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 scope modern NLP, we will proceed through a series of *case studies* which make use of a diverse range of concepts and techniques, including programming in both R and Python. In the course of doing so, we'll gain experience with the foundations of probabilistic language modeling, web scraping, naive Bayes classification, topic modeling, and more.

Email spam detection

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])

- Write a Python script to convert all text in the emails to lowercase and remove the following punctuation: periods, commas, colons, semicolons,

---

[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.

exclamation marks, question marks, apostrophes, and quotation marks. (Simple text pre-processing will improve the performance of our classifier; we'll cover pre-processing in greater depth in a later section.)

- 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

---

[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."

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* 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.)

- To save on computation time, select only the columns of the training and test data frames which correspond the 20,000 most common words in each one (where "commonness" is measured by the number of emails in which a word appears at least once).

The method of *naive* Bayes refers to the assumption of conditional independence among the classifier features. That is, if $P(w_i)$ denotes the probability of word w_i appearing in an email, we assume that $P(w_i \mid w_j) = P(w_i)$ for all $i \neq j$. In addition, we assume that there is no *a priori* reason for us to believe that an incoming message is more likely than not to be spam, so $P(\text{spam}) = P(\text{ham})$. These two assumptions allow us to obtain a very simple expression for the probability of an email containing the set of unique words $\{w_1, w_2, \ldots, w_n\}$ being spam:

$$P(\text{spam}) = p = \frac{p_1 p_2 \cdots p_n}{p_1 p_2 \cdots p_n + (1 - p_1)(1 - p_2) \cdots (1 - p_n)},$$

where $p_i = P(\text{spam} \mid w_i)$, the proportion of spam emails out of all the emails in which the word w_i appears.

- Calculate $P(\text{spam} \mid w_i)$ for each word in the training set. If a word appears only in spam emails, assign it a probability of $P(\text{spam} \mid w_i) = 0.999$ instead of 1; similarly, if a word appears only in ham emails, assign it a probability of 0.001 rather than 0. (The assignment of probabilities aside from 0 or 1 when a word only shows up in a single category is called Laplace or additive smoothing, where we say that not observing any instances of an event should cause us to assign it a very *low* probability, but not one exactly equal to zero.[4])

- Examine the words with the highest and lowest values of $P(\text{spam} \mid w_i)$. Interpret the results. Are they as you expected?

It is difficult for computers to precisely calculate the product of many small

---

[4]Regarding the rising of the sun, Laplace wrote: "But this number [the probability of the sun coming up tomorrow] is far greater for him who, seeing in the totality of phenomena the principle regulating the days and seasons, realizes that nothing at present moment can arrest the course of it." The statement in general is intended to illustrate the difficulty of using probability theory, specifically Laplace's rule of succession in the particular case of the sunrise problem, to infer parameters of probability distributions.

probabilities due to floating-point underflow.[5] As such, instead of performing a direct calculation of p, it is better to rewrite the above expression as

$$\frac{1}{p} - 1 = \frac{1 - p_1}{p_1} \frac{1 - p_2}{p_2} \cdots \frac{1 - p_n}{p_n}$$

so that the right hand side becomes a single product and to then take the logarithm of both sides to obtain

$$\log\left(\frac{1}{p} - 1\right) = \sum_{i=1}^{n} \left(\log(1 - p_i) - \log p_i\right).$$

$P(\text{spam}) = p$ can therefore be calculated from the value of the summation, which is much more amenable to floating-point calculations than our earlier product.

- Following the above description, calculate $P(\text{spam})$ for each email in the test set. (If a word occurs in the test set but not in the training set, give it $p_i = 0.4$.) Plot the associated ROC curve; calculate the AUC, false positive rate, and false negative rate.

- Examine the emails with the highest and lowest values of $P(\text{spam})$.

- **Extra:** Test the classifier on some spam and ham emails from your own personal inbox. Are they classified correctly?

Regularized logistic regression

Let's compare the performance of our naive Bayes classifier to logistic regression! The standard formulation of naive Bayes requires *binary* features, which is why we only considered the presence or absence of words, discarding information about *how many time* a word shows up in each email. However, the flexibility of logistic regression allows us to use word count information in our classification model.

- Install the `tm` package, which provides a variety of useful functions for NLP-related text manipulation and serves as the backbone of many other NLP packages.

---

[5]For a given data type in some programming language, there is typically some value $f_{\min}$ which is the magnitude of the smallest precisely representable floating-point number. As such, it becomes impossible to represent values in the interval $(-f_{\min}, f_{\min})$, known as the *underflow gap*. In the past, it was typical to simply set values in the underflow gap equal to 0, a practice known as *flushing to zero* on underflow. Newer versions of IEEE 754, the international standard for floating-point arithmetic, have introduced subnormal numbers, a set of linearly spaced numbers in the underflow gap which allow for imprecise representation of values falling within that gap. (In contrast, if you look at the non-exponent part of regular floating-point numbers written in scientific notation, they are *logarithmically* spaced.)

- Use `DirSource()` to get a list of `.eml` files (ensure that these files are pre-processed as specified above: all lowercase and with punctuation removed). Call `VCorpus()` on the output of `DirSource()` to create a `VCorpus` object with the text of the email files.

- Call `DocumentTermMatrix()` on the `VCorpus` object to create a matrix with word frequency data (representing emails as rows and words as columns) from the corpus of emails.

- To reduce the computation time required, remove columns corresponding to words which occur in fewer than 20 emails.

- Subset 80% of the rows of the document-term matrix to use as a training set, leaving the remaining 20% as a test set. Remove columns of *both* matrices corresponding to words which only show up in one but not both matrices.

- Train an L^1 regularized logistic classifier to distinguish between spam and ham. Examine the words associated with the greatest coefficients.

- Make predictions for the test set. Plot the associated ROC curve and calculate the AUC, false positive rate, and false negative rate. Compare with the performance of the naive Bayes classifier. Finally, plot two overlapping, semi-transparent histograms of the predicted log-odds on the test set for spam and ham emails; interpret this plot.

Sentiment analysis of Github comments

Linus Torvalds, the creator of the Linux kernel, is well-known for a very direct manner of communication.[6] For example, in 2012, he replied with the following email regarding 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?

---

[6]Linus is a Benevolent Dictator For Life of the Linux project, a term which refers to open-source project founders who maintain a position of final authority *in perpetuum*. The term actually originated with Guido van Rossum, creator of Python.

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 improper usage of `overflow_usub()`, GitHub pull requests, and a host of other topics.

How different is Linus compared to everyone else? As a brief case study in combining web scraping with NLP, we will use the GitHub API to scrape all the comments on a particularly notorious comment thread with Linus and then perform sentiment analysis on the messages to quantify these differences.

Using the Github API

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`. The output of GitHub's API calls is provided in the JSON format.

- Referencing the API documentation on issue comments, determine what API query will return the latest comments for issue #17 on the Linux repository.[7] (A parameter beginning with a colon (:) denotes a *variable* which you should fill in with the appropriate value *without* retaining the colon.) **Note:** If you encounter API rate-limiting problems when trying to

---

[7] Issue #17 is viewable online here (archive link here).

access the API, log in to GitHub or, *as a last resort*, use the pre-downloaded API requests in the `github-issue-17` folder.[8]

Additional pages of the API request can be accessed by appending `?page=2` and such to the URL.

- How many pages of API requests need to be downloaded to retrieve all comments on issue #17 posted on or before May 12th, 2012?

- Write a Python script using the `urllib.request` module to download the API calls which include all comments on issue #17 of the GitHub Linux repository on or before May 12, 2012. Save the downloaded API calls to disk.

- Open the downloaded files in a text editor and replace all instances of the *strings* `"\r\n"` or `"\n"` with spaces.

- Use the `json` module, specifically the `loads()` function, along with the `csv` module to create a CSV with columns for the comment poster's username and the comment text.

Performing sentiment analysis

Next, we'll perform some simple sentiment analysis in R on the GitHub comments and use *t*-tests to analyze the results.

- Load the CSV containing the GitHub comments. Ensure that the columns are loaded as character vectors rather than as factor vectors by setting `stringsAsFactors=FALSE`.

- Install and load the qdap package, which contains functions for both cleaning text and performing sentiment analysis.[9]

- Following the Cleaning Text & Debugging vignette, use qdap to clean the GitHub comments in preparation for sentiment analysis. In particular, `check_text()` should suggest some text-cleaning functions to use.

- Create a binary column indicating whether or not a comment was authored by Linus Torvalds.

- Use `polarity()` with its default settings to perform sentiment analysis on the GitHub comments, passing in the binary indicator column as the grouping variable.

---

[8]You can also call `curl -u "username" https://api.github.com/...` from the command line in Linux or OS X to download the API call result as a logged in user.

[9]On OS X, loading qdap may result in an error pertaining to the JVM. According-ing to StackOverflow, try running `sudo R CMD javareconf` in the Terminal followed by `install.packages('RJava', type='source')` in R.

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

- Examine the commit messages which had the lowest and highest polarity scores. What do high and low polarity represent?

- Plot two overlapping, semi-transparent histograms of the polarity scores for Linus's comments and everyone else's comments. Interpret the results.

- Use a two-sample *t*-test with `t.test()` to check if the difference in mean polarity between the two groups of comments is statistically significant.[10]

- **Extra:** User "shepik" commented on May 12th:

  > I love it that you can easily know just by
  > looking at word wraps of the comment
  > whether a person is supporting Linus or not

  Check to see if comments with manual linebreaks (those containing `"\r\n"` or `"\n"` in the raw JSON) have a lower polarity than those without.

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

---

[10]See Using t-tests in R for a brief explanation of *t*-tests.

8

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 $T("a") = 0.3$ and $T("b") = 0.7$.

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, scraping the actual content we want from each of those 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.

- 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 words with fewer than 4 characters (corresponding to punctuation like "?" and "!" and words like "but" and "he" which we'll want to remove later anyway).

- The `string` module contains a variable `string.punctuation` which is a string containing common punctuation characters. Remove all tokens which contain one or more punctuation characters.

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:[11]

```
>>> from nltk.stem import WordNetLemmatizer
>>> wnl = WordNetLemmatizer()
```

---

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

```
>>> 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.

- 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()`. (If your `Dictionary` object is called d and a tokenized document is called x, you can obtain the term–frequency representation with `d.doc2bow(x)`.)

- 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.show_topics(..., formatted=False)`, which returns a list of pairs of topic–term probabilities where words are specified by their IDs. Word IDs can be converted to words by calling `Dictionary.get()`. Finally, the number of topics in a `LdaModel` object m can be accessed via `m.num_topics`.

- Write a function `save_topics(lda_model, dict, fname)` which takes in a `LdaModel` object `lda_model`, a `Dictionary` model `dict`, and a filename `fname`. Your function should determine the top 20 words associated with each topic, form a string for each topic consisting of its top 20 terms joined together with a space in between each pair of words, and form a larger string with each topic-specific string on a different line. The final string should be saved to `fname`.

  E.g., if there are two topics, topic 1 has the words "apple", "orange", "banana", ... (in order of most to least probable), and topic 2 has the words "book", "school", "class", ..., then the following should be saved to `fname`:

  ```
  apple orange banana ...
  book school class ...
  ```

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.[12]

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

---

[12]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.

- 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.

- **Extra:** Latent semantic analysis (LSI) is a simpler topic modeling technique developed before LDA, analogous to principal component analysis in the same way that LDA is analogous to factor analysis. LSI requires much less computation time, but the underlying model is much simpler.[13] Run LSI in gensim (implemented in `gensim.models.lsimodel`) and compare the results to those from LDA.

Evaluating document similarity

LDA models each document as a mixture of different topics; as such, we can think of it as a dimensionality reduction technique that *represents* documents as linear combinations of topics. Looking at the vectors of topic proportions allows us to estimate the *similarity* between two documents. Indeed, if two documents have topic vectors **x** and **y**, we can look at the Euclidean distance $D(\mathbf{x}, \mathbf{y})$ between the two vectors, given by the relation

$$D(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n} (\mathbf{x}_i - \mathbf{y}_i)^2}.$$

This is the standard L^2 norm.

The topic distribution for each document can be accessed in gensim via `LdaModel.get_document_topics()`.

- Choose two Wikipedia pages which you think are topically very similar and two Wikipedia pages which you think are topically very dissimilar. Normalize the topic distribution vector for all four pages and compute the Euclidean distance between each of the two pairs' normalized distribution vectors. Are the results as you expect?

---

[13]Between the two is *probabilistic* LSI (pLSI), which is completely superseded by LDA; Radim Řehůřek, creator of gensim, writes: "With uniform topic priors, pLSI and LDA use essentially the same model, so with enough training data, you'll get the same results. But an added advantage of LDA (apart from more flexibility w.r.t. non-uniform priors) is that you can infer topics for previously unseen documents, which you can't with pLSI."

It is commonly said that the cosine similarity—the cosine of the angle between two vectors—rather than the Euclidean distance (as given above) is the proper distance metric to use for natural language processing. **This is typically incorrect.** When **x** and **y** are normalized to have a length of 1, the Euclidean distance is related to the cosine similarity $C(\mathbf{x}, \mathbf{y})$ by

$$D^2(\mathbf{x}, \mathbf{y}) = 2\left(1 - C(\mathbf{x}, \mathbf{y})\right).$$

The Euclidean distance and cosine similarity are related through a monotonic transformation, meaning that if $D(\mathbf{x}, \mathbf{y}) < D(\mathbf{a}, \mathbf{b})$ then $C(\mathbf{x}, \mathbf{y}) > C(\mathbf{a}, \mathbf{b})$ and vice versa.[14] In natural language, this means that if the Euclidean distance indicates that X and Y are closer together than documents A and B, then the cosine similarity X and Y is correspndingly smaller than the cosine similarity between A and B, with the converse also being true. As such, for purposes of comparing document similarity, the Euclidean distance and cosine similarity produce different results *only* if you work with unnormalized vectors![15]

- The cosine similarity can be computed as $C(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y}$ when **x** and **y** are normalized. Verify the above statements using the two pairs of documents you chose earlier.

Visualizing the 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.

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

- Use `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.

---

[14]This is because $C(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|}$. When the vectors are normalized to unit length (*i.e.*, $\|\mathbf{x}\| = \|\mathbf{y}\| = 1$), then $D^2(\mathbf{x}, \mathbf{y}) = (\mathbf{x} - \mathbf{y}) \cdot (\mathbf{x} - \mathbf{y}) = \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 - 2\mathbf{x} \cdot \mathbf{y} = 2 - 2C(\mathbf{x}, \mathbf{y})$. The transformation between the two is clearly monotonic.

[15]That being said, there are valid reasons to occasionally work with unnormalized or slightly unnormalized vectors. It just seems to be the case that the cosine similarity is over-recommended for NLP tasks without any good justification!

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

This section is optional.

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.