Lecture 2 — Information Retrieval and Similarity Search

36-462/662

28 August 2019

Contents

Good approaches to information retrieval we will neglect **1**
Metadata . . . 1
Boolean queries . . . 1

Similarity search **1**

Representation **3**
Textual features . . . 4

Measuring Similarity **7**
Normalization . . . 8

Practice the Criterion of Truth **9**

Abstraction **9**

Exercises **10**

References **10**

One of the fundamental problems with having a lot of data is finding what you're looking for. This is called **information retrieval**.

Good approaches to information retrieval we will neglect

Metadata

The oldest approach is to have people create data about the data, **metadata**, to make it easier to find relevant items. Library catalogues are like this (Figure 1): people devise detailed category schemes for books, magazines, etc. For instance, a book has a title, one or more authors (possibly "anonymous" or pseudonyms), a publisher, a place of publication, a date of publication, possibly an ISBN, and its contents belong to one or more subject topics, with a lot of work going in to designing the set of subject-matter topics. A magazine doesn't have an author, it has multiple volumes with multiple dates, and it has an ISSN instead of an ISBN, if it has a number like that at all. (Nowadays we'd call this sort of scheme an **ontology**.) Having fixed on a scheme like this, people would actually examine the objects, decide which categories they belong to, and write down that information along with their location on the shelves, and then copy this information so that it appeared in multiple places — by title, by author, by subject, etc. This worked OK for a few thousand years, but it needs people, who are slow and expensive and don't scale. It's not feasible for searching census records, or purchase histories at an online store, or the Web.

Boolean queries

The next oldest approach is Boolean queries: things like "all census records of Presbyterian or Methodist plumbers in Rhode Island with at least two but no more than five children". The first electro-mechanical

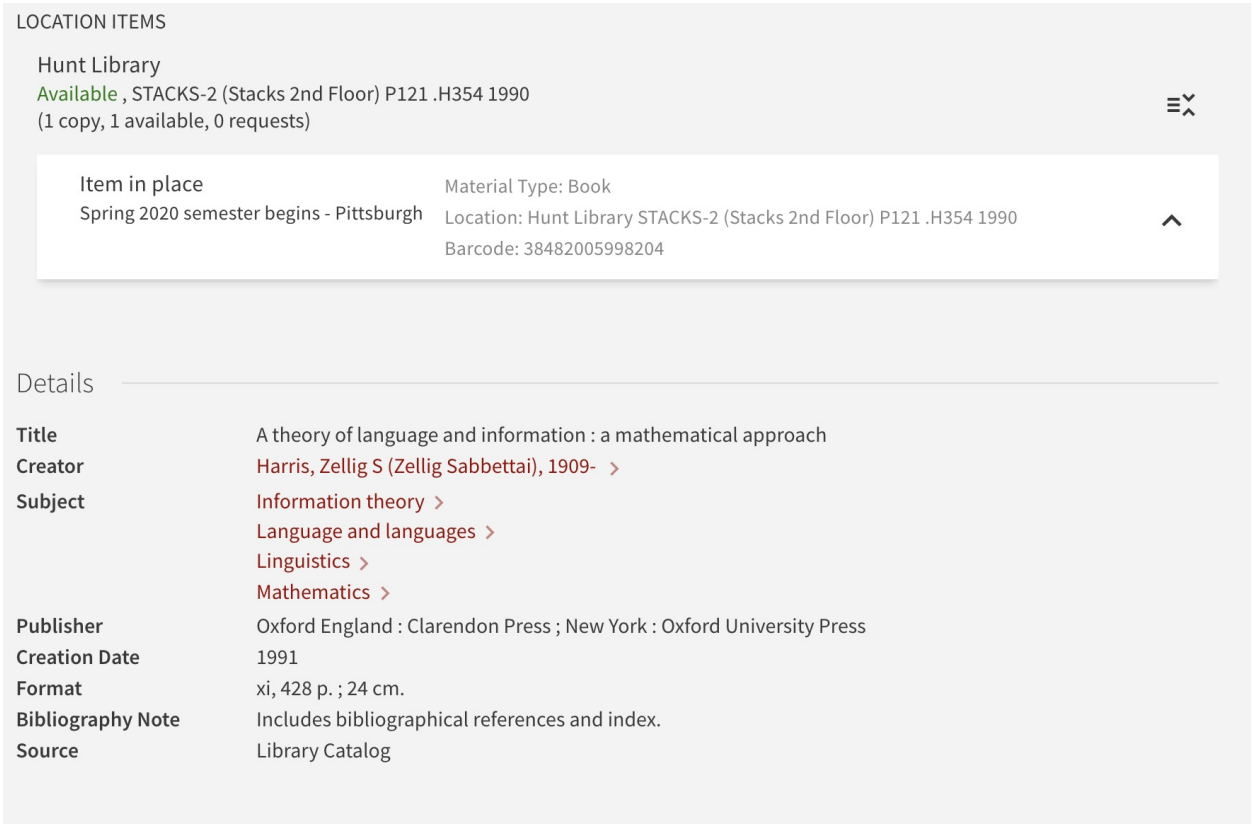

Figure 1: Example of **metadata** (from [search.library.cmu.edu]). This is information *about* the book, which is not part of its actual contents. Notice that this is *structured*: every book in the catalogue will have a title, author(s), subject(s), a location, etc. All of this information is humanly-generated.

data-processing machines were invented in the late 1800s years ago to do searches like this. The advantages are that it's very easy to program, it doesn't need a lot of human intervention, and sometimes it's exactly what you want to do, say if you're taking a census. But there are disadvantages: most people don't think in Boolean queries; it works best when it's dealing with **structured** data, like census records, not **unstructured** data like most documents on the Web; and it's got no sense of *priority*. Suppose you're loking for a used car, thinking of buying a 2009-model-year Mustang, and want to know what problems they're prone to. Imagine doing a Boolean search of the whole Web for "Mustang AND 2009 AND problems". *Some* of those documents will be just what you want, but you'll also get a lot about horses, about places named "Mustang", about sports teams named "Mustangs", and so on.

Similarity search

This is where **searching by similarity** comes in. Suppose you can find *one* Web page which is about problems in 2009-model Mustangs. We're going to see today how you can tell your computer "find me more pages like this". We will see later how you can avoid the step of initially lucking into the first page, and how you can use the same sort of trick to search other kinds of data — say images on the Web, or hospital patient records, or retail transactions, or telephone-call records.

To illustrate these ideas concretely, we're going to use a part of the *New York Times Annotated Corpus* (Sandhaus 2008). This consists of 1.8×10^6 stories from the *Times*, from 1987 to 2007, which have been hand-annotated with metadata about their contents. (See Figure 2.) We are going to look at methods for information retrieval which do *not* use the metadata, but having it there will help us determine how well those methods are working. You will get to use a small part of this data set in the homework[1].

Representation

The crucial first step is to chose a **representation** of the data. There are multiple considerations here.

- The representation ought to be something our methods can work with easily.
- The representation ought to be something we can easily generate from the raw data.
- The representation ought to highlight the important, helpful aspects of the data, and suppress others. (If the representation *didn't* ignore some aspects of the data, it would be the data again.)

One of the things we'll see as we go along in the course is that getting a good representation — trading these concerns off against each other — is at least as important as picking the right algorithms.

For today, we are searching for documents by similarity, so our data are natural-language documents.

We could try to represent the *meaning* of the documents. Look at Figure **??** again. The abstract reads "Chairs of the 1920s and 30s are featured at Johnson & Hicks, new home furnishing store in TriBeCa". We could try to represent the meaning here in something like mathematical-logical[2] notation:

```
exhibit.of(chairs(age-1920--1930),Johnson&Hicks,now)
is.a(Johnson&Hicks,store,type="home furnishing")
location.of(Johnson&Hicks,TriBeCa)
begin.date(Johnson&Hicks,now)
```

and then we'd probably want to go on to tell the system that chairs are a kind of furniture, where TriBeCa is, that chairs that old are a kind of vintage product, etc. If we could do this, there are tons of methods for automated deduction which we could use to find stories about displays of contemporary chairs, or vintage lamps, etc. The snag is that extracting this sort of meaning *from the text alone*, without using human readers,

[1] The metadata and annotations are in a language called XML. You will *not* have to learn it, but it's a useful skill if you're going to be doing a lot of data mining with stuff scraped from the Web, or from certain sorts of semi-structured data.

[2] Or the venerable programming language LISP, which is almost logic.

```
<nitf change.date="June 10, 2005" change.time="19:30" version="-//IPTC//DTD NITF 3.3//EN">
 - <head>
    <title>A Seated Tour Of Europe</title>
    <meta content="01JOHN$01" name="slug"/>
    <meta content="1" name="publication_day_of_month"/>
    <meta content="1" name="publication_month"/>
    <meta content="2004" name="publication_year"/>
    <meta content="Thursday" name="publication_day_of_week"/>
    <meta content="House & Home/Style Desk" name="dsk"/>
    <meta content="3" name="print_page_number"/>
    <meta content="F" name="print_section"/>
    <meta content="4" name="print_column"/>
    <meta content="Home and Garden; Style" name="online_sections"/>
  - <docdata>
      <doc-id id-string="1547299"/>
      <doc.copyright holder="The New York Times" year="2004"/>
      <series series.name="CURRENTS: FURNISHINGS"/>
    - <identified-content>
        <classifier class="indexing_service" type="descriptor">Chairs</classifier>
        <org class="indexing_service">Johnson & Hicks (NYC)</org>
        <person class="indexing_service">Louie, Elaine</person>
        <classifier class="online_producer" type="taxonomic_classifier">Top/Features/Home and Garden</classifier>
        <classifier class="online_producer" type="taxonomic_classifier">Top/Features/Style</classifier>
        <classifier class="online_producer" type="general_descriptor">Chairs</classifier>
      </identified-content>
    </docdata>
    <pubdata date.publication="20040101T000000" ex-ref="http://query.nytimes.com/gst/fullpage.html?res=9C07E4DE1E3EF932A35752C0A9629C8B63" item-length="174"
    name="The New York Times" unit-of-measure="word"/>
  </head>
 - <body>
  - <body.head>
    - <hedline>
        <hl1>A Seated Tour Of Europe</hl1>
      </hedline>
      <byline class="print_byline">By ELAINE LOUIE</byline>
      <byline class="normalized_byline">Louie, Elaine</byline>
    - <abstract>
      - <p>
          Chairs of 1920's and 30's are featured at Johnson & Hicks, new home furnishings store in TriBeCa; photos (M)
        </p>
      </abstract>
```


Figure 2: Example of the structured meta-data provided with the *New York Times Annotated Corpus*

turns out to be insanely hard. People in artificial intelligence have been saying it'll happen within the next twenty years since the 1950s; so we'll wish them good luck and leave them to their work.

A less ambitious sort of representation which still tries to get more or less explicitly at meanings would be to draw up a big list of categories, and then go over the documents and check of which categories they fall in to. This is basically what the annotating meta-data does for the *Times* corpus. Social scientists sometimes do this and call it **coding** the documents (Franzosi 2004); they'll often code for the emotional tone or **valence** as well. (This tends to be done with smaller, more focused sets of documents, so they can get away with smaller sets of categories.) Once we have this, our representation is a bunch of discrete variables, and there are lots of methods for analyzing discrete variables. But, again, extracting this sort of information *automatically* from the text is too hard for this to be useful to us[3]. In fact, the best automatic methods for this sort of thing use the bag-of-words representation which is coming up next.

Textual features

We don't know enough about how people extract meanings from texts to be able to automate it, but we do know that we extract meanings from texts — and different meanings from different texts[4]. This suggests that we should be able to use **features** or aspects of the text as proxies or imperfect signs of the actual meanings. And the text is right there in the computer already, so this should be easy. We just have to decide which textual features to use.

[3]With modern interactive websites, you can sometimes persuade your users to code documents (or images) by attaching "tags" to them. These can be surprisingly useful, but you are relying on strangers, and such "folksonomies" have self-consistency issues that more formalized ontologies lack. (If a novel is tagged "sf", does that tag have the same meaning as "science fiction" or "San Francisco" or something else?)

[4]Conversely, different people extract different meanings from the *same* texts, raising another set of issues, which we will ignore (though see Richards (1929)).

Think of what goes on in trying to distinguish between Saturn the brand of car, Saturn the planet, Saturn the mythological figure, Saturn the type of rocket, etc. Documents about all of these will of course contain the word "Saturn", so looking for that word wouldn't help us tell them apart. However, the *other* words in the document will tend to differ. "Saturn" together with words like "automobile", "wheels", "engine", "sedan" tends to indicate the car, whereas words like "rings", "hydrogen", "orbit", "Titan", "Voyager", "telescope", "Cassini", etc., indicate the planet. This suggests a *very* straightforward representation of the document: just list all the distinct words it contains.

That representation is actually a little *too* simple to work well in practice. The classic **bag-of-words (BoW)** representation is to list all of the distinct words in the document together with how often each one appears[5]. The name comes from imagining printing out the text, cutting the paper into little pieces so each word is on its own piece, and then throwing all the pieces into a bag. This is a definitely set of purely textual features (one per distinct word), and it's not hard for us to calculate it automatically from the data. What remains to be seen is whether we can actually do useful work with this representation.

Vectors

There are actually two different ways you could try to code up the bag-of-words, at least as I've described it so far. To illustrate this, let's look at the full text of the story featured in Figure **??** and see how to turn it into a bag of words.

> Lisa Weimer, right, opened her home furnishings store, Johnson & Hicks, in TriBeCa in September, and discovered her passion for chairs, especially those from the 1920's and 30's. "I love simple, clean lines and the richness of woods," said Ms. Weimer, who was once a home furnishings buyer at Bergdorf Goodman.
>
> A pair of French 1930's walnut chairs with checkerboard backs, above, are $8,500; steel folding chairs from the 1930's, originally used on a French ferry, are $575 each; tubular steel dining chairs upholstered in Ultrasuede, right, are 12 for $14,000. There are 500 chairs, and 100 tables. Johnson & Hicks is at 100 Hudson Street at Franklin Street. Information: (212) 966-4242.

Throwing away punctuation, and treating all numbers as just "#", we get the word-counts in Table 1.

There are (at least) two different data structures we could use to store this information. One is a list of **key-value pairs**, also known as an **associative array**, a **dictionary** or **hash**. The keys here would be the words, and the associated values would be the number of occurrences, or count, for each word. If a word does not appear in a document, that word is not one of its keys. Every document would have, in principle, its own set of keys. The order of the keys is entirely arbitrary; I printed them out alphabetically above, but it really doesn't matter.

It turns out, however, that it is a lot more useful to implement bags of words as **vectors**. Each component of the vector corresponds to a different word in the total **lexicon** of our document collection, in a fixed, standardized order. The value of the component would be the number of times the word appears, possibly including zero.

We use this vector bag-of-words representation of documents for two big reasons:

- There is a huge pre-existing technology for vectors: people have worked out, in excruciating detail, how to compare them, compose them, simplify them, etc. Why not exploit that, rather than coming up with stuff from scratch? (How would you measure the distance between two associate arrays?)
- In practice, it's proved to work pretty well.

To illustrate, I have, in the next code-chunk, taken ten random documents from the *Times* corpus — five of them about music, five about the arts (excluding music) — and turned them all into bag-of-words vectors[6].

[5] A bit of jargon is actually useful here. The bag-of-words vector has an entry for each word **type**, and that entry counts the number of **tokens** of that word. For example, this document (currently!) contains 7 tokens of the word type "metadata".

[6] You will get these documents, and more, in the first problem set.

#	1920s	1930s	a	above
9	1	3	3	1
and	are	at	backs	bergdorf
4	3	3	1	1
buyer	chairs	checkerboard	clean	discovered
1	4	1	1	1
especially	ferry	folding	for	franklin
1	1	1	2	1
french	from	furnishings	goodman	her
2	2	2	1	2
hicks	home	hudson	i	in
2	2	1	1	2
information	is	johnson	lines	lisa
1	1	2	1	1
love	ms	of	on	once
1	1	2	1	1
opened	originally	pair	passion	richness
1	1	1	1	1
right	said	september	simple	steel
2	1	1	1	1
store	street	tables	the	there
1	2	1	3	1
those	tribeca	used	walnut	was
1	1	1	1	1
weimer	who	with	woods	
2	1	1	1	

Table 1: Counts of the distinct words in the story, mapping all pure numbers to "#".

```
# Load functions for turning NYT Annotated Corpus documents into bag-of-word
# data frames
  # Having a separate source file does go against the spirit of complete
  # reproducibility somewhat, but this makes it easier for you to re-use
  # the code for your homework
source("http://www.stat.cmu.edu/~cshalizi/dm/19/hw/01/nytac-and-bow.R")

# read.doc() reads a document in and extracts the body of the story as a
# vector of words in order
# strip.text() does some tidying up (e.g. removes punctuation marks)
# table() turns the vector of strings into a table of _counts_ of words
  # table() is an R built-in, the others are from the provided code
music.1 <- table(strip.text(read.doc("nyt_corpus_small/music-0416371.xml")))
```

```
## Loading required package: XML
```

```
music.2 <- table(strip.text(read.doc("nyt_corpus_small/music-0430743.xml")))
music.3 <- table(strip.text(read.doc("nyt_corpus_small/music-0702928.xml")))
music.4 <- table(strip.text(read.doc("nyt_corpus_small/music-1343561.xml")))
music.5 <- table(strip.text(read.doc("nyt_corpus_small/music-1439011.xml")))
art.1 <- table(strip.text(read.doc("nyt_corpus_small/art-0437155.xml")))
art.2 <- table(strip.text(read.doc("nyt_corpus_small/art-1573733.xml")))
art.3 <- table(strip.text(read.doc("nyt_corpus_small/art-1590454.xml")))
art.4 <- table(strip.text(read.doc("nyt_corpus_small/art-1657584.xml")))
art.5 <- table(strip.text(read.doc("nyt_corpus_small/art-1812955.xml")))
small.bow <- make.BoW.frame(list(music.1,music.2,music.3,music.4,music.5,
                                 art.1,art.2,art.3,art.4,art.5),
                            row.names=c(paste("music", 1:5, sep="."),
                                        paste("art", 1:5, sep=".")))
```

There were 497 distinct word types in these ten documents (excluding "singletons" which only appeared in a single text). This means that each document is represented by a vector with 497 components — that we have 497 features. Obviously I can't show you these vectors, but I will show a few of these components (Table **??**).

```
kable(small.bow[,c("a","film","members","new","old","program","programs","sale","series","said","space"
```

	a	film	members	new	old	program	programs	sale	series	said	space	time	tuesday	wife
music.1	7	0	0	1	0	0	1	0	0	0	0	2	0	0
music.2	14	0	0	3	1	1	0	0	1	0	0	1	0	0
music.3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
music.4	12	0	0	2	0	0	0	0	0	0	0	2	2	0
music.5	12	0	0	1	0	1	0	0	1	0	2	0	0	0
art.1	48	15	0	3	1	0	0	0	1	11	0	0	0	0
art.2	26	0	0	2	1	1	0	0	0	10	0	1	0	0
art.3	29	0	5	1	0	0	1	2	0	11	0	0	0	1
art.4	14	0	1	6	0	3	2	0	2	0	1	0	1	0
art.5	35	1	0	3	0	0	2	1	9	4	0	6	1	1

Things to notice:

- The data take the form of a matrix. Each row corresponds to a distinct **item** (or instance **case**, **instance**, **unit**, **subject**, ...) — here, a document — and each column to a distinct feature. Often, the number of cases is written as n and the number of features is p. It is no coincidence that this is the same format as the data matrix $\mathbf{X}$ in linear regression.

- Look for contrasts between the documents about music and those about art. Some of them ("film", "sale") you could probably have guessed, if you'd thought about it — but what's going on with "said", or "wife"?
- The word "a" seems to appear more in stories about art than in stories about music. Why might this be? Do we really want to pay attention to it?
- Does it really make sense to distinguish here between "program" and "programs"?

Measuring Similarity

Right now, we are interested in saying which documents are similar to each other because we want to do search by content. But measuring **similarity** — or equivalently measuring **dissimilarity** or **distance** — is fundamental to data mining. Most of what we will do will rely on having a sensible way of saying how similar to each other different objects are, or how close they are in some geometric setting. Getting the right measure of closeness will have a huge impact on our results.

This is where representing the data as vectors comes in so handy. We already know a nice way of saying how far apart two vectors are, the ordinary or **Euclidean** distance, which we can calculate with the Pythagorean formula:

$$\|\vec{x} - \vec{y}\| \equiv \sqrt{\sum_{i=1}^{p} (x_i - y_i)^2}$$

where x_i, y_i are the i^{th} components of $\vec{x}$ and $\vec{y}$. Remember that for bag-of-words vectors, each distinct word — each entry in the lexicon — is a component or feature.

(The **Euclidean length** or **Euclidean norm** of any vector is

$$\|\vec{x}\| \equiv \sqrt{\sum_{i=1}^{p} x_i^2}$$

so the distance between two vectors is the norm of their difference $\vec{x} - \vec{y}$. Equivalently, the norm of a vector is the distance from it to the origin, $\vec{0}$.)

Now, there are other ways of measuring distance between vectors. Another possibility is the **taxicab** or **Manhattan** distance

$$\sum_{i=1}^{p} |x_i - y_i|$$

It's a perfectly good distance metric; it just doesn't happen to work so well for our applications.

Normalization

Just looking at the Euclidean distances between document vectors doesn't work, at least if the documents are at all different in size. Instead, we need to **normalize** by document size, so that we can fairly compare short texts with long ones. There are (at least) two ways of doing this.

Document word-count normalization

Divide the word counts by the total number of words in the document. In symbols,

$$\vec{x} \mapsto \frac{\vec{x}}{\sum_{i=1}^{p} x_i}$$

Notice that all the entries in the normalized vector are non-negative fractions, which sum to 1. The i^{th} component is thus the probability that if we pick a word out of the bag at random, it's the i^{th} entry in the lexicon.

In the R code provided, there's a function, `div.by.sum()`, which will take a matrix and divide each row by the sum of its entries, giving a new matrix where each row sums to 1. Applying this function before calculating distances is equivalent to normalizing by word count.

Euclidean length normalization

Divide the word counts by the Euclidean length of the document vector:

$$\vec{x} \mapsto \frac{\vec{x}}{\|\vec{x}\|}$$

For search, normalization by Euclidean length tends to work a bit better than normalization by word-count, apparently because the former de-emphasizes words which are rare in the document.

The accompanying R code provides a function `div.by.euc.length()` which similarly divides each row in a matrix by its Euclidean norm.

Cosine similarity

For our purposes, saying "there's little distance between documents" is the same as saying "these documents are very similar", so search and other data-mining applications often use measures of similarity rather than distance. One which has proved to work well is the "cosine similarity", which is the normalized inner product between vectors:

$$d_{\cos}(\vec{x}, \vec{y}) = \frac{\sum_i x_i y_i}{\|\vec{x}\|\|\vec{y}\|}$$

Inverse document frequency

Someone asked in class last time about selectively paying less attention to certain words, especially common words, and more to the rest. This is an excellent notion. Not all features are going to be equally useful, and some words are so common that they give us almost no ability at all to discriminate between relevant and irrelevant documents. In (most) collections of English documents, looking at "the", "of", "a", etc., is a waste of time[7]. We could handle this by a fixed list of **stop words**, which we just don't count, but this at once too crude (all or nothing) and too much work (we need to think up the list).

Inverse document frequency (IDF) is a more adaptive approach. The **document frequency** of a w is the number of documents it appears in, n_w. The IDF weight of w is

$$IDF(w) \equiv \log \frac{n}{n_w}$$

where n is the total number of documents. Now when we make our bag-of-words vector for the document x, the number of times w appears in x, x_w, is multiplied by $IDF(w)$. Notice that if w appears in every document, $n_w = n$ and it gets an IDF weight of zero; we won't use it to calculate distances. This takes care of most of the things we'd use a list of stop-words for, but it also takes into account, implicitly, the kind of documents we're using. (In a collection of scientific papers on genetics, "gene" and "DNA" are going to have IDF weights of zero, or nearly zero.) On the other hand, if w appears in only a few documents, it will get a weight of about $\log n$, pushing all documents containing w closer to each other. (In a corpus of scientific-papers-in-general, "DNA" will have a positive IDF weight, pushing genetics papers together.)

[7] A possible exception is in **stylometry**, where we try to assign documents to authors by looking at subtle variations in word choice or other writing habits.

You could tell a similar story about any increasing function of n/n_w, not just log, but log happens to work very well in practice, in part because it's not very sensitive to the exact number of documents[8]. Notice also that using IDF is *not* guaranteed to help. Even if w appears in every document, so $IDF(w) = 0$, it might be common in some of them and rare in others, so we'll ignore what might have been useful information. (Maybe genetics papers about laboratory procedures use "DNA" more often, and papers about hereditary diseases use "gene" more often, even though everyone uses both words.)

Practice the Criterion of Truth

I've been pretty free with saying that things work or they don't work, without being at all concrete about what I mean by "working". We are going to see many, many different ways of elaborating on "working", but for right now, a first cut is to look at which stories come closest to each story in our little collection:

```
rownames(small.bow)[nearest.points(small.bow)$which]
```

```
##  [1] "music.5" "music.5" "music.1" "music.5" "music.1" "art.5"   "art.5"
##  [8] "music.2" "music.4" "art.2"
```

(Notice that all but one of these are matched to a document in the same class.)

I am going to leave it as an exercise to you to re-do this, normalizing the vectors either by word-count or by Euclidean length.

Abstraction

Let's step back and summarize how we've set ourselves up to search documents by content.

1. We gather a large collection of potentially-relevant documents.
2. We calculate a bag-of-words vector from each document.
    a. At this step, we may do some cleaning or pre-processing, like removing punctuation marks or capitalization.
    b. We gather the vectors into an $n \times p$ matrix.
3. We normalize the vectors and/or weight their components, if we want.
4. We calculate the $n \times n$ matrix of distances between documents.
5. We find the closest matching document(s) to any document of interest.

It's important to note here that only steps (1) and (2) actually use the documents. Everything else after that uses only the feature vectors. In programming terms, we'd say that the feature vectors are an **abstraction** of the documents, and thereafter we're working at the abstract level.

There are two reasons programmers use abstraction: convenience, and generalization. "Convenience" here means that we don't have to keep dealing with the actual documents, just their representations, and we're mathematically trained enough to find vectors convenient to work with. But "generalization" means that we can use the *same* tools for *different* objects, so long as they have the same kind of abstraction.

What that in turn means is that we haven't just seen how to do similarity search for text documents. We've seen how to similarity search for images, music, social-network nodes, credit card transactions, phone calls, etc., etc. — for anything where we can represent in terms of feature vectors. This is the great power and attraction of abstraction. This is in fact why we can have a course on data mining *in general*, rather than many courses on data mining for text (what kind of text?), images (what kind of images?), music, credit-card transactions, etc., etc.

There are two big drawbacks to abstraction: the representation you're using may not capture much (or any) useful information about the underlying items, and you can't *tell* whether it is working well from the

[8] So this is not the same log we will see later in information theory, or the log in psychophysics.

abstraction alone. Intuitively, the bag-of-words representation of documents feels like it should work better than the bag-of-letters representation, though the later would have only 26 dimensions and would in many ways be much more convenient. But they're equally good vector-space representations, and nothing you do to the *vectors* will be able to tell you that words are a useful level of abstracting text for search, but letters are not. Nor is anything we do to the vectors alone going to tell us that we want to use, say, normalization by word-count and inverse document frequency weighting. To decide on the *right* representation, we need to be able to go back and connect the abstract feature vectors, or things we've calculated from them, to either the items themselves, or to other information about those items. Looking at whether the closest match to a story about art is another story about art is a first step towards that sort of checking.

Exercises

Do not turn these in, but do think them through.

1. Why is it called the "Manhattan metric" or "taxicab metric"? (Think before looking this up.)
2. Why is it called the "cosine similarity"? (Again, think before looking this up.)
3. Why do we need to normalize bag-of-word vectors to compare documents with different sizes?
4. Why does normalization by Euclidean length de-emphasize a document's rare words more strongly than normalization by word count? *Hint:* think about the relationship between $\sum_i |x_i|$ and $\|\vec{x}\|$.
5. Why is the cosine distance "the cosine of the angle between the two vectors"?
6. Explain how the cosine distance is related to the Euclidean-length normalized distance between two vectors.

References

Franzosi, Roberto. 2004. *From Words to Numbers: Narrative, Data, and Social Science*. Cambridge, England: Cambridge University Press.

Richards, I. A. 1929. *Practical Criticism: A Study of Literary Judgment*. London: Kegan Paul.

Sandhaus, Evan. 2008. "The New York Times Annotated Corpus." Philadelphia: Electronic database; Linguistic Data Consortium. http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2008T19.