Natural Language Processing

Signal Data Science

In this lesson, we will focus on natural language processing (NLP), *i.e.*, the application of data science to human-generated text. To illustrate the diversity and depth of modern NLP, we will proceed through a series of classical exercises in both Python and R.

Natural language processing is particularly enjoyable because its results tend to be very human-interpretable. Be sure to note any project ideas which occur to you while working through this assignment.

Email spam classification

In 2009, Symantec estimated that almost 90% of global email traffic consisted entirely of spam.[1] Modern email providers make extensive use of machine learning techniques to automatically classify and divert spam emails from your inbox. Without those algorithms, the enormous amount of spam received on a daily basis would be overwhelming!

We will begin with naive Bayes spam filtering, one of the oldest methods of statistical spam classification. After writing our own naive Bayes classifier and implementing various improvements, we will then compare its performance to that of elastic net regularized logistic regression. For training our model, we will use the CSDMC2010 SPAM corpus, downloadable online and located in the `csdmc2010-spam` dataset folder.

- Download the CSDMC2010 SPAM dataset. Examine several emails labeled as spam and several emails labeled as ham. (We will refer to "not-spam" emails as "ham" for convenience.[2])

[1] Symantec, September 2009, Report #33, State of Spam: A Monthly Report: "Overall spam volumes averaged at 87 percent of all email messages in August 2009. Health spam decreased again this month and averaged at 6.73 percent, while over 29 percent of spam is Internet related spam. Holiday spam campaigns have begun leveraging Halloween and Christmas, following closely after Labor Day-related spam."

[2] "Ham" is a commonly used term for not-spam, not just something we made up.

- Load the CSDMC2010 SPAM training data into R, storing the text of each `.eml` file in a string. (You may find `list.files()` and `scan()` useful.) Use the entirety of each file, including the HTML tags and email headers.

Naive Bayes classification

Using a naive Bayes classifier for spam filtration dates as far back as 1998.[3] The basic idea behind Bayesian spam filtering to classify spam based on which words appear or don't appear in any given email. For example, words such as "Viagra" or "refinance" show up often in spam emails, whereas words such as "brunch" are more likely to show up in non-spam emails. Bayes' theorem can therefore be used (with some simplifying assumptions) to train a spam vs. not-spam classifier based on the presence or absence of various words.

Three important assumptions are made in Bayesian spam filtering:

1. First, each email is treated as a bag of words in which the *order* of words is irrelevant, which enormously simplifies the task at hand (otherwise we might want to look at every possible *pairing* of words, or every possible *triplet* of words, and so on and so forth).
2. Second, we consider only the *presence* or *absence* of each word. That is, if the word is present, we do not consider its *frequency* in the email.
3. Third, the presence of each word in an email is assumed to be *statistically independent* from the presence of each other word. The assumption of *independence of events* is the core assumption of the naive Bayes method, turning a very computationally difficult problem into a tractable one.

We will proceed to process the data accordingly.

- Randomly select 80% of the emails as a training set, leaving the remaining 20% as the test set.
- Create two data frames from the training and test sets such that each row corresponds to a single email, each column corresponds to a particular word, and each entry is 0 or 1 depending on whether or not the column's corresponding word is present in the row's corresponding email. For now, consider a "word" to be any sequence of non-space characters without further characters on either side; *e.g.*, in `"Lorem4 ipsum; dolor. Sit <b>amet</b>"`, the words are `"Lorem4"`, `"ipsum;"`, `"dolor."`, `"Sit"`, and `"<b>amet</b>"`. The columns for the two data frames should *not*

[3] See Sahami *et al.* (1998), A Bayesian Approach to Filtering Junk E-Mail: "In addressing the growing problem of junk E-mail onthe Internet, we examine methods for the automated construction of filters to eliminate such unwanted messages from a user's mail stream. By casting this problem in a decision theoretic framework, we are able to make use of probabilistic learning methods in conjunction with a notion of differential misclassification cost to produce filters which are especially appropriate for the nuances of this task."

be identical (*i.e.*, there should be words in the training set which do not appear in the test set and vice versa). (You may find `strsplit()` helpful.)

Consider a subset of the words in the entire training corpus consisting of n words. Let w_i, S, and H respectively denote events corresponding to word i being present in an email, an email being spam, and an email being ham, and let W denote the *conjunction* of all the events w_i, *i.e.*, the event corresponding to words 1 through n all being present in an email. From Bayes' theorem, we can write an expression for $P(S \mid W) = P(S \mid w_1, w_2, \ldots, w_n)$, the probability of a message being spam given that it contains words 1 through n.

$$P(S \mid W) = \frac{P(W \mid S)P(S)}{P(W)} = \frac{P(W \mid S)P(S)}{P(W \mid S)P(S) + P(W \mid H)P(H)}.$$

Note that since $P(W \mid S)P(S) = P(W \text{ and } S)$, we can rewrite it as

$$P(W \mid S)P(S) = P(w_1 \mid w_2, \ldots, w_n, S)P(w_2 \mid w_3, \ldots, w_n, S) \cdots P(w_n \mid S)P(S)$$

and similarly for $P(W \mid H)$.

We make the *naive* assumption that the presences of the words are conditionally independent, *i.e.*, that $P(w_i \mid w_j) = P(w_i)$ for $i \neq j$. We then arrive at

$$P(W \mid S) = P(w_1 \mid S)P(w_2 \mid S) \cdots P(w_n \mid S)P(S).$$

This simplification is the key to a naive Bayes classifier. Although the assumption of conditional independence is often false, it nevertheless yields serviceable results in practice.

- For each word in the training set, calculate $P(w_i \mid S)$, *i.e.*, the proportion of spam emails out of all the emails in which the word appears. If a word appears only in spam emails, assign it a probability of $P(w_i \mid S) = 0.999$ instead of 1; similarly, if a word appears only in ham emails, assign it a probability of 0.001 rather than 0.

It is simplest to assume *a priori* that any given email is equally as likely to be spam or ham, *i.e.*, $P(S) = P(H) = \frac{1}{2}$. Then our expression for $P(S \mid W)$ simplifies into

$$P(S \mid W) = \frac{P(w_i \mid S) \cdots P(w_n \mid S)}{}.$$

Due to floating-point underflow, instead of calculating $P(S)$ directly, it is better to calculate

$$\log\left(\frac{1}{P(S)} - 1\right) = \sum_{i=1}^{N} (\log(1-p_i) - \log p_i)$$

because the summation doesn't have problems with underflow due to multiplying many small numbers together, and to then calculate $P(S)$ after a numeric expression for the right side of the above equation has been obtained.

- Look at the words with the highest and lowest p_is. Interpret the results.
- Calculate the true positive, true negative, false positive, and false negative rates for your classifier.
- Modify your classifier so that it converts all uppercase characters to lowercase. Does this improve the performance of your classifier on the training data?
- Find examples of both spam and non-spam emails from your personal email accounts. See if your classifer classifies them correctly.

Using *n*-grams with logistic regression

We can compare our naive Bayes classifier with logistic regression. For our logistic regression, we will use as features the frequency counts of individual words, *i.e.*, the number of time each word we know appears in an email. Additionally, we will also use *n*-grams, which are sequences of n consecutive words.

- Use the `ngram` package to create a dataframe of 1-grams and 2-grams from the training data with the `ngram()` and `get.phrasetable()` functions. Each row should represent a particular email and each column should be one of the 1-grams or 2-grams.
- Use regularized elastic net logistic regression to predict spam vs. not-spam, selecting the hyperparameters α and λ with the `caret` package.
  - To reduce computational demands, restrict consideration to the 1000 most common n-grams.
- Compute the true positive, true negative, false positive, and false negative rates for you logistic regression spam classifier. Compare its performance to that of your naive Bayes spam classifier.

Sentiment analysis of Github commit logs

Linus Torvalds, the creator of the Linux kernel, is well-known for a very direct manner of communication. For example, in 2012, he wrote the following about a proposed kernel patch:

> On Sun, Dec 23, 2012 at 6:08 AM, Mauro Carvalho Chehab `<mchehab@redhat.com>` wrote:
>
>> Are you saying that pulseaudio is entering on some weird loop if the returned value is not -EINVAL? That seems a bug at pulseaudio.
>
> Mauro, SHUT THE FUCK UP!
>
> It's a bug alright - in the kernel. How long have you been a maintainer? And you *still* haven't learnt the first rule of kernel maintenance?
>
> If a change results in user programs breaking, it's a bug in the kernel. We never EVER blame the user programs. How hard can this be to understand?
>
> To make matters worse, commit f0ed2ce840b3 is clearly total and utter CRAP even if it didn't break applications. ENOENT is not a valid error return from an ioctl. Never has been, never will be. ENOENT means "No such file and directory", and is for path operations. ioctl's are done on files that have already been opened, there's no way in hell that ENOENT would ever be valid.
>
>> So, on a first glance, this doesn't sound like a regression, but, instead, it looks tha pulseaudio/tumbleweed has some serious bugs and/or regressions.
>
> Shut up, Mauro. And I don't *ever* want to hear that kind of obvious garbage and idiocy from a kernel maintainer again. Seriously.
>
> I'd wait for Rafael's patch to go through you, but I have another error report in my mailbox of all KDE media applications being broken by v3.8-rc1, and I bet it's the same kernel bug. And you've shown yourself to not be competent in this issue, so I'll apply it directly and immediately myself.
>
> WE DO NOT BREAK USERSPACE!

... and so on and so forth. Torvalds has commented similarly regarding the usage of `overflow_usub()`, GitHub pull requests, and a host of other topics.

But is Linus's personality as well reflected in his GitHub commit messages? Let's compare his activity to that of Bram Moolenaar, creator, maintainer, and benevolent dictator for life of the Vim text editor. First, we will scrape the commit messages for all their contributions to their respective projects using the GitHub API; afterward, we will perform sentiment analysis on their commit messages to determine if such a difference exists and, if so, how large it is.

Using the Github API

Since Linus and Bram make most of their commits to Linux and Vim respectively, we can use the API to (1) get some of the latest commits to Linux and Vim and (2) strip out all of the commits which don't come from either of them.

The Github API can be accessed directly via your browser. In general, you begin with the url `https://api.github.com/` and then successively append text to it, *e.g.*, `https://api.github.com/users/JonahSinick`.

- Referencing the API documentation on commits, figure out which API queries will return the latest commits for Linux and for Vim. (A parameter beginning with a colon (`:`) is a *variable* which you should fill in with the appropriate value.)
- Write a Python script to access the Github API and download our desired data. Follow these specifications:
  - Use `urllib.request` to download the results of API calls. Use the `json` module, particularly the `loads()` function, to strip out all the commit messages which don't come from Linus or Brad.
  - Write the commit messages to two files, `linus.txt` and `brad.txt`, with one message per line.

Performing sentiment analysis

- Load the files containing the commit messages for Linus and Brad. Process them, creating one vector for Linus's messages and another vector for Brad's messages.
- Install and load the `qdap` package, which has functions for both cleaning text and performing sentiment analysis.
- Following the Cleaning Text & Debugging vignette, use `qdap` to clean the commit messages in preparation for sentiment analysis. In particular, `check_text()` should suggest the usage of some text-cleaning functions to use.
- Combine all of the commit messages into a character vector with many entries. In addition, create a vector of labels (integers 0 or 1) which indicate whether the corresponding entry in the aforementioned character vector is from Linus or Brad.
- Use `polarity()` with its default settings to perform sentiment analysis, passing in both the character vector of every commit message as well as the grouping vector.

The results of the analysis are stored in `$all`, a data frame with a column `polarity` for the sentiment polarity score of each message.

- Plot two histograms of the polarity scores for Linus and Brad overlaid on top of each other. Interpret the results.
- Look at the commit messages which had the lowest and highest polarity scores.
- t test…

Topic modeling of Wikipedia articles

One of the largest subfields of natural language processing is the subfield of topic modeling, where we try to extract semantically meaningful "topics" from a large corpus (collection of documents). Topic modeling for text is particularly useful for the initial analysis of large corpora, where there are too many documents for a single human to read and classify them all, and for the discovery of hidden topical structure.

The most popular topic modeling technique is that of latent Dirichlet allocation (LDA) and is broadly analogous to factor analysis. From Wikipedia, here is an intuitive explanation of LDA:

> In LDA, each document may be viewed as a mixture of various topics. […]
>
> For example, an LDA model might have topics that can be classified as **CAT_related** and **DOG_related**. A topic has probabilities of generating various words, such as *milk*, *meow*, and *kitten*, which can be classified and interpreted by the viewer as "CAT_related". Naturally, the word cat itself will have high probability given this topic. The **DOG_related topic** likewise has probabilities of generating each word: *puppy*, *bark*, and *bone* might have high probability. Words without special relevance, such as *the*, will have roughly even probability between classes (or can be placed into a separate category). A topic is not strongly defined, neither semantically nor epistemologically. It is identified on the basis of supervised labeling and (manual) pruning on the basis of their likelihood of co-occurrence. A lexical word may occur in several topics with a different probability, however, with a different typical set of neighboring words in each topic.

Precisely, we first pick a number of topics k. Next, we posit a *generative model* according to the following:

1. Topics are probability distributions over the set of words used in all documents. That is, each topic is represented by the assignment of a number between 0 and 1 to each distinct word such that the sum of all

those numbers is 1. For example, if we have a very simple corpus that only has the words "a" and "b", then a possible topic T would be represented by $T(''\text{a}'') = 0.3$ and $T(''\text{b}'') = 0.7$.

2. Similarly, each document is a probability distribution over the set of topics.
3. Each document has a set number of words. Each word is generated as follows: First, we randomly pick a *topic* based on the proportions of the topics associated with its document. Next, we look at the probabilities associated with that topic and accordingly randomly choose one of the dictionary words.

The algorithm optimizes the probabilities associated with topics, documents, and words so as to maximize the *likelihood* associated with the training data. For each document, the associated distribution of topics is assumed to have a Dirichlet prior, hence the name of latent *Dirichlet* allocation. (The *latent* comes from the generative model falling into the class of *latent variable models*.)

In the following, we will see how to use latent Dirichlet allocation on the corpus of machine learning-related Wikipedia pages.

Scraping Wikipedia pages

We need to scrape the relevant Wikipedia pages from the Internet. It it often easiest to break down a scraping task into two stages: first, scraping a list of URLs to content that we want, and second, actually scraping from our list of URLs.

As such, our first task will be to look at Category:Machine_learning and its subcategories to get a list of every machine learning-related Wikipedia page. As you work, watch out for two traps: (1) infinite loops in the subcategory tree (where a category ends up being its own parent) and (2) subcategories which are *overly general* and branch out into non-machine learning related topics. Both problems cane be dealt with manually if they occur.

- Write a Python script to find the URL to every Wikipedia page in the machine learning category. Follow these specifications:
  - Use `urllib.request` to write a function `download_page(url)` which downloads the HTML of the page at `url` and returns it.
  - Using `download_page()`, download the Wikipedia page Category:Machine_learning. Write a function `get_urls(html)` which takes in the HTML of a Wikipedia category page and returns a dictionary with two entries: `pages`, a list of URLs to articles listed in the category page, and `subcategories`, a list of URLs to subcategories listed in the category page. Use Beautiful Soup to parse raw HTML. You can test your function on Category:Machine_learning to verify that it works.

  - Using `get_html()`, get a list of the links on Category:Machine_learning, and then add to the list the links on the *subcategories* of Category:Machine_learning, and then add to the list the links on the subcategories of the subcategories of Category:Machine_learning, and so on and so forth until there are no more subcategories to traverse.
  - Write the list of article URLs to a text file, `wp_ml_urls.txt`, with one URL per line.

Next, the easy part: we'll actually go through our list of URLs and download the Wikipedia articles.

- Write a Python script to download each of the Wikipedia articles in the machine learning category. Follow these specifications:
  - We need to create a folder in which we can store our downloaded HTML files. Using the `os` and `shutil` modules, check if a folder called `raw_text` exists and delete it, along with all of its contents, if it does, and then create a new folder called `raw_text`.
  - Copy and paste your `download_page()` function from before into your current script. Write a function `download_article(url)` which downloads the Wikipedia article at `url`, parses the HTML with Beautiful Soup, and returns the text of the article without any of the HTML tags. (You may find the `.get_text()` function, available for Beautiful Soup HTML objects, helpful.)
  - Read in the text file of URLs which you created earlier. Iterate over the URLs and call `download_page()` on each one. For each url `[..]/en/Page_name`, save the associated text to `raw_text/page_name.txt`; that is, take everything in the URL after the last forward slash (`/`), remove all non-alphanumeric characters, make the string completely lowercase, and use that as the file name for the article's text.

Preprocessing Wikipedia pages

In general, before using NLP techniques to analyze a corpus of text, it is useful to *pre-process* the text. Doing so correctly can improve the quality of our results substantially. We will therefore use Natural Language Toolkit (NLTK), a popular Python module for NLP-related text manipulation, to process our downloaded pages.

- In the following, retain a pristine copy of the downloaded articles in `raw_text`, *i.e.*, do not overwrite your downloaded text with its pre-processed form; if you do so and make an error in your pre-processing, you will have to repeat the entire download process. We recommend writing a single script to do all pre-processing, updating it as you go along.

- Install NLTK by running `pip install nltk`.

It's often useful to throw away information about capitalization. For the purposes of topic modeling, which is concerned with the extraction of *semantically meaningful* topics, we typically want to consider words at the beginning of a sentence to be the same as words within a sentence.

First, we want to *tokenize* each document in Python, that is, turn it into a *list* of its constituent words. NLTK provides a built-in tokenizer which splits text on both whitespace and punctuation, which is what we'll use.

- Use `wordpunct_tokenize()` from the `nltk.tokenize` module to tokenize each downloaded Wikipedia page. Turn each word into lowercase and remove all single-character tokens (corresponding to punctuation like "?" and "!" or words like "a" which we'll want to remove later anyway).

A common pre-processing step is the removal of stop words, like "the", "is", "at", and "on", from our documents. Since they occur very frequently in association with all topics, they will dominate all of our extracted topics in LDA and our results will not be meaningful.

- Following the documentation at Installing NLTK Data, install the `stopwords` corpus.

The English portion of the `stopwords` corpus can be accessed via the following Python code:

```
from nltk.corpus import stopwords
stop = stopwords.words('english')
```

Following the above:

- Using NLTK's `stopwords` corpus, remove all stop words from the tokenized Wikipedia pages.

In addition to removing stop words, we also want to remove *conjunctions*, like "or", "while", and "if", along with *determiner* words like "the", "a", "some", and "every". We can do so via part-of-speech tagging (POS tagging), which refers to the process of marking each word in a corpus with its associated part of speech, and removing words tagged as parts of speech which we wish to discard.

POS tagging is typically done through the usage of pre-trained models; we'll use NLTK's *Averaged Perceptron Tagger*.

- Download the `averaged_perceptron_tagger` model via NLTK as you did earlier with the `stopwords` corpus. In addition, download the `tagsets` dataset, containing documentation on the Averaged Perceptron Tagger's output.

The Averaged Perceptron Tagger as well as its associated documentation can be used as follows:

```
>>> nltk.pos_tag(wordpunct_tokenize('Hello world.'))
[('Hello', 'NNP'), ('world', 'NN'), ('.', '.')]
>>> nltk.help.upenn_tagset('NNP')
NNP: noun, proper, singular
    Motown Venneboerger Czestochwa Ranzer Conchita Trumplane
    Oceanside Escobar Kreisler Sawyer Cougar Yvette Ervin ODI
    Shannon A.K.C. Meltex Liverpool ...
```

Following the above:

- Use NLTK's Averaged Perceptron Tagger to remove all conjunctions and determiner words from the tokenized Wikipedia pages.

Finally, we want to somehow group together similar words, like “apples” with “apple” or “abaci” with “abacus”. We have two methods available to us: *stemming* and *lemmatization*.

Stemming will strip away everything aside from the *stem* of a word. Sometimes, the stem itself is a word, like with “cats” → “cat”. However, this is not guaranteed, like with “argue”, “argues”, “arguing” → “argu”. It is often more convenient to use lemmatization, which does not simply cut off the end of words but rather reduces different inflections into the same base form. With lemmatization, “argue”, “argues”, and “arguing” all map to “argue”. Since the base forms are all regular English words, running LDA with lemmatization is typically preferred due to ease of interpretation.

NLTK provides access to the WordNet Lemmatizer as follows:[4]

```
>>> from nltk.stem import WordNetLemmatizer
>>> print(wnl.lemmatize('churches'))
church
>>> print(wnl.lemmatize('aardwolves'))
aardwolf
>>> print(wnl.lemmatize('abaci'))
abacus
```

Following the above:

- Using the WordNetLemmatizer, lemmatize each word in the tokenized Wikipedia pages.

Finally, we're ready to turn the processed, tokenized documents back into text files.

- Save each of the tokenized documents to a `processed_text` folder with a single space between each word and with a single file per document. Use the same filenames as in the `raw_text` folder.

[4] WordNet is a lexical database for English words.

- Examine a couple of the processed text files to verify that each step has been carried out successfully.

Running latent Dirichlet allocation

We will be using gensim, a Python package with fast implementations of NLP algorithms designed for processing large corpora, to run latent Dirichlet allocation on our processed text files.

First, we will prepare our documents for LDA by doing some further gensim-specific processing to represent each document as a vector of word frequencies.

- Install gensim by running `pip install gensim`.
- Create an empty list. Read in all of the processed Wikipedia pages, tokenize each one (*i.e.*, turn each one into a list of individual words), and store each tokenized page as an entry of previously created list.
- Import the `corpora` submodule of `gensim`. Use `corpora.Dictionary()` on the list of lists to create a *dictionary* of all the words which occur in the corpus.
- Use `Dictionary.save()` to save the dictionary to disk. (This means that if the output of `corpora.Dictionary()` object is saved in a variable `d`, you should call `d.save()`, since `d` is an object of class `Dictionary`.)

The `Dictionary` object also contains a mapping between words and word IDs, which is what we will use to form the vectors of word frequencies.

- Briefly examine this mapping by calling `print()` on the `Dictionary` object.
- Convert each tokenized document into a term–frequency representation with `Dictionary.doc2bow()`.
- Use `corpora.MmCorpus.serialize()` to save the processed corpus to disk.

Now, we are ready to run LDA on our corpus of documents!

- Import `LdaModel` from `gensim.models`. Call `LdaModel()` on the corpus with `num_topics=20`, arbitrarily chosen as a starting value to see how LDA works, to train an LDA model and store the results in an `LdaModel` object.

The terms associated with each of the 20 inferred topics can be examined with `LdaModel.get_topic_terms()`, which returns topic–term probabilities in terms of word IDs. Word IDs can be converted to words by calling `Dictionary.get()`.

- Write a function `get_topic(lda_model, dict, topic_num)` which uses an `LdaModel` object `lda_model` and a `Dictionary` object `dict` and returns the 20 most likely words for the `topic_num`th topic ordered from greatest to lowest probability. Verify that the output of ‘get_topic()

- Use `get_topic()` (or `LdaModel.show_topics()`) to examine the 20 topics extracted from the Wikipedia corpus. Interpret the results. How many of the topics are semantically meaningful? Do you think the number of topics extracted was too small or too large?

Picking the right number of topics to extract is generally difficult and constitutes an area of active research. Each model is associated with a *log-likelihood* value representing the probability of observing the training corpus under the calculated model, so the easiest way to choose the number of topics is to perform a grid search over the number of topics and choose the model associated with the highest (cross-validated) log-likelihood. Often, one will see references to the log-perplexity metric, which is equivalent to the average log-likelihood calculated on a per-word basis.[5]

gensim provides access to the log-perplexity via `LdaModel.log_perplexity()`.

- Perform a grid search over $k = 5, 10, 15, \ldots, 40$ for the number of topics extracted in LDA. Calculate the log-perplexity of each model on the entire corpus, increase the search range if necessary, and choose the model with the highest log-perplexity.

- Compare the semantic meaningfulness and interpretability of the topics in the optimal model to the 20 topics of your original LDA model.

- Examine the topics of LDA models with far too few or too many topics extracted. Interpret the results.

- Use `LdaModel.save()` to save the optimal LDA model to disk.

Visualizing an LDA model

We will conclude by using `pyLDAvis` with Jupyter to visualize our best LDA model.

- Install Anaconda, a Python distribution that comes with Jupyter, and follow the Anaconda package management documentation to install the `pyldavis` and `gensim` packages.

- Run the Jupyter Notebook and create a new Python 3 notebook.

[5] More complex methods exist. For instance, one can use the harmonic mean method as given by Ponweiser (2012), Latent Dirichlet Allocation in R, Section 4.3.3 to select the optimal k. In addition, Wallace *et al.* (2009) gives some even more complex (but better) methods for evaluating the quality of topic models.

We will proceed to follow the examples in the `pyLDAvis` demonstration notebook.

- Use Gensim's `Dictionary.load()`, `MmCorpus.load()`, and `LdaModel.load()` to load your previously saved dictionary, corpus, and LDA model objects.
- Call `pyLDAvis.gensim.prepare(lda_model, corpus, dict)` to visualize your LDA results in the Jupyter Notebook! Write down any interesting observations.

Closing notes

LDA is most useful for learning structure for corpora which are too large for humans to immediately fully understand. It has many extensions, such as *correlated topic models* which allow for greater correlations between topics or *dynamic topic models* which track the evolution of topics over time, with a huge amount of the work in this field being done by David Blei.

Some interesting articles on topic modeling to look at include:

- Mimno, Using phrases in Mallet topic models
- Mimno and McCallum (2007), Organizing the OCA: Learning Faceted Subjects from a Library of Digital Books
- Hu and Saul (2003), A Probabilistic Topic Model for Unsupervised Learning of Musical Key-Profiles
- Pritchard *et al.* (2000), Inference of Population Structure Using Multilocus Genotype Data, which was written before the development of LDA as it is now but proposes essentially the same generative model

Writing a spellchecker

We'll conclude with an enjoyable but less-important exercise in NLP, probabilistic modeling, and programmatic text manipulation.

Spelling correction is one of the most natural and oldest natural language processing tasks. It may seem like a difficult task to you at the moment, but it's surprisingly easy to write a spellchecker that does fairly well. (Of course, companies like Google spend millions of dollars making their spellcheckers better and better, but we'll start with something simpler for now.)

- Read Peter Norvig's How to Write a Spelling Corrector, paying particular attention to the probabilistic reasoning (which is similar to the ideas behind a naive Bayes classifier). Recreate it in R and reproduce his results. (The text file `big.txt` is available in the dataset folder `norvig-spellcheck-txt`.)

- **After** implementing your own spellchecker, read about this 2-line R implementation of Norvig's spellchecker.