Vignette: Information Retrieval and Bags of Words

This is a vignette consisting of several exercises. Each exercise is a separate homework submission, and is marked with the number of assignment points it is worth. To start each exercise, use the `python new-homework.py [language] information-retrieval-bow` *command, and submit each exercise as a separate pull request. You can do as many or as few exercises from a vignette as you wish.*

Background

An important statistical challenge that has emerged since the rise of the Internet has been how to deal with text-based data. One form that textual data often comes in is as a (usually large) collection of documents, and a common problem is to find documents within the collection that meet specified criteria. This is area is usually called, somewhat blandly, *information retrieval* (IR).

Documents

For the purposes of IR, we will define a document as a *hierarchical*, *partially ordered* collection of data elements.

- Hierarchical: decomposed into parts in a tree-like arrangment (e.g., Chapters, Sections, Subsections, Paragraphs, Sentences, Words).
- Partially-Ordered: within some parts, data elements are listed in a definite order (e.g., words in sentence, letters in a word).

The data elements can include text, images, embedded objects (e.g., video, audio, music, spreadsheets), embedded *metadata* (data about the document but not part of it), control structures. Text is typically the primary constituent, but the idea is quite general

Documents can have many different *formats* (e.g., plain text, HTML, PDF), and data elements can have many different *encodings* (e.g., ASCII, unicode). They can be unstructured like a plain text file or structured like an XML document.

Usually, IR methods operate on a fixed collection of documents called a *corpus*. In this case, the documents can be considered to be the data points for IR. These data are thus usually high-dimensional, non-quantitative, and complex, raising several statistical challenges. The overarching strategy here is to find a *representation* of the data that is (i) easy to compute from the raw documents, (ii) makes search of the corpus efficient and flexible, and (iii) highlights the features of the documents that we care about (while ideally suppressing features we do not care about).

Similarity Search

For this vignette, we will consider one particular kind of IR search, called similarity search: given a document, find other documents in the corpus that are similar to it. The notion of similarity (or sometimes dis-similarity) that we use must be specified and quantified.

Similarity search proceeds in roughly three steps:

1. We select a representation for our documents, as described above.
2. We define a measure of similarity or dis-similarity (whichever is more convenient).
3. We find all documents in the corpus with at least a specified similarity (or within a specified dis-similarity).

The similarity/dis-similarity measures are usually computed in terms of the representations rather than the raw documents. One way to construct these is to use a distance metric on the representations. This gives a measure of dis-similarity, and we want to find all documents with this measure smaller than some chosen threshold. See below for specifics.

The Bag of Words

Here, we will use a basic representation called the *bag of words* (BoW). Call the collection of all words in the corpus the *lexicon*. For a given document, we associate with each word in the lexicon the number of times that word appears in the document. We can store each bag of words as an associative array, but in practice, it is usually more convenient to use a vector whose dimension is the size of our lexicon. We arrange the lexicon in a fixed but arbitrary order (e.g., alphabetical), and we store the counts in a vector arranged in the same order. For example, suppose that our lexicon contains only twelve words and we have two documents: "She said jump and her sister said how high" and "She observed orange flux, her sister said." The BoW vectors for these documents would be:

```
and flux her high how jump observed orange said she sister
  1    0   1    1   1    1        0      0    2   1      1
  0    1   1    0   0    0        1      1    1   1      1
```

With the bag of words, we represent documents by numeric vectors, so we can measure dis-similarity with any distance metric on such vectors. For example, given BoWs x and y, we might measure dis-similarity between the documents by one of the following:

- Euclidean Distance: $d_2(x, y) \equiv \|x - y\| = \sqrt{\sum_i (x_i - y_i)^2}$.
- Manhattan Distance: $d_1(x, y) \equiv \sqrt{\sum_i |x_i - y_i|}$.
- Maximum Distance: $d_\infty(x, y) \equiv \max_i |x_i - y_i|$.
- Cosine Dissimilarity: $d_{\cos}(x, y) = \dfrac{x \cdot y}{\sqrt{(x \cdot x)(y \cdot y)}} = \dfrac{\sum_i x_i y_i}{\|x\| \|y\|}$

A Few Wrinkles

Most real documents are more complicated than just lists of words. From punctuation marks in regular text to the control structures in a format like XML, document analysis raises a few practical complications.

Isolating Words. To create the BoW vector, we need to find the words, but in real documents, the words are enmeshed in punctuation, structure, and embedded objects (e.g., pictures). In simple cases, we can simply strip out all non-letters, replacing them with spaces and then compressing multiple spaces to single spaces. But this does not solve the problem in general. Consider, for instance, the challenge of dealing with hyphens. A hyphenated word, like non-sequitur, should be kept as is or possibly be split, but a "word" like ter-rific which was hyphenated "artificially" by a line break should have the hyphen deleted only. Similarly, literal numbers might be considered as is, but might be better converted to some common symbol to represent number-ness. Choices, choices. In these exercises, we will take the simpler approach.

Equivalence Classes. What counts as separate words? Are "She" and "she" the same? Do we consider accents and diacritical marks as part of the word or as an adornment? We can convert words in the document into representatives of *equivalence classes*; for instance, we could make equivalence classes by case, mapping "She", "she", and "SHE" to "she."

Another kind of equivalence comes from mapping variant forms of words into a root form (or lemma), a process called *lemmatization*. For example, a form like "arguing" would map to "argue," "provision" to "provide." But this requires that we use vocabulary and morphological analysis to derive the lemma. For example, "sawing" would map to "saw," but"saw" would give "see" or "saw" depending on whether how the word was used.

Normalization. What do we do with documents of unequal length? In that case, the BoWs vectors can be dissimilar because of the document size rather than the distribution of words. It can help to *normalize* the BoW vectors. Two common choices of normalization are

- Document-Length Normalization: $x \to \dfrac{x}{\sum_i x_i}$
- Euclidean-Length Normalization: $x \to \dfrac{x}{\|x\|}$

In practice, Euclidean-length normalization tends to "work" better, apparently because document-length normalization de-emphasizes rare words, which are useful markers in searches.

Weighting. When using the bag of words, the word counts treat all words as equally important for determining relevance to a search. But in practice, all words are not created equal. For example, common words like articles "the" or "a" are not particularly useful for distinguishing documents, but rare words like "flux" are. Weighting is designed to make less common words have a bigger impact on the comparison, hopefully improving the quality of the search.

One common approach to weighting is *Inverse Document Frequency* weighting, or IDF. To calculate this, we consider various counts (usually called "frequencies" in this context). For a word w:

- Word frequency $n_{d,w}$ – the raw count in the bag of words; how many times word w appears in document d
- Collection frequency c_w – number of times word w appears in the collection.
- Document frequency d_w – number of documents containing word w.

The IDF weight is then computed by

$$\text{idf}_w = -\log \frac{d_w}{N},$$

and the IDF-adjusted word frequency (entry in the adjust BoW vector) is given by

$$\text{mb}_{d,w} = n_{d,w} \times \text{idf}_w.$$

This is highest when w occurs many times in a small number of documents; lower when w occurs fewer times in a document or occurs in many documents; and lowest when w occurs in essentially every document.

Task

In the five exercises below, we will use the bag of words representation to search a selection of documents from the New York Times Annotated Corpus. The part of the collection you will analyze is divided into documents that have been tagged (by humans) as being "about" either art or music.

In the `Data/information-retrieval-bow` directory, you will find subdirectories `nyt-collection` and `nyt-collection-text`. The former contains the original documents, which are in XML format; the latter have been processed in simple ways to produce lower case text with only words and spaces, with all numbers converted to a literal #. The simple script `textify.py` in the `Data/information-retrieval-bow` directory describes how this conversion was performed, if you want to know the details. You can use either form of the corpus; processing the XML files is an added challenge, with some tips given below.

Several of the exercises ask you to write a function that performs some specific operation on its input. If functions are not yet familiar to you, then you can write each of these exercises as a stand-alone program in your chosen language.

Questions

Here are a few optional questions to help you think about the above material. You can give your answers in comments on your GitHub pull request for any of the exercises you submit from this vignette.

1. Give a BoW vector for the document "To be or not to be, that is the question."
2. Suppose do a search for the word "be." What is the BoW representation of this query when considered as a document? What is the distance (Euclidean, say) between the query document and the document in the previous question?

3. Describe a simple text search that cannot be done using the BoW representation, no matter which similarity measure is used.

4. Explain, in your own words, the logic behind the IDF weighting.

Task

Exercise #1. Tokenizing Documents. (1 point)

Write a function *tokenize* that takes the name of a document as an argument and returns a list/vector/array of all the words in the document, in the order in which they appear.

For example, if the document `foo` contains "Now is the time to head west toward wilder lands," then `tokenize("foo")` should return the list

Now, is, the, time, to, head, west, toward, wilder, lands

in whatever data structure is appropriate to your language.

If you are using the text versions of the documents, you can use ordinary string-processing commands. If you are using the XML versions, you will need to extract the text. Each document contains a single `<block>` tag with class `full_text`, and the contents of all the `<p>` tags within that block give the text contents of the document. In R, you can use the package "xml2." In Python, you can use the built-in package `xml.etree` or the `beautifulsoup` library. If you need help doing this in other languages and cannot find it in the documentation, let us know. In all cases, the XPath expression for the block is `".//block[@class='full_text']"`, and you want find all `p` tags within.

Next, write a function *makeBoW* that takes two arguments: a list/vector/array as produced by *tokenize* in the last and an ordered list/array/vector of words representing the lexicon. The function should return a bag of words vector for the corresponding document, ordered according to the lexicon vector given as input. (In R, it would be nice to set the `names` attribute for the vector to show the word associated with each count.)

Exercise #2. Analyzing the Corpus. (2 points)

Write a function *analyzeCorpus* to create bags of words for all documents in a corpus. This function should take a single argument, which is a collection of file names for the documents. It should use the *makeBow* function from the previous exercise and apply it to each of the documents. The function *analyzeCorpus* should return a collection of bag-of-words vectors for the documents in the same order.

Pick a data structure that is convenient to store the resulting collection. In languages like R, in which matrices are easy to construct, it will probably be most natural to return a matrix whose rows are the bag-of-words vectors. (In R, it would be nice to set the `dimnames` attribute of this matrix to label the columns with the words of the lexicon and label the rows with the root name of the document, e.g., `foo/bar/mydoc.txt` would have its row labeled `mydoc`.) For other languages, you can also return a list of BoW vectors, or an associative array mapping documents to BoW vectors, or whatever is most convenient for downstream processing.

Apply your function to the New York Times corpus and save the resulting collection of BoW vectors. It will be useful for what follows to arrange the document names so that the 57 art documents come first and the 45 music documents last.

Exercise #3. Checking Distances. (2 points)

Select two distance measures from the list above and write a function for each that computes the distance between BoW vectors given the two vectors as input. (The vectors are assumed to be on the same lexicon in the same order.)

Now use your distance functions to compute a matrix of pairwise distances between documents in the corpus. Save each of these matrices. If you arranged the documents in order as suggested in the previous exercise, you can see from this matrix whether the art and music documents are more similar to each other than to documents in the other category.

Construct a visualization of these distances for each measure, and put it in PDF or PNG files named `nyt-clustering1` and `nyt-clustering2`, respectively. For instance, you can plot the distance matrix as an image and compare the coloring of the diagonal versus off diagonal blocks. Or you can use classic multi-dimensional scaling. (This is available in R via the function *`cmdscale`*. Python has a similar function in the `scikit-learn` library. Other languages have similar libraries available.) What do these visualizations suggest about our ability to distinguish different kinds of documents from the BoW representation? Did you detect any notable differences among the two distance measures that you used? (Answer these questions in a comment on your pull request.)

Exercise #4. Document Search. (1 point)

Write a function that *`IrSearch`* that takes a BoW vector for a query document, the collection of BoW vectors for a corpus, and a positive integer *`k`* (defaulting to 1 if possible). The function return the *`k`* documents in the corpus that are closest to the query document.

Select a few random documents in the New York Times corpus and apply your *`IrSearch`* function to each, either excluding the query document from the corpus or ignoring it in the results. In a comment on your pull request, describe what you learned from these searches.