Latent Dirichlet Allocation (LDA) with Python

Jordan Barber

What is LDA?

Latent Dirichlet allocation (LDA) is a topic model (http://en.wikipedia.org/wiki/Topic_model) that generates topics based on word frequency from a set of documents. LDA is particularly useful for finding reasonably accurate mixtures of topics within a given document set.

LDA walkthrough

This walkthrough goes through the process of generating an LDA model with a highly simplified document set. This is not an exhaustive explanation of LDA. The goal of this walkthrough is to guide users through key steps in preparing their data and providing example output.

Packages required

This walkthrough uses the following Python packages:

- NLTK (http://www.nltk.org/install.html), a natural language toolkit for Python. A useful package for any natural language processing.
    - For Mac/Unix with `pip`: `$ sudo pip install -U nltk`.
- stop_words (https://pypi.python.org/pypi/stop-words), a Python package containing stop words.
    - For Mac/Unix with `pip`: `$ sudo pip install stop-words`.
- gensim (https://radimrehurek.com/gensim/install.html), a topic modeling package containing our LDA model.
    - For Mac/Unix with `pip`: `$ sudo pip install gensim`.

Importing your documents

Here is our sample documents:

```
doc_a = "Brocolli is good to eat. My brother likes to eat good brocolli, but not my moth
er."
doc_b = "My mother spends a lot of time driving my brother around to baseball practice."
doc_c = "Some health experts suggest that driving may cause increased tension and blood
 pressure."
doc_d = "I often feel pressure to perform well at school, but my mother never seems to d
rive my brother to do better."
doc_e = "Health professionals say that brocolli is good for your health."

# compile sample documents into a list
doc_set = [doc_a, doc_b, doc_c, doc_d, doc_e]
```

Cleaning your documents

Data cleaning is absolutely crucial for generating a useful topic model: as the saying goes, “garbage in, garbage out.” The steps below are common to most natural language processing methods:

- Tokenizing: converting a document to its atomic elements.
- Stopping: removing meaningless words.
- Stemming: merging words that are equivalent in meaning.

Tokenization

Tokenization segments a document into its atomic elements. In this case, we are interested in tokenizing to words. Tokenization can be performed many ways–we are using NLTK’s `tokenize.regexp` module:

```
from nltk.tokenize import RegexpTokenizer
tokenizer = RegexpTokenizer(r'\w+')
```

The above code will match any word characters until it reaches a non-word character, like a space. This is a simple solution, but can cause problems for words like “don’t” which will be read as two tokens, “don” and “t.” NLTK provides a number of pre-constructed tokenizers like `nltk.tokenize.simple` . For unique use cases, it’s better to use `regex` and iterate until your document is accurately tokenized.

Note: this example calls `tokenize()` on a single document. You’ll need to create a `for` loop to traverse all your documents. Check the script at the end of this page for an example.

```
raw = doc_a.lower()
tokens = tokenizer.tokenize(raw)

>>> print(tokens)
['brocolli', 'is', 'good', 'to', 'eat', 'my', 'brother', 'likes', 'to', 'eat', 'good',
'brocolli', 'but', 'not', 'my', 'mother']
```

Our document from `doc_a` is now a list of tokens.

Stop words

Certain parts of English speech, like conjunctions (“for”, “or”) or the word “the” are meaningless to a topic model. These terms are called stop words and need to be removed from our token list.

The definition of a stop word is flexible and the kind of documents may alter that definition. For example, if we’re topic modeling a collection of music reviews, then terms like “The Who” will have trouble being surfaced because “the” is a common stop word and is usually removed. You can always construct your own stop word list or seek out another package to fit your use case.

In our case, we are using the `stop_words` package from Pypi, a relatively conservative (https://github.com/Alir3z4/stop-words/blob/master/english.txt) list. We can call `get_stop_words()` to create a list of stop words:

```
from stop_words import get_stop_words

# create English stop words list
en_stop = get_stop_words('en')
```

Removing stop words is now a matter of looping through our tokens and comparing each word to the `en_stop` list.

```
# remove stop words from tokens
stopped_tokens = [i for i in tokens if not i in en_stop]

>>> print(stopped_tokens)
['brocolli', 'good', 'eat', 'brother', 'likes', 'eat', 'good', 'brocolli', 'mother']
```

Stemming

Stemming words is another common NLP technique to reduce topically similar words to their root. For example, “stemming,” “stemmer,” “stemmed,” all have similar meanings; stemming reduces those terms to “stem.” This is important for topic modeling, which would otherwise view those terms as separate entities and reduce their importance in the model.

Like stopping, stemming is flexible and some methods are more aggressive. The Porter stemming algorithm (http://tartarus.org/~martin/PorterStemmer/) is the most widely used method. To implement a Porter stemming algorithm, import the Porter Stemmer module from NLTK:

```
from nltk.stem.porter import PorterStemmer

# Create p_stemmer of class PorterStemmer
p_stemmer = PorterStemmer()
```

Note that `p_stemmer` requires all tokens to be type `str`. `p_stemmer` returns the string parameter in stemmed form, so we need to loop through our `stopped_tokens`:

```
# stem token
texts = [p_stemmer.stem(i) for i in stopped_tokens]

>>> print(stemmed_tokens)
['brocolli', 'good', 'eat', 'brother', 'like', 'eat', 'good', 'brocolli', 'mother']
```

In our example, not much happened: `likes` became `like`.

Constructing a document-term matrix

The result of our cleaning stage is `texts`, a tokenized, stopped and stemmed list of words from a single document. Let's fast forward and imagine that we looped through all our documents and appended each one to `texts`. So now `texts` is a list of lists, one list for each of our original documents.

To generate an LDA model, we need to understand how frequently each term occurs within each document. To do that, we need to construct a **document-term matrix** with a package called gensim:

```
from gensim import corpora, models

dictionary = corpora.Dictionary(texts)
```

The `Dictionary()` function traverses `texts`, assigning a unique integer id to each unique token while also collecting word counts and relevant statistics. To see each token's unique integer id, try `print(dictionary.token2id)`.

Next, our dictionary must be converted into a bag-of-words (http://en.wikipedia.org/wiki/Bag-of-words_model):

```
corpus = [dictionary.doc2bow(text) for text in texts]
```

The `doc2bow()` function converts `dictionary` into a bag-of-words. The result, `corpus`, is a list of vectors equal to the number of documents. In each document vector is a series of tuples. As an example, `print(corpus[0])` results in the following:

```
>>> print(corpus[0])
[(0, 2), (1, 1), (2, 2), (3, 2), (4, 1), (5, 1)]
```

This list of tuples represents our first document, `doc_a`. The tuples are `(term ID, term frequency)` pairs, so if `print(dictionary.token2id)` says `brocolli`'s id is `0`, then the first tuple indicates that `brocolli` appeared twice in `doc_a`. `doc2bow()` only includes terms that actually occur: terms that do not occur in a document will not appear in that document's vector.

Applying the LDA model

`corpus` is a document-term matrix and now we're ready to generate an LDA model:

```
ldamodel = gensim.models.ldamodel.LdaModel(corpus, num_topics=3, id2word = dictionary, p
asses=20)
```

The `LdaModel` class is described in detail in the gensim documentation (https://radimrehurek.com/gensim/models/ldamodel.html). Parameters used in our example:

Parameters:

- `num_topics` : *required*. An LDA model requires the user to determine how many topics should be generated. Our document set is small, so we're only asking for three topics.
- `id2word` : *required*. The `LdaModel` class requires our previous `dictionary` to map ids to strings.

- `passes` : *optional*. The number of laps the model will take through `corpus` . The greater the number of passes, the more accurate the model will be. A lot of passes can be slow on a very large corpus.

Examining the results

Our LDA model is now stored as `ldamodel` . We can review our topics with the `print_topic` and `print_topics` methods:

```
>>> print(ldamodel.print_topics(num_topics=3, num_words=3))
['0.141*health + 0.080*brocolli + 0.080*good', '0.060*eat + 0.060*drive +
0.060*brother', '0.059*pressur + 0.059*mother + 0.059*brother']
```

What does this mean? Each generated topic is separated by a comma. Within each topic are the three most probable words to appear in that topic. Even though our document set is small the model is reasonable. Some things to think about: - `health` , `brocolli` and `good` make sense together. - The second topic is confusing. If we revisit the original documents, we see that `drive` has multiple meanings: driving a car and driving oneself to improve. This is something to note in our results. - The third topic includes `mother` and `brother` , which is reasonable.

Adjusting the model's number of topics and passes is important to getting a good result. Two topics seems like a better fit for our documents:

```
ldamodel = gensim.models.ldamodel.LdaModel(corpus, num_topics=2, id2word = dictionary, p
asses=20)

>>> print(ldamodel.print_topics(num_topics=2, num_words=4))
['0.054*pressur + 0.054*drive + 0.054*brother + 0.054*mother', '0.070*brocolli + 0.070*g
ood + 0.070*health + 0.050*eat']
```

So what does LDA *actually* do?

This explanation is a little lengthy, but useful for understanding the model we worked so hard to generate.

LDA assumes documents are produced from a mixture of topics. Those topics then generate words based on their probability distribution, like the ones in our walkthrough model. In other words, LDA assumes a document is made from the following steps:

1. Determine the number of words in a document. Let's say our document has 6 words.
2. Determine the mixture of topics in that document. For example, the document might contain 1/2 the topic "health" and 1/2 the topic "vegetables."
3. Using each topic's multinomial distribution, output words to fill the document's word slots. In our example, the "health" topic is 1/2 our document, or 3 words. The "health" topic might have the word "diet" at 20% probability or "exercise" at 15%, so it will fill the document word slots based on those probabilities.

Given this assumption of how documents are created, LDA backtracks and tries to figure out what topics would create those documents in the first place.

Sample script in full

```
from nltk.tokenize import RegexpTokenizer
from stop_words import get_stop_words
from nltk.stem.porter import PorterStemmer
from gensim import corpora, models
import gensim

tokenizer = RegexpTokenizer(r'\w+')

# create English stop words list
en_stop = get_stop_words('en')

# Create p_stemmer of class PorterStemmer
p_stemmer = PorterStemmer()

# create sample documents
doc_a = "Brocolli is good to eat. My brother likes to eat good brocolli, but not my moth
er."
doc_b = "My mother spends a lot of time driving my brother around to baseball practice."
doc_c = "Some health experts suggest that driving may cause increased tension and blood
 pressure."
doc_d = "I often feel pressure to perform well at school, but my mother never seems to d
rive my brother to do better."
doc_e = "Health professionals say that brocolli is good for your health."

# compile sample documents into a list
doc_set = [doc_a, doc_b, doc_c, doc_d, doc_e]

# list for tokenized documents in loop
texts = []

# loop through document list
for i in doc_set:

    # clean and tokenize document string
    raw = i.lower()
    tokens = tokenizer.tokenize(raw)

    # remove stop words from tokens
    stopped_tokens = [i for i in tokens if not i in en_stop]

    # stem tokens
    stemmed_tokens = [p_stemmer.stem(i) for i in stopped_tokens]

    # add tokens to list
    texts.append(stemmed_tokens)

# turn our tokenized documents into a id <-> term dictionary
dictionary = corpora.Dictionary(texts)

# convert tokenized documents into a document-term matrix
corpus = [dictionary.doc2bow(text) for text in texts]

# generate LDA model
```

```
ldamodel = gensim.models.ldamodel.LdaModel(corpus, num_topics=2, id2word = dictionary, p
asses=20)
```