009572	male	indUnk	
9	27	1011153	female	Technology	
10	25	1011289	female	indUnk	
11	17	1011311	female	indUnk	
12	17	1013637	male	RealEstate	
13	23	1015252	female	indUnk	
14	34	1015556	male	Technology	
15	41	1016560	male	Publishing	
16	26	1016738	male	Publishing	
17	24	1016787	female	Communications-Media	
18	27	1019224	female	RealEstate	
19	24	1019622	female	indUnk	
20	16	1019710	male	Student	
21	25	1021779	female	indUnk	
22	23	1022037	male	indUnk	
23	17	1022086	female	Student	
24	17	1024234	female	indUnk	
25	17	1025783	female	Student	
26	23	1026164	female	Education	
27	15	1026443	female	Student	
28	16	1028027	female	indUnk	
29	26	1028257	male	Education	
...	...	...	...	...	
19290	25	970774	male	Banking	
19291	24	971317	female	indUnk	
19292	34	973521	female	Religion	
19293	23	97387	male	Arts	
19294	17	977947	female	indUnk	
19295	14	978748	male	Student	
19296	26	979795	male	Technology	
19297	25	980769	male	indUnk	
19298	16	980975	female	indUnk	
19299	16	983202	male	indUnk	
19300	24	983609	male	Non-Profit	
19301	27	984615	female	Technology	
19302	23	986006	female	indUnk	
19303	23	987163	female	Education	
19304	16	987614	female	Student	
19305	17	988941	female	Student	

19306	24	989359	female	Arts
19307	23	990045	female	indUnk
19308	24	991110	female	Education
19309	17	992078	male	indUnk
19310	34	99290	male	Internet
19311	25	99382	male	indUnk
19312	25	993945	female	HumanResources
19313	27	994348	female	indUnk
19314	17	994616	male	Student
19315	36	996147	female	Telecommunications
19316	25	997488	male	indUnk
19317	16	998237	female	indUnk
19318	27	998966	male	indUnk
19319	25	999503	male	Internet

	Posts	Sign
0	Well, everyone got up and going this morning...	Leo
1	Yeah, sorry for not writing for a whole there...	Libra
2	cupid,please hear my cry, cupid, please let y...	Capricorn
3	and did i mention that i no longer have to de...	Cancer
4	B-Logs: The Business Blogs Paradox urlLink Hi...	Sagittarius
5	1/03 DrKioni.com Awarded ByRegion.net Healers...	Libra
6	Friday My dear wife was walking on her gradua...	Aries
7	Sorry, but I gotta..I couldn't remember the w...	Pisces
8	Planning the Marathon I checked Active.com, a...	Cancer
9	The astute among you will note that this run ...	Virgo
10	MSN conversation: 11.17am Iggbalbollywall (t...	Libra
11	Hey!!! Tonight kids, I saw Harry Potter and t...	Scorpio
12	Hey you clowns and kids with Down's. Sorry I ...	Virgo
13	aint no such things as halfway crooks. this i...	Pisces
14	well, the retreat was a success, i think. at ...	Virgo
15	In his urlLink February 14, 2003 post Ray Mat...	Sagittarius
16	as astute followers of the urlLink Assistant ...	Libra
17	You love me... I have you here by my side... ...	Leo
18	Dear Susan, You are a fat Wombat... I hate yo...	Libra
19	yay! it changed! :) i think i get it now. you...	Aquarius
20	Yes i survived not eating for 24 hours, I am ...	Pisces
21	IN THE SPACESHIP, THE SILVER SPACESHIPï¿½ I f...	Scorpio
22	Aaaaaahhhh... mornings! know that it is s...	Cancer
23	All right, I officially live in THE nut house...	Cancer
24	wait...now dat i think about it...dat GBC thi...	Libra
25	Lily's Lullaby- Part SIX It was cold that eve...	Gemini
26	I've posted a new test on my AIM profile...yo...	Aries
27	Hello hello hello again. It's lovely to speak...	Scorpio
28	OK here goes nothing, I'm starting over. My o...	Libra
29	Wednesday of this week, my wife Elizabeth and...	Aries
...	...	...
19290	hello if im a sex and the city fanatic, beb...	Taurus

```
19291   My very own blog. Reading anya's stuff made m...     Aquarius
19292   The Canadian Anglican Church's top governing ...       Taurus
19293   Folx the good news is that I am most probably...  Sagittarius
19294   Pondering the Mountaintop Hello everybody! I ...        Aries
19295   Yeah, I decided I want to start blogging agai...     Aquarius
19296   One Hundred Fourteen: The divine beauty Of he...       Taurus
19297   In politics there are certain goals, such as ...    Capricorn
19298   Well today was such a change in life... I ha...       Pisces
19299   GARR!R!!!!!!! Ok, in my german class today we...       Gemini
19300   Tradition Every Halloween week, since I could...       Gemini
19301   Happy Thanksgiving! For those of us who are i...     Aquarius
19302   Can you name these totally awesome chicks fro...       Taurus
19303   Arun: I'm not deviant! Mell: You're Canadian!...          Leo
19304   urlLink cool anime pic.. haha=).. quite into ...       Gemini
19305   No one has joined yet, because I've only just...    Capricorn
19306   I guess I was sort of trying to ignore it, bu...  Sagittarius
19307   (Read of the mo: The End of the Affair - Grah...       Gemini
19308   The other night a friend of mine commented th...     Aquarius
19309   Bush LIED ? That's not possible!!! If your sa...      Scorpio
19310   Another day another post. Anyone else really ...  Sagittarius
19311   GLORP! You will be able to view the old site ...     Aquarius
19312   I AM FREE, FREE AT LAST! Yes people, I have f...          Leo
19313   POX OFF! I was missed by Claudia for days, an...       Pisces
19314   urlLink mental garage (she's nubs) I dont kno...        Libra
19315   Today's Babbling It's grey. It's snowy. And I...          Leo
19316   i'm not feeling introspective today, so i wil...       Cancer
19317   yea well lets see tonight theres a show but i...        Virgo
19318   2:12AM 11/13. I am watching Martha Stewart ma...       Taurus
19319   I have done such a good job for so long pusin...       Cancer

[19320 rows x 6 columns]
```

There are a lot of entries in these dataframes. And the saved CSV is 765MB (!!!) on my machine.

```
In [15]: num_words = sum(len(i.split()) for i in blog_dataframe["Post"])
         num_authors = blog_dataframe['Author ID'].nunique()
         print(f"Number of posts:             {blog_dataframe.shape[0]:,}")
         print(f"Number of authors:           {num_authors:,}")
         print(f"Approximate number of words: {num_words:,}")
```

```
Number of posts:             681,288
Number of authors:           19,320
Approximate number of words: 136,854,709
```

Now, let's look at some of the ways we can process this text with various libraries. We'll use the very first blog post as an working example to show what some of these processes do.

```
In [16]: demo_post = list(blog_dataframe["Post"])[0]
         print(demo_post)
```

```
 Well, everyone got up and going this morning. It's still raining, but that's okay with me. Sort
```

6.2 Before we continue: some basic NLP ideas

6.2.1 The big challenge: Sparsity

NLP is fundamentally a branch of machine learning (ML) that focuses just on language data. As such, it inherits a lot of ideas and concerns from ML. One of the biggest ones is *sparsity*. Sparsity is the phenomenon of having a lot of features with "null" values for a lot of your observations (this usually manifests as "missing data" or "zero").

In language, sparsity is everywhere, since *most utterances don't use most features.* So if we want to categorize any non-trivial corpus of language data, we'll need a *lot* of features, but most of them won't appear in most of our utterances.

Sparsity is bad. It makes models have less data to work with for finding patterns, and thus, models will tend to overfit–they'll fit very well to the *current data,* but will generalize poorly to new data. Thus, almost all of the work in NLP is actually geared at finding ways to reduce sparsity in language data.

6.2.2 Data at scale

All of the models of language used in the NLP community are essentially data-driven. Rules-based models are all but completely dead.

This means that models love data. More data is better than less, and will make better models, since they're ultimately statistically-based models.

But more data means more computing time. So, we strive to build models that balance computational efficiency against power and accuracy. As it turns out, one of the best ways to do this is to get a basic, but working, model of language, and just train it on a massive dataset. Your results may not be perfectly accurate for every single document, but they should be pretty accurate for the whole dataset, and generally accurate for most documents, if you do it right.

Bear in mind: computational approaches care about balancing precision and recall, but they often come at the expense of the other.

6.2.3 A good enough model of language*: Just count the words!

* *At least, for many basic purposes.*

One of the ways to deal with sparsity is to just ignore certain classes of features and focus on a richer subset. As it turns out, we can do a surprisingly large amount by ignoring every aspect of language except for *what lexical units are used* and *how often they are used*. I.e., ignore syntax, ignore pragmatics, ignore even semantics–just count words.

It's a brutally simplistic model of language. But for many tasks, it works, and it is useful. Consider: if my task is to find out "what are people talking about," e.g. tracking discussions on Twitter at a large scale, looking at the words used is probably going to be my fastest way to get to that.

Equally important, this is a *computationally* simple, straightforward, and fast approach.

6.2.4 Reducing sparsity through preprocessing

One of the most important ways to reduce sparsity is through preprocessing your data. We want to "condense" the data down to a still meaningful representation that irons out all the noise. E.g., we often want everything to be in the same casee, because computers think that "WORD" (in uppercase) is different from "word", "Word", "WoRd", etc, and does not see any similarities between these tokens unless we explicitly tell it.

When we're doing a word-counting type analysis, remember that we're trying to get at a dense representation of the *content* of our data. Some common preprocessing steps to that end are: * Convert to lowercase * (Sometimes, but not always) De-accent characters, e.g. convert "â" to "a". * Tokenize, i.e. split a doucment into a list of tokens (words, punctuation, etc) * Remove tokens: * *Stopwords* (generally, *function* words), e.g. "the", "a", "to", etc. * Words with very low frequencies (contain very little information, and are not useful) * Words with extremely high *document-wise* frequencies (these don't help us discriminate between documents in our corpus) * Stem, or lemmatize, the text (not always–depends on the analysis!) * Stemming is significantly faster, but lemmatization can be more accurate. * But for downstream tasks and analysis, both stemming and lemmatization tend to perform about the same. * Find multi-word phrases * This might add features, but those features may be more meaningful than individual word counts.

In almost all cases, these steps are followed by generating a *vectorized* representation of the text–e.g. Bag-of-Words, Word2Vec, Doc2Vec, GloVe, etc. These vector representations can then be used for any quantitative, statistical, or machine learning analyses, e.g. regressions and classifications.

For other analyses (rather than just counting words), we might be interested in preserving more sophisticated, perhaps structural, features of the text: * Identifying (named) entities * Identifying noun chunks (more or less NPs/DPs) * Syntactic parsing (dependency and constituent parsing are most common) * Part-of-speech tagging

As for the actual analyses we might do, they are many: * Topic modeling * Document scoring/classification (e.g., author identification)

The rest of this notebook is a whirlwind tour through some of these capabilities in Python.

6.3 Preprocessing

We can use the two excellent libraries we've already seen: spaCy and Gensim.

Gensim is a much *faster* library for preprocessing, since it only operates based on raw string patterns and is designed to be fast and scale to massive datasets. There is the assumption that any error we induce through the very simplistic approaches will be balanced out by the amount of data we work with–generally a good, and correct, assumption. Gensim does no syntactic parsing, POS tagging, or other such structure-related analysis, but it's not designed for that.

spaCy is the much *more accurate* library, since it parses text based on large, powerful pre-trained models (think Stanford's CoreNLP toolkit–it's very much analogous to that). While still very fast, spaCy is painfully slow compared to Gensim. But, it has a far more robust tokenizer, it can do part-of-speech tagging, lemmatization, dependency parsing, entity and noun chunk identification, and it even has pre-trained word vectors (and can easily compute vectors for strings of words or entire documents).

6.3.1 Gensim

Let's first look at Gensim's preprocessing. There's one function–`gensim.parsing.preprocessing.preprocess_string()`–which encompasses almost all the basic functions we need: lowercasing, de-accenting, tokenizing, stemming (with the Porter stemmer algorithm), stopword removal, removal of numbers, and removal of very short words (which are generally noise to use).

We can also use the Phrases()/Phraser() objects we saw before to find multi-word phrases with ease, though given the size of our demo post, we won't see any show up. We probably want to do this *after* we run the main preprocessing.

```
In [17]: from gensim.models.phrases import Phrases
         from gensim.parsing.preprocessing import preprocess_string

         processed = preprocess_string(demo_post)
         phrases = Phrases(processed)
         processed = list(phrases[processed])

         print("Original post:")
         print(demo_post)

         print("\nAfter Gensim preprocessing:")
         print(" ".join(processed))
```

```
c:\users\andersonh\appdata\local\programs\python\python36\lib\site-packages\gensim\models\phrase
  warnings.warn("For a faster implementation, use the gensim.models.phrases.Phraser class")


Original post:
 Well, everyone got up and going this morning. It's still raining, but that's okay with me. Sort

After Gensim preprocessing:
got go morn rain okai sort suit mood easili stai home bed book cat lot rain peopl wet basement l
```

Notice how the Porter Stemmer finds the uninflected forms of words. It bases its processing *purely* on the *letters of a word*. This makes it fast, but it doesn't always give real words or the most correct forms, e.g. `"morning"` –> `"morn"`, and `"okay"` –> `"okai"`. But, for most text mining or NLP tasks, this is actually not that big of an issue. This only *really* matters, in any practical sense, for human interpretation–which, admittedly, is a non-trivial concern.

6.3.2 spaCy

spaCy requires us to load models in, as we saw earlier when doing stem-based concordancing. As before, we'll use their small English model, but we would just change the model name in `spacy.load()` if we wanted a different model. The small model will generally be a bit faster, which is all we need for this demo's purposes.

As before, applying the spaCy pipeline is easy, though we do need to manually filter our stopwords and punctuation with some explicit checks. And as with the Gensim examples, we'll run the Phrases() model on our post, though as before we won't see any changes.

```
In [18]: %%time

         from gensim.models.phrases import Phrases
         import spacy

         nlp = spacy.load("en_core_web_sm")
         processed = [
             i.lemma_
             for i in nlp(demo_post)
             if i.is_stop == False
             and i.is_punct == False
         ]
         phrases = Phrases(processed)
         processed = list(phrases[processed])

         print("Original post:")
         print(demo_post)

         print("\nAfter spaCy preprocessing:")
         print(" ".join(processed))
```

```
c:\users\andersonh\appdata\local\programs\python\python36\lib\site-packages\gensim\models\phrase
  warnings.warn("For a faster implementation, use the gensim.models.phrases.Phraser class")


Original post:
 Well, everyone got up and going this morning. It's still raining, but that's okay with me. Sort

After spaCy preprocessing:
  well get go morning -PRON- be rain be okay sort suit mood -PRON- easily stay home bed book cat
Wall time: 1.03 s
```

Notice how the processed text is much more *human-readable* with this approach (this is due to the use of a lemmatizer, rather than a stemmer). While nice for reporting and inspecting results, the extra overhead in runtime (not evident in this small example) might make this an unreasonable proposition for large datasets if time is an issue. And, as mentioned earlier, if you're doing an *automated analysis* of your text later, there isn't always a big difference, if any, in how well stemming versus lemmatization performs.

6.4 A Detour through Linguistic Analysis with spaCy

spaCy's language models have a LOT of functionality. Let's look at just some of the most easily accessible ones.

First, we've already seen spaCy's ability to do lemmatization, stopword tagging, and punctuation tagging.

```
In [19]: import spacy
```

```
        nlp = spacy.load("en_core_web_sm")

        print(f"{'Token':<15s}\t{'Lemma':<15s}\t{'Is stopword?':<15s}\t Is punctuation?")
        for i in nlp(demo_post)[:15]:
            token = i.text
            lemma = i.lemma_
            is_stop = str(i.is_stop)
            is_punct = str(i.is_punct)
            print(f"{token:<15s}\t{lemma:<15s}\t{is_stop:<15s}\t{is_punct}")
```

Token	Lemma	Is stopword?	Is punctuation?
		False	False
Well	well	False	False
,	,	False	True
everyone	everyone	True	False
got	get	False	False
up	up	True	False
and	and	True	False
going	go	False	False
this	this	True	False
morning	morning	False	False
.	.	False	True
It	-PRON-	False	False
's	be	False	False
still	still	True	False
raining	rain	False	False

And, we can also do part-of-speech tagging.

```
In [20]: print(f"{'TOKEN':<15s}\t{'COARSE POS':<17s}\t{'FINE POS'}")
         for i in nlp(demo_post)[:15]:
             token = i.text
             coarse = i.pos_
             fine = i.tag_
             print(f"{token:<15s}\t{coarse:<17s}\t{fine}")
```

TOKEN	COARSE POS	FINE POS
	SPACE	
Well	INTJ	UH
,	PUNCT	,
everyone	NOUN	NN
got	VERB	VBD
up	PART	RP
and	CCONJ	CC
going	VERB	VBG
this	DET	DT
morning	NOUN	NN
.	PUNCT	.

```
It                      PRON                    PRP
's                      VERB                    VBZ
still                   ADV                     RB
raining                 VERB                    VBG
```

Entity recognition...

```
In [21]: from pprint import pprint
         ents = nlp(demo_post).ents
         pprint(list(ents))
```

```
[this morning,
 26 degrees,
 Friday,
 next week,
 Winnipeg,
 an "Old Testament",
 CBC Radio,
 One last week,
 Floods]
```

Noun chunk identification...

```
In [22]: from pprint import pprint
         noun_chunks = nlp(demo_post).noun_chunks
         pprint(list(noun_chunks))
```

```
[everyone,
 It,
 me,
 Sort of suits,
 my mood,
 I,
 bed,
 my book,
 the cats,
 a lot,
 rain,
 People,
 wet basements,
 lakes,
 golf courses,
 fields,
 everything,
 it,
 26 degrees,
 Friday,
```

```
 we,
 mosquitos,
 I,
 Winnipeg,
 an "Old Testament" city,
 it,
 Floods,
 infestations]
```

And, as we might suspect from the above information, spaCy also does dependency parsing.

```
In [23]: print(f"{'TOKEN':<15s}\t{'HEAD':<15s}\t{'DEPENDENCY RELATION':<20s}\t{'CHILDREN'}")
         for i in nlp(demo_post)[:25]:
             token = i.text
             head = i.head.text
             dep = i.dep_
             children = ", ".join(c.text for c in i.children if not c.is_punct)
             print(f"{token:<15s}\t{head:<15s}\t{dep:<20s}\t{children}")
```

```
TOKEN                   HEAD                    DEPENDENCY RELATION         CHILDREN
                        Well
Well                    got                     intj
,                       got                     punct
everyone                got                     nsubj
got                     got                     ROOT                        Well, everyone, up, an
up                      got                     prt
and                     got                     cc
going                   got                     conj                        morning
this                    morning                 det
morning                 going                   npadvmod                    this
.                       got                     punct
It                      raining                 nsubj
's                      raining                 aux
still                   raining                 advmod
raining                 raining                 ROOT                        It, 's, still, but, 's
,                       raining                 punct
but                     raining                 cc
that                    's                      nsubj
's                      raining                 conj                        that, okay, with
okay                    's                      acomp
with                    's                      prep                        me
me                      with                    pobj
.                       's                      punct
Sort                    of                      advmod
of                      suits                   advmod                      Sort
```

We can use the built-in disiplaCy tool to generate a visualization of the dependency parse (though only of the first sentence, for space's sake):

```
In [24]: from spacy import displacy
         displacy.render(
             nlp("Well, everyone got up and going this morning."),
             style="dep",
             jupyter=True # to make this render correctly in the Jupyter notebook
         )
```

```
<IPython.core.display.HTML object>
```

There's more that spaCy can do, and there are other models and libraries available for doing this sort of automated parsing and annotation of text (e.g., there are interfaces to Stanford's CoreNLP suite), but spaCy is always a good bet since it's fast (for the amount of work it does), pretty accurate, easy to use, and flexible.

6.5 Topic Modeling

Topic Modeling refers to a wide variety of algorithms that are used to explore and discover "topics" within a corpus. "Topic" is being used with a very specific meaning here–a topic is a *statistical distribution of words*. You might already be familiar with Latent Semantic Analysis (LSA; sometimes called Latent Semantic Indexing, or LSI), which is an older model for this sort of analysis.

Most modern algorithms are based on Latent Dirichlet Allocation (LDA), which uses word co-occurrences within documents to determine the topics. LDA has given rise to a number of subsequent topic models: * Author-Topic models, which are LDA with metadata (usually, but not always, the author of a piece) * Dynamic topic models, which include time metadata in the modeling process * Hierarchical Dirichlet Process, an extension of LDA that is *nonparametric* with regards to the number of topics (but can give less clear results in some cases).

All of these models are implemented in Genim. Due to the size of our corpus and the fact that this is a live demo, we'll use Gensim's speedier preprocessing tools to work with our data and prepate it for topic modeling. And we'll only show LDA, since the others do the same basic thing and look basically the same in code.

Gensim requires that the corpus be in a *bag of words* format for topic modeling, so we'll need to put our documents in that format first. Fortunately this requires little code: just do our preprocessing, then use some pre-built tools from Gensim to do the rest.

```
In [25]: %%time

         from gensim.corpora import Dictionary
         from gensim.corpora.mmcorpus import MmCorpus
         from gensim.models.phrases import Phrases, Phraser
         from gensim.parsing.preprocessing import preprocess_string
         from tqdm import tqdm_notebook as tqdm

         # preprocess our corpus
         corpus = [
             preprocess_string(i)
             for i in tqdm(blog_dataframe["Post"], desc="Preprocessing")
         ]
```

```
        phrases = Phrases(tqdm(corpus, desc="Phrase-finding"), min_count=100)
        print("Generating Phraser() object for faster phrasing.")
        phrases = Phraser(phrases)
        corpus = list(phrases[tqdm(corpus, desc="Phrasing")])
        id2word = Dictionary(tqdm(corpus, desc="id2word"))
        vocabsize = len(id2word)
        # remove tokens with extremely high or low frequencies
        id2word.filter_extremes(
            no_above=.5,  # remove tokens in > 50% of the documents (default)
            no_below=5,   # remove tokens in < 5 documents (default)
            keep_n=500000 # only keep 500k tokens, max--up from default 100k for good measure
        )
        # Reset index spacings for better efficiency
        id2word.compactify()
        print(f"Removed {vocabsize - len(id2word)} tokens based on frequency criteria.")
        corpus = [
            id2word.doc2bow(i)
            for i in tqdm(corpus, desc="BoW")
        ]
```

```
HBox(children=(IntProgress(value=0, description='Preprocessing', max=681288), HTML(value='')))
```

```
HBox(children=(IntProgress(value=0, description='Phrase-finding', max=681288), HTML(value='')))
```

```
Generating Phraser() object for faster phrasing.
```

```
HBox(children=(IntProgress(value=0, description='Phrasing', max=681288), HTML(value='')))
```

```
HBox(children=(IntProgress(value=0, description='id2word', max=681288), HTML(value='')))
```

```
Removed 507524 tokens based on frequency criteria.
```

```
HBox(children=(IntProgress(value=0, description='BoW', max=681288), HTML(value='')))
```

```
Wall time: 22min 17s
```

Around this time we should also notice that we're using a *huge* amount of RAM. So much, in fact, that the computer might be close to running out. (It turns out that it's almost entirely from the dataframe still in memory, but let's just pretend for the moment that it's our actual corpus). How do we deal with this? Simple: *streaming.* Gensim is built around the concept of working with data one chunk at a time–so we can save our data to a file, then read one "chunk" of that file at a time! When dealing with a few hundred, or a few thousand, documents this isn't needed. But when dealing with *millions* of documents or more, it's absolutely required, unless you want to spend thousands of dollars on extremely high-end computing hardware or cloud computing time/space.

Our corpus doesn't actually require this. (The cell below will show it's only using a few megabytes–perfectly reasonable). But we'll do it anyways, just for demonstration's sake, since this same idea and approach would scale up nicely.

In [26]:
```
%%time

from sys import getsizeof

# getsizeof returns bytes--convert to megabytes
# nb: factors of 1024, not 1000
id2word_size = getsizeof(id2word) / 1024
corpus_size = getsizeof(corpus) / 1048576
blog_size = getsizeof(blog_dataframe) / 1048576
phrases_size = getsizeof(phrases) / 1048576

print(f"id2word dictionary size in memory: {id2word_size:.2f}KB")
print(f"Gensim corpus size in memory: {corpus_size:.2f}MB")
print(f"Blog Dataframe size in memory: {blog_size:.2f}MB")
print(f"Phrases model size in memory: {phrases_size:.2f}MB")

# Delete that massive blog dataframe first--we already saved
# it to file.
del blog_dataframe

print("Saving id2word dictionary.")
id2word.save("corpus data files/id2word")
print("Serializing bag-of-words corpus.")
MmCorpus.serialize(
    fname="corpus data files/bow_corpus.mm",
    corpus=corpus,
    id2word=id2word
)
print("Corpus and id2word dict saved.")

# we can reload the id2word and corpus data from the files
```

```
        # we just saved, too.
        del corpus
        del id2word

        # Phrases can be retrained in a few minutes, and
        # the training is faster than the appliction of
        # a trained model, so we can delete this.
        del phrases
```

```
id2word dictionary size in memory: 0.05KB
Gensim corpus size in memory: 5.83MB
Blog Dataframe size in memory: 969.19MB
Phrases model size in memory: 0.00MB
Saving id2word dictionary.
Serializing bag-of-words corpus.
Corpus and id2word dict saved.
Wall time: 1min 13s
```

When re-reload the corpus, we'll see that it's *much* smaller in memory. This is because, using Gensim's tools, we're only looking at the size of the *thing that accesses the data,* but which does not currently store any of the data–it's all in a file on disk.

Accessing data from disk will be slow, though. Even the fastest SSDs are still at least 10x slower than even slow RAM. So our runtime will be limited by *disk access speed*. BUT, disk space is *significantly* cheaper than RAM space! At the time of writing, 16 gigabytes of RAM costs about $200*. Meanwhile, $200 can get you between 7 and 10 *terabytes* of hard drive space (less if you want SSDs, though). So while reading corpora off disk is very slow, it lets us work with *much* larger datasets.

We can re-load the corpus and dictionary we just saved. The dictionary didn't take up much memory at all, but saving a copy was a good idea anyways. The corpus also didn't take much space.

**RAM prices are currently (as of early 2018) very highly inflated, though; normally this much RAM should only cost about $100. But the point still stands: RAM is expensive, hard drives are dirt cheap.*

```
In [27]: print("Loading id2word dictionary.")
         id2word = Dictionary.load("corpus data files/id2word")
         print("Loading bag-of-words corpus.")
         corpus = MmCorpus("corpus data files/bow_corpus.mm")
```

```
Loading id2word dictionary.
Loading bag-of-words corpus.
```

6.5.1 Aside: spaCy bag-of-words pipeline

We won't run this during this demo, but just for comparison, here's a spaCy preprocessing pipeline that does the same thing. The only thing that changes is the first pass through the corpus (the first `corpus = [...]` bit)–the bag-of-words steps are as before. The change is shown below–all the other code would be the same.

```
import spacy

nlp = spacy.load("en_core_web_sm", disable=["parser", "tagger", "ner"])
corpus = [
    [
        i.lemma_
        for i in nlp(j)
        if i.is_stop == False
        and i.is_punct == False
    ]
    for j in tqdm(texts, desc="Preprocessing")
]
```

When I ran this on my computer, it took about an hour and a half to run (~200 documents per second), compared to the Gensim pipeline which took about 11 minutes (~1000 documents per second). Of course, if you remove the `disable` line, you'll get much higher-quality results, but it will run about 10x slower (~20 documents per second on my computer, nearly 12 hours total runtime).

However, given the size of the corpus we're working from, the differences aren't actually that significant in terms of how they'll affect our results, so we can go with the considerably faster Gensim approach.

6.5.2 Back to topic models

Now, let's run some of these topic models. We won't bother tweaking any of the default settings (except for chunksize, which should give us a bit more speed), meaning each one will search for 100 topics. This is the most important parameter in the models, by far–and sadly, the only really good way to find a good value is to run it at a range of different topic numbers and see what gives you useful results. You *can* look at the *coherence* of each topic (calculated by Gensum automatically) and use that to evaluate your model, but the be-all-end-all is the human interpretability and the usefulness of your topics.

These models will take a while. So we can skip down to the next cell and just load the models back up from disk.

We also need to do a *term frequency-inverse document frequency* transform on the corpus. This is a weighting scheme that converts the raw word counts into values that *decrease* the weight of a word based on how many documents it appears in (more documents –> proves less informatio about any individual document, so decrease the weight), and *increases* it based on how often it appears *within the current document* (more occurrences –> more important to the document).

```
In [28]: from gensim.corpora import Dictionary
         from gensim.corpora.mmcorpus import MmCorpus
         from gensim.models.ldamulticore import LdaMulticore
         from gensim.models.atmodel import AuthorTopicModel
         from gensim.models.hdpmodel import HdpModel
         from gensim.models.tfidfmodel import TfidfModel
         from tqdm import tqdm_notebook as tqdm

In [29]: id2word = Dictionary.load("corpus data files/id2word")
         corpus = MmCorpus("corpus data files/bow_corpus.mm")
```

```
        tfidf_model = TfidfModel(tqdm(corpus, desc="TFIDF Fitting"))
        # TfidfModel returns a generator.  We want it as a list to
        # re-use it for all the models.
        corpus = list(tfidf_model[tqdm(corpus, desc="TFIDF Transforming")])
```

```
HBox(children=(IntProgress(value=0, description='TFIDF Fitting', max=681288), HTML(value='')))
```

```
HBox(children=(IntProgress(value=0, description='TFIDF Transforming', max=681288), HTML(value=''
```

```
Serializing tf-idf corpus.
```

Gensim has a multi-threaded LDA implementation that can take advantage of multi-core processors for significant speedups; we'll use that.

```
In [30]: %%time

        # corpus = MmCorpus.load("corpus data files/tfidf_corpus.mm")
        print("Running LDA model.")
        lda = LdaMulticore(corpus, workers=3, id2word=id2word, chunksize=50000)
        print("Saving LDA model to file.")
        lda.save("model files/LDA.model")
        print("Done.")
```

```
Running LDA model.
```

```
c:\users\andersonh\appdata\local\programs\python\python36\lib\site-packages\gensim\models\ldamod
  diff = np.log(self.expElogbeta)
```

```
Saving LDA model to file.
Done.
Wall time: 15min 6s
```

Let's look at some of the LDA outputs and see if we can interpret them. We'll look at only the top ten highest-likelihood topics, sorted by decreasing likelihood. Since we did no tweaking of the model parameters (most notably, the number of topics), we shouldn't expect to see very coherent topics. Of course, tweaking these parameters is largely a guessing game, involving tweaks followed by manually inspecting the results. This is beyond the scope of this talk, so for now it's enough to show how a basic LDA model is run in Gensim.

```
In [31]: from pprint import pprint
         print("Loading LDA model from file.")
         lda = LdaMulticore.load("model files/LDA.model")
         pprint(lda.top_topics(corpus, topn=10)[:10])

         # Delete the model and corpus to conserve RAM space.
         # We've already saved them to disk, so it can be reloaded.
         del lda
         del corpus

Loading LDA model from file.
[([(0.0032867447, 'urllink'),
   (0.0027863914, 'like'),
   (0.0026201669, 'know'),
   (0.0025476436, 'love'),
   (0.0024109564, 'want'),
   (0.0023818174, 'feel'),
   (0.0023229425, 'time'),
   (0.0022997775, 'thing'),
   (0.0022506749, 'think'),
   (0.0022419214, 'dai')],
  -0.991923335075992),
 ([(0.0044276407, 'urllink'),
   (0.0023756386, 'like'),
   (0.0022402934, 'know'),
   (0.0020682053, 'dai'),
   (0.0020471383, 'think'),
   (0.0020066646, 'thing'),
   (0.001993226, 'look'),
   (0.0019889951, 'time'),
   (0.0019226717, 'want'),
   (0.0018823334, 'good')],
  -1.0021072909489532),
 ([(0.0031452833, 'urllink'),
   (0.002618549, 'like'),
   (0.0023778444, 'love'),
   (0.0023155215, 'know'),
   (0.0022554197, 'time'),
   (0.0022332505, 'dai'),
   (0.0022041786, 'thing'),
   (0.0021714524, 'think'),
   (0.0020662425, 'want'),
   (0.0020051461, 'feel')],
  -1.0371516683648605),
 ([(0.0037324338, 'urllink'),
   (0.0025839917, 'like'),
   (0.0023313707, 'know'),
   (0.0022363507, 'time'),
```

```
  (0.0022093586, 'dai'),
  (0.0021959213, 'think'),
  (0.0020698688, 'want'),
  (0.0020404994, 'thing'),
  (0.0020326914, 'peopl'),
  (0.0020270168, 'work')],
 -1.037523835694815),
([(0.0033950717, 'urllink'),
  (0.002547115, 'like'),
  (0.0024895007, 'know'),
  (0.002475114, 'love'),
  (0.0022665106, 'time'),
  (0.0021941478, 'want'),
  (0.00216542, 'dai'),
  (0.0021385716, 'thing'),
  (0.0021323161, 'think'),
  (0.0020106426, 'feel')],
 -1.0389515068966084),
([(0.0040940056, 'urllink'),
  (0.0025974775, 'like'),
  (0.0022480788, 'know'),
  (0.0022426376, 'dai'),
  (0.0021083711, 'time'),
  (0.002065473, 'work'),
  (0.002014777, 'new'),
  (0.001946062, 'want'),
  (0.0019344733, 'think'),
  (0.0018878003, 'thing')],
 -1.0416680131366436),
([(0.0039211577, 'urllink'),
  (0.0035663845, 'nbsp'),
  (0.0026781426, 'work'),
  (0.0025530022, 'like'),
  (0.0023340273, 'dai'),
  (0.0022568388, 'know'),
  (0.002200635, 'think'),
  (0.002124303, 'thing'),
  (0.0020629035, 'time'),
  (0.002020318, 'go')],
 -1.0648945292671508),
([(0.0033071246, 'nbsp'),
  (0.0031024385, 'urllink'),
  (0.0029407777, 'like'),
  (0.0027964371, 'know'),
  (0.0024651678, 'dai'),
  (0.0024367403, 'think'),
  (0.002396396, 'work'),
  (0.0023568869, 'time'),
```

```
  (0.00229206, 'thing'),
  (0.0022025094, 'want')],
 -1.0649106448359777),
([(0.003832554, 'nbsp'),
  (0.0035766826, 'urllink'),
  (0.0027839274, 'like'),
  (0.0026074082, 'know'),
  (0.0024505686, 'think'),
  (0.0023473722, 'want'),
  (0.0023310697, 'work'),
  (0.002326278, 'thing'),
  (0.0022509075, 'peopl'),
  (0.002218816, 'time')],
 -1.0716893580334352),
([(0.0032079602, 'urllink'),
  (0.0028091543, 'like'),
  (0.0023741648, 'know'),
  (0.002320525, 'blog'),
  (0.0023021041, 'want'),
  (0.0022548395, 'dai'),
  (0.002238216, 'think'),
  (0.0022324405, 'time'),
  (0.0022161962, 'work'),
  (0.0021896486, 'thing')],
 -1.0718338146744584)]
```

6.6 Word Embeddings

Word embeddings have absolutely taken over the entire field of NLP, starting with Mikolov et al's 2013 paper on Word2Vec. The basic idea of these embeddings: * Some notion of "meaning" is recoverable from the *contexts* that a word occurs in * It is possible to generate a *vector* representation of a word such that *words appearing in similar contexts have similar vectors* (or, more intuitively, "are close to each other").

Word vectors are used for almost every kind of task: document scoring and classification, sentiment analysis, document and word clustering, machine translation (though less commmonly), and more. Plus, they've been demonstrated to excel at a number of lexical similarity and analogy tasks.

Gensim has an impelementation of Word2Vec that we can use on our corpus. spaCy, meanwhile, comes pre-bundled with word vectors computed using Stanford's GloVe algorithm (which works differently under the hood, but in terms of how well the vectors actually perform in any application, is basically identical).

6.6.1 Brief aside: Poincare Embeddings

In May 2017, Maximilian Nickel and Douwe Kiela published an extremely exciting paper on embeddings performed in *hyperbolic space* rather than *Euclidean space* (all prior models were in Euclidean space). These embeddings–while not yet well-suited to extracting word similarities and

meanings from large bodies of unlabeled text–show an extreme aptitude for embedding *graph*- and *network*-like data, e.g. WordNet, and can capture various relations like hypernymy/hyponymy and synonymy quite well. Keep an eye on "Poincare Embeddings"–lots of interesting things are sure to happen there soon.

6.6.2 spaCy word vectors

We'll work with spaCy's pre-trained vectors just for time's sake–training word2vec models in Gensim is surprisingly fast, but we'll be doing basically the same thing downstream. First, as always, we need to load our data and our model. We'll just use a single blog post, and we'll turn off a bunch of the spaCy parsing stuff for speed.

```
In [32]: import pandas as pd
         import spacy

         nlp = spacy.load("en_core_web_lg")

         # just use the very first post for demos
         demo_post = pd.read_csv(
             "corpus data files/Blog Data.csv",
             usecols=["Post"],
             nrows=1,
             squeeze=True
         )[0]
         demo_post = nlp(demo_post)
```

```
In [33]: print(demo_post[1])
         print(demo_post[1].vector)
```

```
Well
[-1.2486e-01  6.9180e-02 -3.1364e-01 -3.1354e-01  1.4388e-01  1.6573e-01
 -4.0073e-02 -3.4590e-01 -1.7483e-01  2.6147e+00  9.1120e-03  1.8054e-02
  8.3494e-02 -9.3186e-02 -1.0852e-01 -1.4856e-01  1.4402e-01  1.1995e+00
 -5.1814e-01 -4.3844e-02 -2.8039e-01  1.3527e-01 -3.6054e-02 -1.3734e-01
  3.1807e-02  1.7668e-02  4.7540e-02 -3.0738e-02  1.7169e-01 -1.0349e-01
  1.0784e-01  1.9757e-02  6.9675e-02 -1.5200e-01 -1.9508e-01 -1.7867e-01
  1.1583e-01  4.7459e-02 -4.5048e-02 -1.0148e-02 -6.7003e-02  3.0717e-02
 -9.5259e-02 -2.2538e-02  8.2868e-02  1.9983e-01 -1.2923e-01 -1.3680e-01
  2.9010e-02  1.8272e-01  2.2101e-02  1.5804e-01  3.7986e-02  3.1765e-02
 -3.0443e-03  3.1779e-02 -9.1168e-02  3.6951e-02  4.4161e-02  1.0407e-01
  1.5687e-02 -9.7470e-02  4.2405e-02  2.6701e-01 -1.0596e-01 -1.4289e-01
 -6.7763e-02  1.7992e-01  2.6175e-01  4.5349e-02  2.5674e-01  1.2484e-01
  3.6875e-01  7.5486e-02 -1.4867e-01  8.2897e-03  8.9685e-02 -1.3242e-01
 -2.4698e-01  2.3017e-01 -2.4372e-03  1.8298e-01 -2.2386e-01  1.7260e-01
 -2.6223e-02 -3.0136e-01 -1.8803e-01  1.0426e-01 -4.1197e-02  6.9884e-02
 -3.3139e-03 -1.4249e-01  4.4817e-02  3.6930e-01  5.2759e-01 -1.4461e-01
  2.2260e-01 -2.1296e-01 -2.1575e-01 -2.1324e-02 -2.3094e-01 -9.1427e-02
 -1.3952e-01 -6.5416e-02  9.9365e-02 -1.1914e+00 -1.0889e-01 -3.6120e-01
 -6.4289e-02 -1.4312e-01  9.1272e-03 -1.6777e-01  2.1744e-01 -4.6958e-02
```

```
 7.1629e-03 -8.1238e-03  4.3175e-02 -5.4813e-02 -1.1981e-01  4.2927e-02
 2.3405e-01  1.0804e-01  1.9480e-01 -5.6464e-02  1.2108e-01  1.0770e-01
-9.7876e-02 -2.4924e-01  2.4288e-01 -2.4311e-02 -8.8623e-02 -3.1197e-01
-1.8214e-01  2.1236e-01  2.4771e-01  3.9045e-01  1.9372e-01 -4.0742e-01
 1.3977e-01  1.5621e-01 -1.2266e+00  2.0881e-01  4.3454e-01 -1.0045e-01
-1.3093e-01 -2.2994e-01 -2.5303e-01 -5.7255e-02  4.4188e-02  3.9772e-03
-2.1284e-01  2.0268e-01  1.5135e-01 -8.1819e-03 -1.3508e-01 -6.2223e-02
 1.2250e-01 -7.0757e-02 -5.4248e-02 -4.5870e-01  2.6266e-01  1.4630e-01
 4.0067e-02 -1.7087e-01 -2.3487e-02 -2.6435e-01  5.8678e-02 -7.8438e-02
 3.0261e-01 -1.4065e-01 -5.5911e-02  1.8330e-01 -1.0979e-01 -7.1047e-02
-1.7407e-01 -1.4674e-01 -2.8062e-01  1.3984e-01  1.1339e-01  1.0495e-01
-1.4357e-01 -1.2663e-01 -3.4565e-01 -1.4041e-01 -1.2706e-01 -2.8720e-01
-2.9107e-02  5.2271e-02  1.9180e-01 -5.9188e-02  1.1118e-02 -3.6953e-02
-4.6103e-02 -5.5561e-02 -4.2408e-02 -5.4556e-02 -8.5381e-02 -2.5072e-01
 1.3272e-01  8.9385e-02  5.0679e-02 -1.4402e-01  1.5704e-02  8.2595e-02
 2.8950e-01  1.1995e-01  8.2218e-03  8.6943e-02 -1.4487e-01 -6.4140e-02
-2.8995e-01 -4.3396e-02  1.0147e-01 -3.6916e-01  1.3197e-01  2.2112e-01
 2.2586e-01 -1.3024e-01  3.4924e-02  1.7595e-01 -5.5850e-04  1.4002e-01
 7.2031e-02  9.4518e-02  2.5377e-01  1.5240e-01 -9.7878e-02  8.5805e-02
 3.9108e-02  1.6801e-01 -2.7371e-01 -8.4246e-02 -1.9891e-01  1.9488e-01
 1.0244e-01 -2.1371e-01  1.9158e-01 -4.9702e-01 -7.6887e-02  1.3054e-01
 1.4384e-01  9.2271e-02 -2.4776e-01  2.9863e-01  4.5713e-01  6.3761e-02
-2.8826e-01 -1.8834e-01 -6.4870e-02  2.0241e-01  2.9338e-01 -1.1416e-01
-2.9192e-01 -1.3655e-01  5.2752e-02  2.9894e-02  3.6916e-01  1.2743e-02
-2.2503e-01  8.0966e-02  3.2966e-01  1.3930e-01 -7.8549e-02  1.1004e-01
-9.0624e-02 -1.9984e-02  2.2853e-02  3.1763e-03  6.1005e-01  2.6677e-01
 4.9331e-02 -2.5631e-01 -2.4592e-01 -3.0870e-01 -4.1584e-01  3.6741e-01
 1.0777e-01 -1.3235e-02 -8.0141e-02  4.4847e-01  2.7414e-01  1.1039e-01
-8.1114e-02 -1.6639e-01  1.8136e-02  7.6002e-02  2.0605e-01 -1.8203e-01
 2.9575e-01  5.4778e-02 -4.6968e-01  1.5817e-02 -2.2619e-01  1.1062e-02
 1.8545e-01 -1.1914e-01  2.1583e-01 -4.0342e-01  1.7759e-01  8.9240e-02]
```

```
In [34]: print(f"{'Token':<15s}\tSimilarity to '{demo_post[1].text}'")

         for i in demo_post[:15]:
             # try/except because some token comparisons throw errors on spaCy's end
             try:
                 print(f"{i.text:<15s}\t{demo_post[1].similarity(i):.3f}")
             except:
                 pass
```

```
Token                   Similarity to 'Well'
Well                    1.000
everyone                0.648
got                     0.552
up                      0.602
and                     0.682
```

```
going                0.631
this                 0.604
morning              0.387
It                   0.714
's                   0.412
still                0.710
raining              0.262
```

Word vectors can also be extended to computer vectors for entire documents. Usually, this is done by simply adding or averaging the vectors for each individual word in the document. (Incidentally, this exact same approach lets you get a vector for any arbitrary bit of text!)

```
In [35]: print(demo_post.vector)
```

```
[ 1.31642865e-02  2.37451762e-01 -1.33002967e-01 -1.58166736e-01
  9.16091949e-02  4.56015505e-02 -1.51400210e-03 -1.62576497e-01
 -7.07884729e-02  2.12497878e+00 -1.87663123e-01  2.05625482e-02
  7.54114017e-02 -6.97415769e-02 -1.99725553e-01 -5.24708331e-02
 -2.69564018e-02  1.06616330e+00 -1.82011917e-01 -4.86053340e-02
 -5.35848886e-02  2.18726844e-02 -5.59682995e-02 -3.07797678e-02
  2.45223865e-02 -1.28578171e-02 -1.20025635e-01 -3.46809365e-02
  7.66581818e-02 -7.82760903e-02 -2.33661868e-02  3.36482115e-02
 -4.14343439e-02  2.24824902e-02  5.23805320e-02 -4.10539694e-02
  4.96976711e-02  3.10765095e-02 -5.10399900e-02 -3.25182676e-02
 -1.58267170e-02  3.63390930e-02  2.94042882e-02 -3.33312154e-02
  2.69403644e-02  1.00509606e-01 -1.64681092e-01 -4.72940765e-02
  5.93143664e-02 -5.76425642e-02 -5.66710643e-02  9.40750092e-02
 -2.73930281e-02  2.85059139e-02  4.11496833e-02 -6.96792779e-03
 -1.54412789e-02 -4.68339510e-02  5.63570932e-02 -3.10750045e-02
 -6.35085851e-02 -5.31633161e-02 -4.81112249e-04  1.96307719e-01
  1.30728194e-02 -7.01612234e-02 -2.12039668e-02  4.46476564e-02
 -2.95005795e-02  1.79688618e-01  4.91093583e-02  1.00777403e-01
  1.98810115e-01 -6.47725305e-03  8.86943787e-02  5.91295697e-02
  9.21648890e-02  3.84680904e-03 -6.41978085e-02  1.93143263e-01
 -4.62955497e-02  8.67331773e-02 -6.86863884e-02 -1.38426665e-02
  4.33658510e-02 -2.46838838e-01  1.19494922e-01 -7.30767846e-02
  2.77736813e-01  1.34293601e-01 -1.80763267e-02 -1.45396953e-02
  4.63578245e-03  5.40753379e-02  1.41394556e-01 -2.58688144e-02
  1.91447865e-02  5.76385530e-03 -5.65039255e-02  1.07032163e-02
 -2.87960656e-02  4.17623743e-02 -1.27336815e-01 -5.02941720e-02
  6.15321919e-02 -5.78176796e-01  5.08880280e-02 -7.64341326e-03
  1.52477883e-02  1.23977093e-02  3.75351422e-02 -1.55868232e-01
  1.34658828e-01 -1.35198385e-01 -4.05474156e-02 -4.07232419e-02
  1.43934898e-02 -8.09895433e-03 -1.22347800e-02 -9.73605067e-02
  6.38979748e-02  2.94547584e-02  1.99218541e-02 -7.03123733e-02
  7.94450473e-03  7.23707601e-02  1.59296431e-02 -1.17330894e-01
  1.45509187e-02 -4.16304134e-02  2.19151489e-02 -8.63423571e-02
```

```
 -7.60464519e-02  5.95238097e-02  1.16800264e-01  4.88185436e-02
 -1.56893209e-03 -1.40665406e-02  2.85443738e-02  2.45357323e-02
 -1.26371944e+00  1.50656730e-01  2.11000651e-01 -2.11142749e-03
  1.73429642e-02  2.67431065e-02 -6.83486834e-02  2.42024250e-02
  5.20412624e-03 -5.24368025e-02 -5.55904321e-02  2.24909615e-02
  1.34837732e-01  4.61816378e-02 -9.35395807e-02 -7.21071884e-02
 -8.52866769e-02 -6.69751465e-02 -4.74636741e-02 -7.50315413e-02
  7.53237261e-03  2.78376527e-02 -3.41446958e-02 -5.79334050e-02
 -3.71873602e-02 -1.12035863e-01  2.50705909e-02 -5.78852706e-02
  1.34372517e-01  1.50861293e-02 -3.01528946e-02  4.10258621e-02
 -9.67732910e-03 -4.56370153e-02 -1.12796240e-01  6.34944364e-02
  6.69909967e-03 -3.51219364e-02  8.46422929e-03 -9.01845619e-02
  2.66135065e-03 -7.27335662e-02 -1.45241603e-01 -4.90702353e-02
 -2.65349410e-02 -9.75504294e-02 -4.19219211e-02  1.10685546e-02
  8.61035287e-02  2.06585191e-02  4.93311137e-02  7.12807402e-02
 -1.00086510e-01  2.63198689e-02 -1.67035796e-02  1.25243828e-01
 -1.05668139e-02 -1.19949043e-01 -2.14458443e-02  1.60552979e-01
 -9.00433362e-02 -7.66610503e-02 -5.50686717e-02  1.10140545e-02
  1.95168465e-01 -1.84920114e-02  4.93176617e-02  1.34883001e-02
  7.02754483e-02 -3.39424312e-02 -8.85552689e-02 -7.08585307e-02
  4.05594474e-03 -1.70666635e-01  2.76947170e-02  9.22666490e-02
 -4.02171798e-02 -3.87982689e-02 -1.57628641e-01  7.38738105e-02
  1.36203943e-02  6.59089442e-03 -2.76750401e-02  6.13207519e-02
  2.38623507e-02 -2.92682089e-03 -5.13713285e-02  8.41584802e-02
 -5.21717183e-02  2.70015523e-02 -6.06019832e-02  6.67507872e-02
  9.67155546e-02  9.47949141e-02 -1.70890838e-02 -6.73476160e-02
  2.93387454e-02 -1.62215143e-01 -3.64345051e-02  1.06720850e-01
  1.13053983e-02  5.38144931e-02 -5.98892234e-02  8.59192312e-02
  9.49497223e-02 -5.38322888e-02 -6.84643313e-02 -1.06010884e-01
 -9.19809863e-02  1.01062536e-01  4.23326939e-02 -8.91820937e-02
 -9.48470924e-03 -5.99794555e-03  5.45678847e-02  1.95466697e-01
  9.99818891e-02 -8.39695185e-02 -1.06593117e-01  6.10944480e-02
  1.49142385e-01  1.10690340e-01 -3.90812680e-02  7.86308348e-02
  9.38537791e-02 -2.91924439e-02 -5.98246008e-02  1.69358682e-02
  2.10155666e-01  5.94129860e-02 -5.30707985e-02 -5.44334985e-02
 -9.95674431e-02 -1.54171541e-01 -1.36872202e-01  3.95603441e-02
 -3.21762450e-02  7.44344369e-02 -1.45444786e-03  1.98339060e-01
  2.58016914e-01 -7.31731132e-02 -2.78430469e-02 -8.77422020e-02
 -4.37610596e-02 -3.28980982e-02  1.48785040e-01 -1.25699237e-01
  1.48799390e-01 -2.67258789e-02 -1.65224969e-01  1.99796744e-02
 -1.94624401e-04 -2.79833637e-02 -1.45414320e-03  1.93717834e-02
 -3.79961953e-02 -6.66726455e-02 -2.77482066e-02  3.50829475e-02]
```

```
In [36]: print(f"{'Token':<15s}\tSimilarity to whole document")
         for i in demo_post[:15]:
             print(f"{i.text:<15s}\t{demo_post.similarity(i):<.3f}")
```

```
Token                   Similarity to whole document
```

```
                0.000
Well            0.832
,               0.559
everyone        0.743
got             0.699
up              0.755
and             0.714
going           0.788
this            0.715
morning         0.581
.               0.626
It              0.840
's              0.530
still           0.805
raining         0.413
```

6.7 Classification and Regression with Text

Since we have vectors representing documents (we can also use the bag-of-words representation for this, too!), we can do any classical statistical test or modelling we want, like trying to build a model to predict the age of a post's author from its contents, or predicting their gender. But we won't use statistics, because we have tools better suited to the specifics of the tasks at hand: machine learning.

Without getting too deep in the weeds, machine learning models are essentially extensions of statistics that are based on *computational mathematics* rather than *classical mathematics*. They tend to scale better to massive datasets (i.e. run faster and predict better) and are often more powerful for pure prediction, but they are designed for larger datasets and may not work well at lower sizes.

However, ultimately, the distinction between "machine learning" and "statistics" is a false one. The real difference is "experimentalists" (who seek to understand *causation*) and "data miners," who only want to build models with good *predictive power*. (I won't get any further into this–but Leo Breiman's 2001 paper Statistical Modeling: The Two Cultures is an interesting exploration of this divide).

Python has some absoltuely top-tier machine learning libraries: Scikit-Learn for non-neural models (e.g. decision trees, random forests, support vector machines), and various environments like Keras, Tensorflow, and Pytorch for building neural networks. We'll use non-neural models for this demo–they both run faster (as in, by a factor of days, sometimes) and are *much* easier to interpret.

We'll build six models, using two different sets of target variables and three different sets of predictors.

Targets: * Author age (regression) * Author gender (binary classification) * Author's astrological sign (multi-class classification)

Predictors: * Document vectors, generated by the Doc2Vec algorithm, which we will train ourselves. * Bag-of-Words model, *without* tf-idf weighting (it ends up being only marginally helpful for the model performance of SVMs, but detrimental to the model interpretability)

We'll just one type of model–Support Vector Machines–for both regression and classification. SVMs strike a nice balance between speed (though there are faster models) and performance (though there are more powerful models), and are always a good go-to. For binary classification

tasks, they're still among the best models around. Specifically, we'll use SVMs with a *linear kernel*–don't worry for now about the impelementation details of that, just know that a linear kernel SVM runs faster than other kinds of SVM, and is often plenty good for most tasks. W

(Spoilers: our predictions will be kinda crap)

```
In [37]: # we need a lot more imports for this than before
         from gensim.corpora import Dictionary
         from gensim.corpora.mmcorpus import MmCorpus
         from gensim.matutils import corpus2csc
         from gensim.models import doc2vec
         from gensim.models.callbacks import CallbackAny2Vec
         from gensim.models.phrases import Phrases, Phraser
         from gensim.models.tfidfmodel import TfidfModel
         from gensim.utils import simple_preprocess
         import numpy as np
         import pandas as pd
         from sklearn.svm import LinearSVC, LinearSVR
         from sklearn.metrics import f1_score, make_scorer
         from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
         from sklearn.metrics.pairwise import cosine_similarity
         from sklearn.preprocessing import MaxAbsScaler, Normalizer
         from scipy.sparse import csr_matrix
         import spacy
         from tqdm import tqdm_notebook as tqdm

         from pprint import pprint
         import re
```

```
In [38]: print("Reading in target data.")
         targets = pd.read_csv(
             "corpus data files/Author Data.csv",
             usecols=["Age", "Gender", "Sign"],
             squeeze=True
         )
         ages = targets["Age"].values
         genders = targets["Gender"].values
         signs = targets["Sign"].values
```

```
Reading in target data.
```

```
In [39]: from collections import Counter
         # Look for bad values we might need to remove;
         # the Counter() will tell us how many of each
         # value we have in our dataset.
         pprint(Counter(ages))
         pprint(Counter(genders))
         pprint(Counter(signs))
```

```
Counter({17: 2381,
         16: 2152,
         23: 2026,
         24: 1895,
         15: 1771,
         25: 1620,
         26: 1340,
         14: 1246,
         27: 1205,
         13: 690,
         33: 464,
         34: 378,
         35: 338,
         36: 288,
         37: 259,
         38: 171,
         39: 152,
         40: 145,
         41: 139,
         42: 127,
         43: 116,
         45: 103,
         44: 94,
         48: 77,
         46: 72,
         47: 71})
Counter({'female': 9660, 'male': 9660})
Counter({'Virgo': 1783,
         'Cancer': 1722,
         'Libra': 1700,
         'Taurus': 1645,
         'Scorpio': 1631,
         'Leo': 1619,
         'Aries': 1597,
         'Gemini': 1595,
         'Pisces': 1580,
         'Sagittarius': 1549,
         'Aquarius': 1474,
         'Capricorn': 1425})
```

Fortunately, all of our categorical data is pretty well-balanged (i.e., there isn't too huge of a discrepancy between the number of observations in each class). For classification tasks this is extremely important, sometimes moreso than regression tasks.

Now, we *could* get the GloVe vectors from spaCy, but this will take a pretty long time (about 5 hours on my computer, by my estimation). So we won't do that. But if we wanted to, the code would look like this:

```
In [ ]: %%time
```

```
print("Loading spaCy model.")
# Only large model has pre-trained word vectors.
# Disable the parser, tagger, and named entity recognizer
# for extra speed in this step.
nlp = spacy.load("en_core_web_lg", disable=["parser", "tagger", "ner"])
print("Loading blog posts.")
glove_corpus = list(pd.read_csv(
    "corpus data files/Author Data.csv",
    usecols=["Posts"],
    squeeze=True
))
print("Retrieving vectors.")
glove_corpus = np.array([
    nlp(i).vector
    for i in tqdm(glove_corpus, desc="spaCy vectors")
])
# Cast to a 32-bit floating point format (smaller memory/on-disk size)
# and save.  This means we don't have to re-run the above time-
# consuming steps every time we want to do some work with this
# matrix.
np.save(
    "corpus data files/GloVe matrix.npy",
    glove_corpus.astype(np.float32)
)
```

This might not be the most practical way to get vectors for large amounts of text due to runtime, but the vectors you get are pretty much guaranteed to be high-quality (assuming, of course, you're working on data that's more or less general English, rather than being highly domain-specific, e.g. financial filings or court decisions). For comparison (both of code and of downstream results), we'll also use Doc2Vec, as implemented in Gensim, to generate our own word vectors. Doc2Vec is one of the many vectorization/embedding approaches, but rather than working at the word level, it works directly at the document level. We'll use a less aggressive preprocessing, since vectorization models are less sensitive to sparsity; this one just de-accents, lowercases, and removes tokens by length. We'll leave all settings at default (so, mininum length 2, maximum length 15).

We'll also use 300-dimensional vectors for our embeddings, which is a pretty standard size for production code. Note that in general, higher dimensional embeddings (e.g. 300 vs 100) will give better, more accurate results, but will be sparser and take longer to run models on.

Note that we're not applying most of the usual preprocessing steps we've already seen. We're just casting to lowercase and splitting at word boundaries with a regular expression. We won't even bother with phrase-finding for this. We're doing this because vectorization algorithms–especially those based on Word2Vec, including Doc2Vec–aren't as senstivie to sparsity as Bag of Words models, and since different inflectional forms of a word might actually occur in consistently different contexts that we wish to capture.

```
In [40]: %%time
```

```
splitter = re.compile(r"[a-z0-9']")
print("Reading blog data.")
df = pd.read_csv(
    "corpus data files/Author Data.csv",
    usecols=["Posts"],
    squeeze=True
).values
df = [
    splitter.findall(i.lower())
    for i in tqdm(df, desc="Preprocessing")
]
# phrasing
# phrases = Phrases(tqdm(df, desc="Phrase-finding"), min_count=100)
# print("Creating Phraser() object for faster phrasing.")
# phrases = Phraser(phrases)
# df = phrases[tqdm(df, desc="Applying phraser")]
df = [
    doc2vec.TaggedDocument(i, [j])
    for j,i in enumerate(df)
]

print("Done.")
```

```
Reading blog data.
```

```
HBox(children=(IntProgress(value=0, description='Preprocessing', max=19320), HTML(value='')))
```

```
Done.
Wall time: 2min 5s
```

We'll create an EpochLogger class from Gensim's CallbackAny2Vec class. The only thing this will do for us is that, when we pass it to our Doc2Vec model, it will print out the progress during training. This doesn't affect the code, but like the tqdm() calls scattered throughout this notebook, it just lets us keep track of progress.

In [41]:
```
# class to let us track training progress
# through Gensim's callbacks interface
class EpochLogger(CallbackAny2Vec):
    "Callback to log information about training"
    def __init__(self):
        self.epoch = 1

    def on_epoch_begin(self, model):
        print("Epoch #{} start".format(self.epoch))
```

```
            def on_epoch_end(self, model):
                print("Epoch #{} end".format(self.epoch))
                self.epoch += 1
```

And now, we run the Doc2Vec model and save it to file for potential later use.

```
In [42]: print("Beginning Doc2Vec.")
         epoch_logger = EpochLogger()
         d2v = doc2vec.Doc2Vec(
             df,
             min_count=5,
             epochs=25,
             vector_size=300,
             window=5,
             worker=3,
             callbacks=[epoch_logger]
         )

         print("Doc2Vec model trained.  Saving model to file.")
         d2v.save("model files/Doc2Vec")
         print("Model saved.")
```

```
Beginning Doc2Vec.
Epoch #1 start
Epoch #1 end
Epoch #2 start
Epoch #2 end
Epoch #3 start
Epoch #3 end
Epoch #4 start
Epoch #4 end
Epoch #5 start
Epoch #5 end
Epoch #6 start
Epoch #6 end
Epoch #7 start
Epoch #7 end
Epoch #8 start
Epoch #8 end
Epoch #9 start
Epoch #9 end
Epoch #10 start
Epoch #10 end
Epoch #11 start
Epoch #11 end
Epoch #12 start
Epoch #12 end
Epoch #13 start
```

```
Epoch #13 end
Epoch #14 start
Epoch #14 end
Epoch #15 start
Epoch #15 end
Epoch #16 start
Epoch #16 end
Epoch #17 start
Epoch #17 end
Epoch #18 start
Epoch #18 end
Epoch #19 start
Epoch #19 end
Epoch #20 start
Epoch #20 end
Epoch #21 start
Epoch #21 end
Epoch #22 start
Epoch #22 end
Epoch #23 start
Epoch #23 end
Epoch #24 start
Epoch #24 end
Epoch #25 start
Epoch #25 end
Doc2Vec model trained.  Saving model to file.
Model saved.
```

```
In [43]: print("Loading model from file.")
         d2v = doc2vec.Doc2Vec.load("model files/Doc2Vec")
         d2v_corpus = np.array([d2v.infer_vector(i.words) for i in tqdm(df)])
         # df takes up a good chunk of memory, and we're done with it,
         # so delete it to conserve RAM.
         del df
         print("Done.  Saving document vectors.")
         np.save("corpus data files/Doc2Vec Matrix.npy", d2v_corpus)
         print("Done.")
```

```
Loading model from file.
```

```
HBox(children=(IntProgress(value=0, max=19320), HTML(value='')))
```

```
Done.  Saving document vectors.
Done.
```

And now we re-generate our bag-of-words representation for the by-author corpus and save it to disk. Note that we could do tf-idf scaling here, and it might help out results a bit, and it would probably help our performance a bit, but we won't–it will make interpreting our model considerably more difficult. Instead, we'll just apply a standard scaling on the sparse matrix, scaling every feature such that the maximum absolute value is 1. (we won't center it to zero mean and unit variance–that would destroy the sparsity and we couldn't load the matrix into memory)

In [44]:
```
%%time

from gensim.parsing.preprocessing import preprocess_string
from gensim.models.phrases import Phrases, Phraser

print("Reading in posts.")
corpus = list(pd.read_csv(
    "corpus data files/Author Data.csv",
    usecols=["Posts"],
    squeeze=True
))

# preprocess our corpus
corpus = [
    preprocess_string(i)
    for i in tqdm(corpus, desc="Preprocessing")
]
phrases = Phrases(tqdm(corpus, desc="Phrase-finding"), min_count=100)
corpus = list(phrases[tqdm(corpus, desc="Phrasing")])
id2word = Dictionary(tqdm(corpus, desc="id2word"))
vocabsize = len(id2word)
# remove tokens with extremely high or low frequencies
id2word.filter_extremes(
    no_above=.5,  # remove tokens in > 50% of the documents (default)
    no_below=5,   # remove tokens in < 5 documents (default)
    keep_n=500000 # only keep 500k tokens, max--up from default 100k for good measure
)
# Reset index spacings for better efficiency
id2word.compactify()
print(f"Removed {vocabsize - len(id2word)} tokens based on frequency criteria.")
corpus = [
    id2word.doc2bow(i)
    for i in tqdm(corpus, desc="BoW")
]

print("Serializing corpus.")
id2word.save("corpus data files/author_id2word")
MmCorpus.serialize(
    fname="corpus data files/author_corpus.mm",
    corpus=corpus,
    id2word=id2word
```

```
        )
```

```
Reading in posts.
```

```
HBox(children=(IntProgress(value=0, description='Preprocessing', max=19320), HTML(value='')))
```

```
HBox(children=(IntProgress(value=0, description='Phrase-finding', max=19320), HTML(value='')))
```

```
HBox(children=(IntProgress(value=0, description='Phrasing', max=19320), HTML(value='')))
```

```
c:\users\andersonh\appdata\local\programs\python\python36\lib\site-packages\gensim\models\phrase
  warnings.warn("For a faster implementation, use the gensim.models.phrases.Phraser class")
```

```
HBox(children=(IntProgress(value=0, description='id2word', max=19320), HTML(value='')))
```

```
Removed 521864 tokens based on frequency criteria.
```

```
HBox(children=(IntProgress(value=0, description='BoW', max=19320), HTML(value='')))
```

```
Serializing corpus.
Wall time: 19min 59s
```

```
In [45]: print("Loading bag-of-words (non tf-idf) corpus.")
         corpus = list(MmCorpus("corpus data files/author_corpus.mm"))
         corpus = corpus2csc(corpus).transpose().tocsr()
         print("Scaling bag-of-words features.")
         corpus = MaxAbsScaler().fit_transform(corpus)
```

```
Loading bag-of-words (non tf-idf) corpus.
Scaling bag-of-words features.
```

Now let's do the actual modeling. We'll use Support Vector Machines for classification and regression; these are pretty powerful models and generally pretty good choices. Others, which we won't use here because they take longer to run, include Random Forests, Adaboost, Stochastic Gradient Descent (which is fast, but has a *lot* of parameters to tune), and neural networks.

We will:

- Use a grid search to exhaustively search for good parameters (within a pre-defined search space). This is called *(hyper)parameter optimization.*
- Use 3-fold cross-validation to assess model performance for each parameter combination we try.
  - Data is split into 3 equally sized "folds." (usually, 5 or 10 folds would be used, but for time's sake, 3 will suffice here).
  - Four folds are used to train the model; the fifth is used to assess its performance.
  - All permutations of "train-on-2, test-on-1" are performed, and scores are averaged to get the overall model performance.
- Print out the best model parameters and the best score for each of our nine feature-target combinations.
- "Open up" one of the models and look at the feature weights it learned.

Cross-fold validation is often used in place of statistical significance testing in the machine learning world. There are a lot of reasons for this, and the differences are pretty interesting, but for our purposes, we only need to know two things to understand why we prefer cross-validation to significance testing. 1. In most practical cases, ML models are being used with an eye towards strong *predictive power* rather than strong *explanatory power.* Thus, we're more interested in assessing how well our model generalizes to new data rather than the statistical significance of relationships we uncover. 2. Cross-fold validation evaluates the model on data that it did not see during training, approximating the process of feeding novel data into the model. This is far more reliable on very large, randomly sampled datasets.

In [53]:
```
from sklearn.externals import joblib

# Load our vector corpus from file and normalize
# each observation (not feature) to have magnitude 1
print("Loading and Doc2Vec corpus.")
d2v_corpus = np.load("corpus data files/Doc2Vec matrix.npy")
d2v_corpus = Normalizer().fit_transform(d2v_corpus)
print("Done.")


# Functions to optimize our classifiers/regressors.
def fit_svr(data, labels, outfile):
    crossval = GridSearchCV(
        estimator=LinearSVR(),
        param_grid={
            "C":np.logspace(-5, 1, 7),
        },
        n_jobs=3,
```

```
            verbose=3,
            cv=3,
        )
        crossval.fit(data, labels)
        joblib.dump(crossval.best_estimator_, outfile)

        return crossval.best_score_, crossval.best_params_

    def fit_svc(data, labels, outfile):
        crossval = GridSearchCV(
            estimator=LinearSVC(),
            param_grid={
                "C":np.logspace(-5, 1, 7)
            },
            n_jobs=3,
            verbose=3,
            cv=3,
            scoring=make_scorer(f1_score, average="macro")
        )
        crossval.fit(data, labels)
        joblib.dump(crossval.best_estimator_, outfile)

        return crossval.best_score_, crossval.best_params_
```

```
Loading and Doc2Vec corpus.
Done.
```

Now, let's optimize some models! We'll save the best-performing models to file for each set.

In [54]:
```
%%time

from sklearn.externals import joblib

print("Beginning CSR-Age regression fit.")
age_bow = fit_svr(corpus, ages, "model files/age_bow.pkl")

print("Beginning Doc2Vec-Age regression fit.")
age_d2v = fit_svr(d2v_corpus, ages, "model files/age_d2v.pkl")

print("Beginning CSR-Gender binary classification fit.")
gender_bow = fit_svc(corpus, genders, "model files/gender_bow.pkl")

print("Beginning Doc2Vec-Gender binary classification fit.")
gender_d2v = fit_svc(d2v_corpus, genders, "model files/gender_d2v.pkl")

print("Beginning CSR-Sign classification fit.")
sign_bow = fit_svc(corpus, signs,"model files/sign_bow.pkl")
```

```
        print("Beginning Doc2Vec-Sign classification fit.")
        sign_d2v = fit_svc(corpus, signs, "model files/sign_d2v.pkl")
```

```
Beginning CSR-Age regression fit.
Fitting 3 folds for each of 7 candidates, totalling 21 fits


[Parallel(n_jobs=3)]: Done  21 out of  21 | elapsed:  2.0min finished


Beginning Doc2Vec-Age regression fit.
Fitting 3 folds for each of 7 candidates, totalling 21 fits


[Parallel(n_jobs=3)]: Done  21 out of  21 | elapsed:    3.3s finished


Beginning CSR-Gender binary classification fit.
Fitting 3 folds for each of 7 candidates, totalling 21 fits


[Parallel(n_jobs=3)]: Done  21 out of  21 | elapsed:  1.9min finished


Beginning Doc2Vec-Gender binary classification fit.
Fitting 3 folds for each of 7 candidates, totalling 21 fits


[Parallel(n_jobs=3)]: Done  21 out of  21 | elapsed:   17.9s finished


Beginning CSR-Sign classification fit.
Fitting 3 folds for each of 7 candidates, totalling 21 fits


[Parallel(n_jobs=3)]: Done  21 out of  21 | elapsed: 20.3min finished


Beginning Doc2Vec-Sign classification fit.
Fitting 3 folds for each of 7 candidates, totalling 21 fits


[Parallel(n_jobs=3)]: Done  21 out of  21 | elapsed: 22.2min finished


Wall time: 1h 35s
```

We can quickly compute the approximate F1 scores for randomly guessing by randomly permuting the target classification variables and calculating the F1 scores from treating these permutations as our predictions.

```
In [55]: from sklearn.preprocessing import LabelEncoder

         genders_ = LabelEncoder().fit_transform(genders)
         signs_ = LabelEncoder().fit_transform(signs)

         genders_f1_chance = f1_score(
             genders_,
             np.random.permutation(genders_),
             average="macro"
         )
         signs_f1_chance = f1_score(
             signs_,
             np.random.permutation(signs_),
             average="macro"
         )

In [56]: print("CSR-Age model:")
         print(f"Best R^2 score:  {age_bow[0]}")
         print(f"Best parameters: {age_bow[1]}")

         print("\nDoc2Vec-Age model:")
         print(f"Best R^2 score:  {age_d2v[0]}")
         print(f"Best parameters: {age_d2v[1]}")

         print("\nCSR-Gender model:")
         print(f"Best F1 score:   {gender_bow[0]}")
         print(f"Chance F1 score: {genders_f1_chance}")
         print(f"Best parameters: {gender_bow[1]}")

         print("\nDoc2Vec-Gender model:")
         print(f"Best F1 score:   {gender_d2v[0]}")
         print(f"Chance F1 score: {genders_f1_chance}")
         print(f"Best parameters: {gender_d2v[1]}")

         print("\nCSR-Astrological Sign model:")
         print(f"Best F1 score:   {sign_bow[0]}")
         print(f"Chance F1 score: {signs_f1_chance}")
         print(f"Best parameters: {sign_bow[1]}")

         print("\nDoc2Vec-Astrological Sign model:")
         print(f"Best F1 score:   {sign_d2v[0]}")
         print(f"Chance F1 score: {signs_f1_chance}")
         print(f"Best parameters: {sign_d2v[1]}")

CSR-Age model:
Best R^2 score:  0.047575101844957314
Best parameters: {'C': 0.1}
```

```
Doc2Vec-Age model:
Best R^2 score:  0.18963770676288672
Best parameters: {'C': 10.0}

CSR-Gender model:
Best F1 score:   0.7567812859164255
Chance F1 score: 0.4988612836438923
Best parameters: {'C': 0.01}

Doc2Vec-Gender model:
Best F1 score:   0.6401506929615315
Chance F1 score: 0.4988612836438923
Best parameters: {'C': 0.1}

CSR-Astrological Sign model:
Best F1 score:   0.08757137308243307
Chance F1 score: 0.08286388077593226
Best parameters: {'C': 10.0}

Doc2Vec-Astrological Sign model:
Best F1 score:   0.08716301883213448
Chance F1 score: 0.08286388077593226
Best parameters: {'C': 10.0}
```

Now, for kicks, let's open up one of the models–the age-bow model–and see what it learned. SVMs are linear models, so we can just look at the coefficient weights.

```
In [62]: age_bow = joblib.load("model files/age_bow.pkl")
         coefs = age_bow.coef_
         # Zip up the coefficients with the actual text
         # of the tokens, for human interpretability.
         id2word = Dictionary.load("corpus data files/author_id2word")
         coefs = [(id2word[i], coefs[i]) for i in range(len(coefs))]
         coefs = sorted(coefs, key=lambda x: abs(x[1]), reverse=True)
         for i in coefs[:25]:
             print(f"{i[0]:<25s} {i[1]}")
```

```
xanga                     -1.9668928332057503
boredom                   -1.8040857581198255
bore                      -1.7574341127238464
common_test               -1.5383788467231674
sec                       -1.452939663775986
bestest_friend            -1.4480233805637324
saw_shrek                 -1.3607749270471736
xanga_com                 -1.3518653749844385
dad                       -1.318006354165074
daughter                  1.2881148284677584
```

skateboard	-1.2838153196574003
local	1.2572066491487446
summer_vacat	-1.2390704503589176
awesom	-1.2380422349340612
sian	-1.2335334775650209
best_friend	-1.2067217024609447
sunburn	-1.2061339708563275
ti	-1.2037574041500079
recess	-1.1926773118455363
newborn	1.1726798398733633
prefect	-1.1691740438884966
dislik	-1.1669345610542374
immatur	-1.147140623661155
song_lyric	-1.1452647578274544
tml	-1.1316503417267156