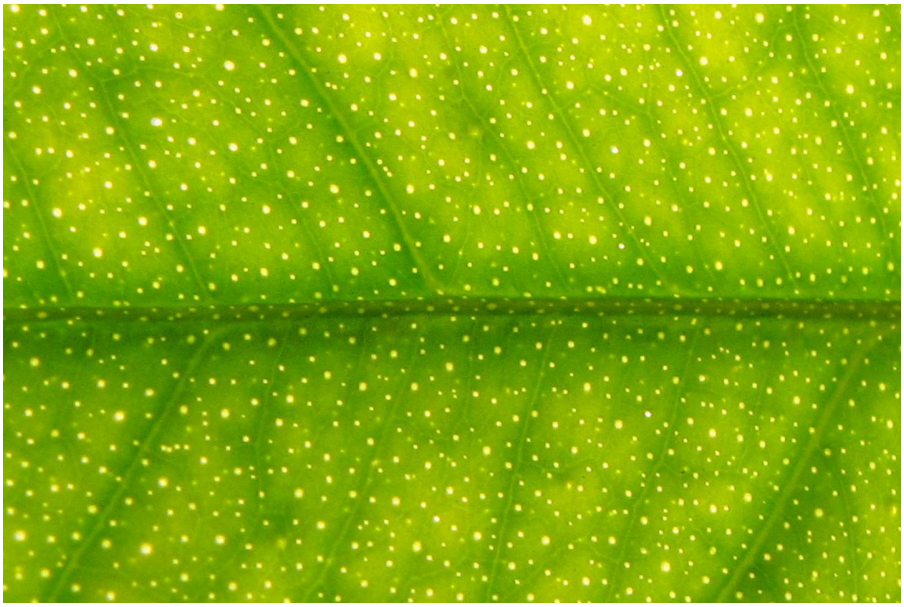
Topic modeling for the newbie

Learning the fundamentals of natural language processing.

By Marie Beaugureau, May 12, 2015



(source: User: ZooFari / Wikimedia Commons / CC-BY-SA-3.0).

Editor's note: This is ***an excerpt from our recent book*** *Data Science from Scratch, by Joel Grus. It provides a survey of topics from statistics and probability to databases, from machine learning to MapReduce, giving the reader a foundation for understanding, and examples and ideas for learning more.*

When we built our Data Scientists You Should Know recommender in Chapter 1, we simply looked for exact matches in people's stated interests.

A more sophisticated approach to understanding our users' interests might try to identify the topics that underlie those interests. A technique called *Latent Dirichlet Analysis* (LDA)

is commonly used to identify common topics in a set of documents. We'll apply it to documents that consist of each user's interests.

LDA has some similarities to the Naive Bayes Classifier we built in Chapter 13, in that it assumes a probabilistic model for documents. We'll gloss over the hairier mathematical details, but for our purposes the model assumes that:

- There is some fixed number *K* of topics.
- There is a random variable that assigns each topic an associated probability distribution over words. You should think of this distribution as the probability of seeing word *w* given topic *k*.
- There is another random variable that assigns each document a probability distribution over topics. You should think of this distribution as the mixture of topics in document *d*.
- Each word in a document was generated by first randomly picking a topic (from the document's distribution of topics) and then randomly picking a word (from the topic's distribution of words).

In particular, we have a collection of `documents`, each of which is a `list` of words. And we have a corresponding collection of `document_topics` that assigns a topic (here a number between 0 and *K* – 1) to each word in each document.

So that the fifth word in the fourth document is:

```
documents[3][4]
```

and the topic from which that word was chosen is:

```
document_topics[3][4]
```

This very explicitly defines each document's distribution over topics, and it implicitly defines each topic's distribution over words.

We can estimate the likelihood that topic 1 produces a certain word by comparing how many times topic 1 produces that word with how many times topic 1 produces *any* word. (Similarly, when we built a spam filter in Chapter 13, we compared how many times each word appeared in spams with the total number of words appearing in spams.)

Although these topics are just numbers, we can give them descriptive names by looking at the words on which they put the heaviest weight. We just have to somehow generate the `document_topics`. This is where Gibbs sampling comes into play.

We start by assigning every word in every document a topic completely at random. Now we go through each document one word at a time. For that word and document, we construct weights for each topic that depend on the (current) distribution of topics in that document and the (current) distribution of words for that topic. We then use those weights to sample a new topic for that word. If we iterate this process many times, we will end up with a joint sample from the topic-word distribution and the document-topic distribution.

To start with, we'll need a function to randomly choose an index based on an arbitrary set of weights:

```
def sample_from(weights):
    """returns i with probability weights[i] / sum(weights)"""
    total = sum(weights)
    rnd = total * random.random() # uniform between 0 and total
    for i, w in enumerate(weights):
        rnd -= w # return the smallest i such that
        if rnd <= 0: return i # weights[0] + ... + weights[i] >= rnd
```

For instance, if you give it weights [1, 1, 3], then one-fifth of the time it will return 0, one-fifth of the time it will return 1, and three-fifths of the time it will return 2.

Our documents are our users' interests, which look like:

```
documents = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
```

```
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

And we'll try to find `K = 4` topics.

In order to calculate the sampling weights, we'll need to keep track of several counts. Let's first create the data structures for them.

How many times each topic is assigned to each document:

```
# a list of Counters, one for each document
document_topic_counts = [Counter() for _ in documents]
```

How many times each word is assigned to each topic:

```
# a list of Counters, one for each topic
topic_word_counts = [Counter() for _ in range(K)]
```

The total number of words assigned to each topic:

```
# a list of numbers, one for each topic
```

```
topic_counts = [0 for _ in range(K)]
```

The total number of words contained in each document:


```
# a list of numbers, one for each document
document_lengths = map(len, documents)
```

The number of distinct words:

```
distinct_words = set(word for document in documents for word in document)
W = len(distinct_words)
```

And the number of documents:

```
D = len(documents)
```

For example, once we populate these, we can find the number of words in `documents[3]` associated with topic 1 as:

```
document_topic_counts[3][1]
```

And we can find the number of times *nlp* is associated with topic 2 as:

```
topic_word_counts[2]["nlp"]
```

Now we're ready to define our conditional probability functions. As in Chapter 13, each has a smoothing term that ensures every topic has a nonzero chance of being chosen in any document and that every word has a nonzero chance of being chosen for any topic:

```
def p_topic_given_document(topic, d, alpha=0.1):
    """the fraction of words in document _d_
    that are assigned to _topic_ (plus some smoothing)"""
    return ((document_topic_counts[d][topic] + alpha) /
            (document_lengths[d] + K * alpha))

def p_word_given_topic(word, topic, beta=0.1):
    """the fraction of words assigned to _topic_
    that equal _word_ (plus some smoothing)"""
    return ((topic_word_counts[topic][word] + beta) /
            (topic_counts[topic] + W * beta))
```

We'll use these to create the weights for updating topics:

```
def topic_weight(d, word, k):
    """given a document and a word in that document,
    return the weight for the kth topic"""
    return p_word_given_topic(word, k) * p_topic_given_document(k, d)

def choose_new_topic(d, word):
    return sample_from([topic_weight(d, word, k)
                        for k in range(K)])
```

There are solid mathematical reasons why `topic_weight` is defined the way it is, but their details would lead us too far afield. Hopefully it makes at least intuitive sense that—given a word and its document—the likelihood of any topic choice depends on both how likely that topic is for the document and how likely that word is for the topic.

This is all the machinery we need. We start by assigning every word to a random topic, and populating our counters appropriately:

```
random.seed(0)
document_topics = [[random.randrange(K) for word in document]
                   for document in documents]

for d in range(D):
    for word, topic in zip(documents[d], document_topics[d]):
        document_topic_counts[d][topic] += 1
        topic_word_counts[topic][word] += 1
        topic_counts[topic] += 1
```

Our goal is to get a joint sample of the topics-words distribution and the documents-topics distribution. We do this using a form of Gibbs sampling that uses the conditional probabilities defined previously:

```
for iter in range(1000):
    for d in range(D):
        for i, (word, topic) in enumerate(zip(documents[d],
                                              document_topics[d])):

            # remove this word / topic from the counts
            # so that it doesn't influence the weights
            document_topic_counts[d][topic] -= 1
            topic_word_counts[topic][word] -= 1
            topic_counts[topic] -= 1
            document_lengths[d] -= 1

            # choose a new topic based on the weights
            new_topic = choose_new_topic(d, word)
            document_topics[d][i] = new_topic

            # and now add it back to the counts
            document_topic_counts[d][new_topic] += 1
            topic_word_counts[new_topic][word] += 1
            topic_counts[new_topic] += 1
            document_lengths[d] += 1
```

What are the topics? They're just numbers 0, 1, 2, and 3. If we want names for them we have to do that ourselves. Let's look at the five most heavily weighted words for each (Table 20-1):

```
for k, word_counts in enumerate(topic_word_counts):
    for word, count in word_counts.most_common():
        if count > 0: print k, word, count
```

Table 20-1: Most common words per topic

Topic 0	Topic 1	Topic 2	Topic 3
Java	R	HBase	regression
Big Data	statistics	Postgres	libsvm
Hadoop	Python	MongoDB	scikit-learn
deep learning	probability	Cassandra	machine learning
artificial intelligence	pandas	NoSQL	neural networks

Based on these I'd probably assign topic names:

```
topic_names = ["Big Data and programming languages",
               "Python and statistics",
               "databases",
               "machine learning"]
```

at which point we can see how the model assigns topics to each user's interests:

```
for document, topic_counts in zip(documents, document_topic_counts):
    print document
    for topic, count in topic_counts.most_common():
        if count > 0:
            print topic_names[topic], count,
    print
```

which gives:

```
['Hadoop', 'Big Data', 'HBase', 'Java', 'Spark', 'Storm', 'Cassandra']
Big Data and programming languages 4 databases 3
['NoSQL', 'MongoDB', 'Cassandra', 'HBase', 'Postgres']
databases 5
['Python', 'scikit-learn', 'scipy', 'numpy', 'statsmodels', 'pandas']
Python and statistics 5 machine learning 1
```

and so on. Given the “ands” we needed in some of our topic names, it's possible we should use more topics, although most likely we don't have enough data to successfully learn them.

Article image: (source: User: ZooFari / Wikimedia Commons / CC-BY-SA-3.0).

Marie Beaugureau

Marie Beaugureau is the lead data editor for O'Reilly Media.

DATA

Oil, Gas, and Data

By Daniel Cowles

High-performance data tools in the production of industrial power

DATA

Designing great data products

The Drivetrain Approach: A four-step process for building data products.

The next 10 years of Apache Hadoop

By Doug Cutting, Tom White and Ben Lorica

Doug Cutting, Tom White, and Ben Lorica explore Hadoop's role over the coming decade.

DATA