CS 429: Information Retrieval

Dr. Aron Culotta

Illinois Institute of Technology

Spring 2014

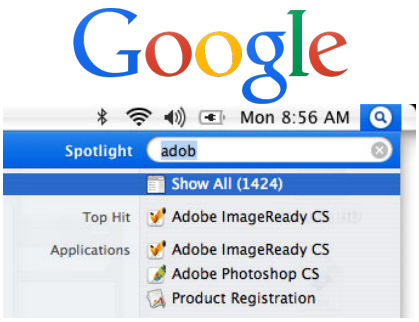
What is Information Retrieval?

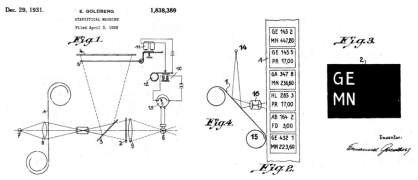
The process of finding relevant data.

Typically:

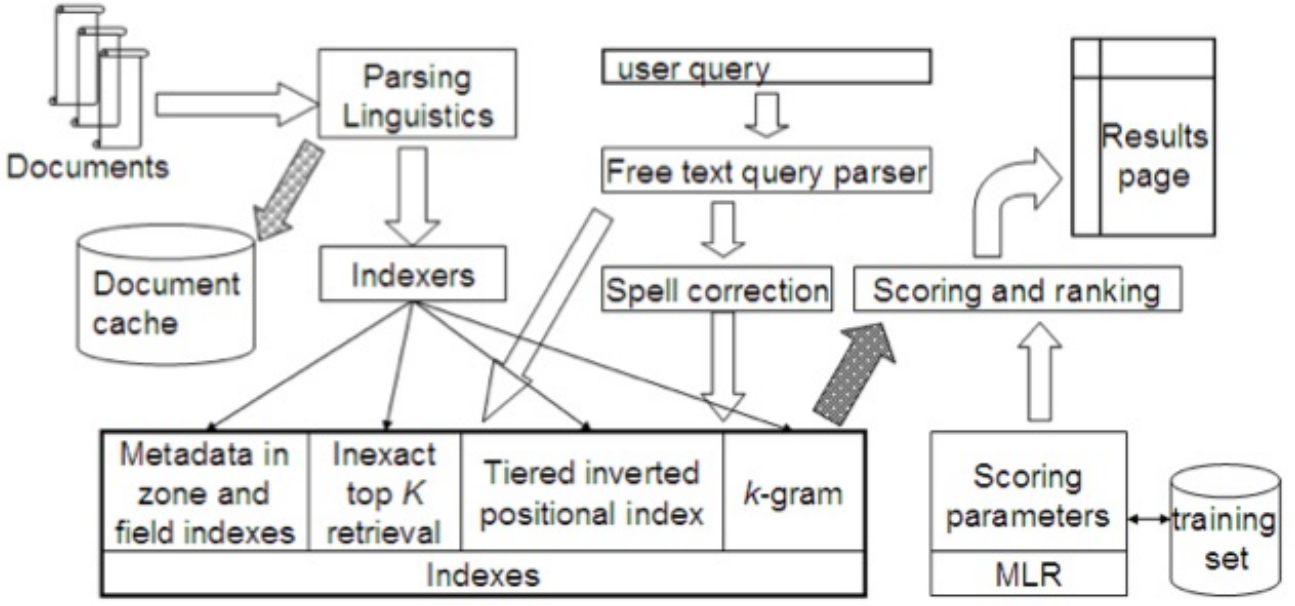
- text, though also images, video, audio
- *unstructured* (in contrast to relational databases)
- using a computer

Examples of Information Retrieval





Architecture



► **Figure 7.5** A complete search system. Data paths are shown primarily for a free text query.

Simplest information retrieval problem

In [14]:

```
documents = ['dog', 'cat', 'zebra', 'cat']
```

In [26]:

```
def search(documents, query):
    return [doc for doc in documents if doc == query]
```

In [27]:

```
'return [doc for doc in documents if doc == query]'
```

Out[27]:

```
'return [doc for doc in documents if doc == query]'
```

In [28]:

```
print search(documents, 'cat')
```

```
['cat', 'cat']
```

Runtime?

T(n) = O(n) , where n=`len(documents)`. **Can we do better?**

Most documents have more than one word...

In [31]:

```
documents = [['dog', 'cat'], ['cat', 'zebra'], ['dog', 'puma']]
```

In [29]:

```
def search(documents, query):
    return [doc for doc in documents if query in doc]
```

In [ ]:

```
' return [doc for doc in documents if query in doc]'
```

In [32]:

```
print search(documents, 'cat')
```

```
[['dog', 'cat'], ['cat', 'zebra']]
```

Runtime?

Naive: O(n*m) , where n= `len(documents)` and m= `max(len(d) for d in documents)`

Inverted Index

Map from *word* \rightarrow *Postings List*

Postings List: List of ids for documents containing the word.

In [33]:

```
# Map each word to the list of indices of documents that contain it.
index = {'dog': [0, 2],
         'cat': [0, 1],  # IDs are sorted. Why?
```

```
        'zebra': [1],
        'puma': [2]}
```

In [34]:

```
def indexed_search(documents, index, query):
    return [documents[doc_id] for doc_id in index[query]]
```

In []:

```
'    return [documents[doc_id] for doc_id in index[query]] '
```

In [37]:

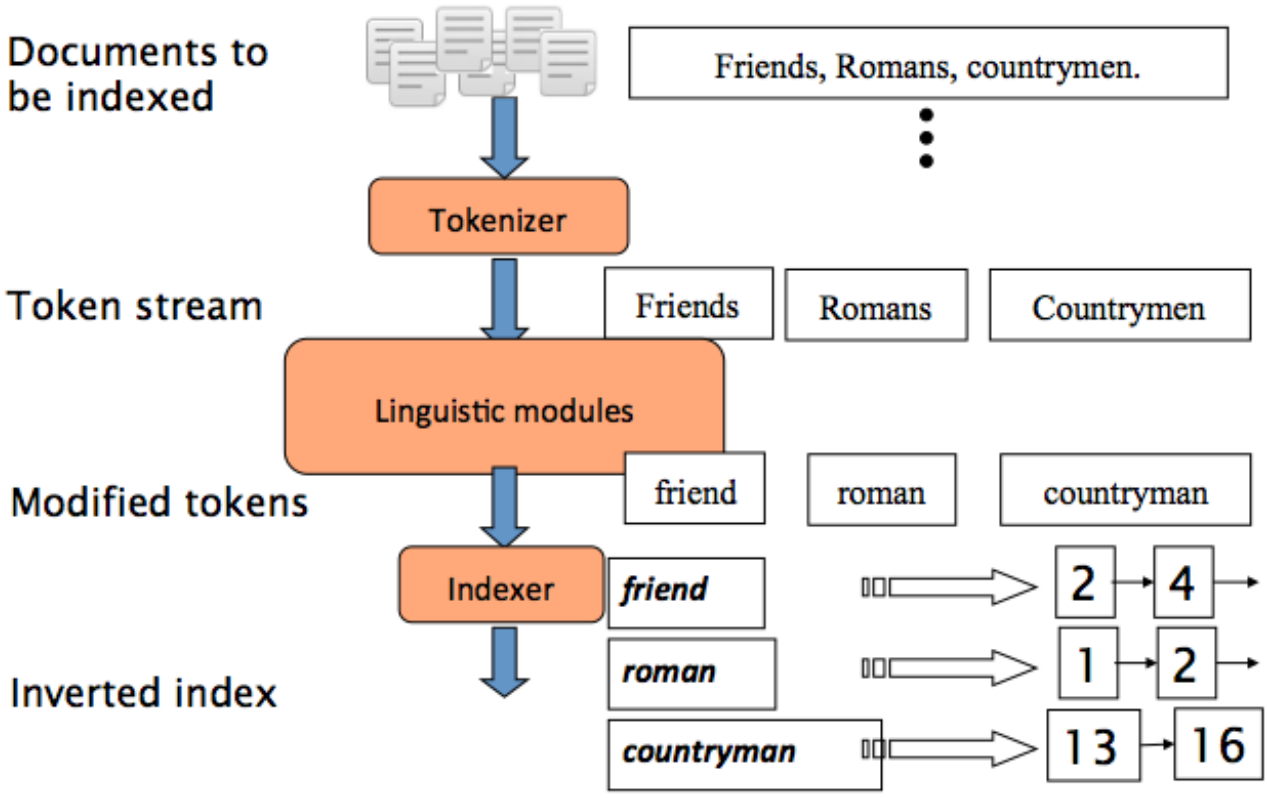
```
print indexed_search(documents, index, 'cat')
```

```
[['dog', 'cat'], ['cat', 'zebra']]
```

Runtime?

O(k) , where k is number of matching documents.

Building an Index



(Source: MRS)

Query Processing

Most queries have more than one word:

dog AND **cat**

Index:
dog \rightarrow \{0, 2\}
cat \rightarrow \{0, 1\}

In [38]:

```
def and_search(documents, index, queries):
    doc_ids = set([doc_id for doc_id in index[queries[0]]])
    for query in queries[1:]: # For remaining words in query
        doc_ids &= set([doc_id for doc_id in index[query]]) # Set intersection
    return [documents[doc_id] for doc_id in doc_ids]
```

In [41]:

```
print and_search(documents, index, ['cat', 'dog'])
```

```
[['dog', 'cat']]
```

Runtime?

Depends on set intersection computation.

Merging Postings Lists

dog \rightarrow \{0, 2, 4, 11, 31, 45, 173, 174\}
cat \rightarrow \{2, 31, 54, 101\}

Intersection \rightarrow \{2, 31\}

Idea: maintain pointers to both lists and walk through both simultaneously

Running time: O(x + y) , where x and y are lengths of two postings lists

Query Optimization

Organize query processing efficiently.

dog \rightarrow \{0, 2, 4, 11, 31, 45, 173, 174\}
cat \rightarrow \{2, 31, 54, 101\}
zebra \rightarrow \{31, 506\}

dog AND **cat** AND **zebra**

Which order?

1. (**dog** AND **cat**) AND **zebra**
2. **dog** AND (**cat** AND **zebra**)
3. (**dog** AND **zebra**) AND **cat**

#2 reduces work by processing rare words first.

What about **dog** AND NOT **cat**?

Course Information

Course information

- **GitHub** will be primary source of course information.
  - https://github.com/iit-cs429/main
- Assignments turned in to GitHub
- Let's look at the syllabus, schedule, and first assignment

Survey results

In [42]:

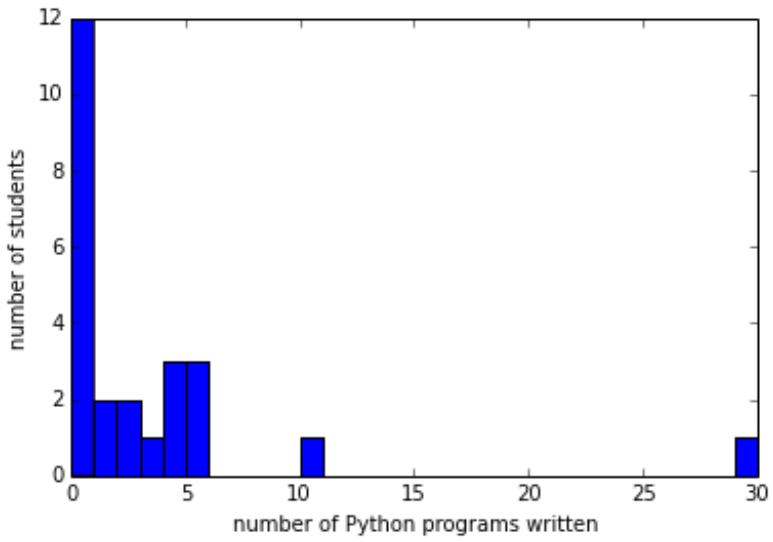
```
counts = [float(line) for line in open('python_counts.txt')]
print counts
```

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 2.0, 2.0
, 3.0, 4.0, 4.0, 4.0, 5.0, 5.0, 5.0, 10.0, 30.0]
```

In [43]:

```
# This allows us to plot directly to the notebook.
% pylab inline
figure()
hist(counts, 30)
xlabel('number of Python programs written')
ylabel('number of students')
show()
```

```
Populating the interactive namespace from numpy and matplotlib
```



In [ ]:

CS 429: Information Retrieval

Lecture 2: Indexing

Dr. Aron Culotta

Illinois Institute of Technology

Spring 2014

Indexing Pipeline

1. Collect documents
2. Tokenize
3. Normalize
4. Index

Indexing Pipeline

document $\xrightarrow{tokenize}$ (tokens, types) $\xrightarrow{normalize}$ terms $\xrightarrow{index}$ inverted index

In [1]:

```
# 1. Collect documents.
document = "he didn't know where he worked."

# 2. Tokenize
tokens = ["he", "didn't", "know", "where", "he", "worked"] # split; remove punct
uation
types = ["he", "didn't", "know", "where", "worked"] # unique tokens.

# 3. Normalize
# remove common words like 'where'; collapse word forms like worked -> work
terms = ["he", "didnt", "know", "work"]

# 4. Index
index = {'he': [0],
         'didnt': [0],
         'know' : [0],
         'work' : [0]}
```

e·quiv·a·lence class (/i'kwivələns klas/) *n.*

1. A subset whose elements are equivalent according to some relation $\sim$.

   $S' \subseteq S \hspace{.2cm}$ s.t. $\hspace{.2cm}x_i \sim x_j \hspace{.1cm} \forall x_i, x_j \in S'$

   E.g., consider the set {dog, cat, spider} and the relation "has same number of legs". Then there are two equivalence classes: {dog, cat} and {spider}.

to·ken (/tōkən/) *n*.

1. A sequence of characters in a document that form a meaningful unit.
2. The output of a *tokenizer*.

type (/tīp/) *n.*

1. An equivalence class of *tokens* under the string equality relation.
2. By analogy to OO-programming, class:object :: type:token

   e.g., "to be or not to be" $\xrightarrow{tokenize}$ *tokens={to, be, or, not, to be}* $\hspace{.2cm}$ *types={to, be, or, not}*

   The type 'to' is an equivalence class containing the first and fifth tokens of the document.

term (/tərm/) *n.*

1. An equivalence class of *types* under the relation "have the same normalized form."
2. The keys in the inverted index.

   e.g., types={John, john, aren't, arent} $\xrightarrow{normalize}$ terms={john, arent}.

   The term "john" is an equivalence class containing types {John, john}.

   The term "arent" is an equivalence class containing types {arent, aren't}.

Tokenization

to·ken·i·za·tion (/ˈtōkən izā-shən/) *n.*

1. The process of splitting a document into tokens.

   Simplest approach: split on whitespace.

In [2]:

```
print document.split()
```

```
['he', "didn't", 'know', 'where', 'he', 'worked.']
```

Tokenization: Compound Nouns

- *San Francisco*; *New York University* vs *York University*
- Solved somewhat by *phrase indexing* (next class)

Tokenization: Segmentation

- *Lebensversicherungsgesellschaftsangestellter*
  - "life insurance company employee"
- 我不能说中国话
- *#androidgames*

- Statistical classification algorithms can be used to split (Part III of course).
- Simpler: index character subsequences (*n-grams*).
  - E.g., *#androidgames* $\rightarrow$ {*#andr, andro, ndroi, droid, roidg, ..., games*}

Tokenization: Punctuation

- Remove all punctuation?

- "didn't", "www.google.com"
- CAR vs C.A.R.
- O'Neill vs ONeill vs O Neill

Tokenization: Regular Expressions

symbol	meaning
\b	Word boundary (zero width)
\d	Any decimal digit (equivalent to [0-9])
\D	Any non-digit character (equivalent to [^0-9])
\s	Any whitespace character (equivalent to [\t\n\r\f\v]
\S	Any non-whitespace character (equivalent to [^ \t\n\r\f\v])
\w	Any alphanumeric character (equivalent to [a-zA-Z0-9_])
\W	Any non-alphanumeric character (equivalent to [^a-zA-Z0-9_])
\t	The tab character
\n	The newline character

(source: http://nltk.org/book/ch03.html)

In [3]:

```
import re  # Regular expression module
print re.split('x', 'axbxc')
```

```
['a', 'b', 'c']
```

In [4]:

```
print re.split('\+\+', 'hi+++there')
```

```
['hi', '+there']
```

In [5]:

```
print re.split('([\W\s]|t)', "what's up?")
```

```
['wha', 't', '', "'", 's', ' ', 'up', '?', '']
```

In [6]:

```
text = "A first-class ticket to the U.S.A. isn't expensive?"
print re.split(' ', text)
```

```
['A', 'first-class', 'ticket', 'to', 'the', 'U.S.A.', "isn't", 'expensive?']
```

How to remove punctuation?

In [7]:

```
print re.split('\W+', text)           # \W=not a word character; +=1 or more
```

```
['A', 'first', 'class', 'ticket', 'to', 'the', 'U', 'S', 'A', 'isn', 't', 'expe
nsive', '']
```

In [8]:

```
print re.findall('\w+', text)         # \w=a word character [a-zA-Z0-9_]
```

```
['A', 'first', 'class', 'ticket', 'to', 'the', 'U', 'S', 'A', 'isn', 't', 'expe
nsive']
```

In [9]:

```
# group punctuation with following letters
print re.findall('\w+|\S\w*', text)  # \S=not a space; |=OR
```

```
['A', 'first', '-class', 'ticket', 'to', 'the', 'U', '.S', '.A', '.', 'isn', "'
t", 'expensive', '?']
```

How to keep hyphenated words and contractions together?

In [10]:

```
print re.findall("\w+(?:[-']\w+)*|[-.(]+|\S\w*", text)
# (?: specifies what to match, not what to capture
```

```
['A', 'first-class', 'ticket', 'to', 'the', 'U', '.', 'S', '.', 'A', '.', "isn'
t", 'expensive', '?']
```

In [11]:

```
print re.findall("(?:[A-Z]\.)+|\w+(?:[-']\w+)*|[-.(]+|\S\w*", text)
```

```
['A', 'first-class', 'ticket', 'to', 'the', 'U.S.A.', "isn't", 'expensive', '?'
]
```

Normalization

nor·mal·iz·a·tion (/ˈnôrməˌlizā-shən/) *n.*

1. The process of clustering *types* into *terms*.

Issues include: removing common words, special characters, casing, morphology

Normalization: Stop words

- Exclude common words
  - *the*, *a*, *be*
- Why?

- save space (length of postings list is huge!)
- no semantic content (?!)

"*to be or not to be*" is all stop words!

Accents/Diacritics

- naive vs. naïve
- pena (sorrow) vs peña (cliff)

- What will users enter?

Case

- Typically, just convert everything to lowercase.
- E.g., search Google for CAT -cat.

Stemming / Lemmatizing

mor·phol·o·gy (/môrˈfäləjē/) *n.*

1. (*Linguistics*) The study of the rules governing how words may take different forms in a language.

E.g.

- Pluralization: *dog* $\xrightarrow{pluralize}$ *dogs* ; *goose* $\xrightarrow{pluralize}$ *geese*

- Tense: *play* $\xrightarrow{past.tense}$ *played* ; *go* $\xrightarrow{past.tense}$ *went*

stem (/stem/) *v.*

1. To normalize based on crude morphology heuristics.

   E.g. remove all "*-s*" *and* "*-ed*" *suffixes*

lem·ma·tize (ˈlemə¸tīz/) *v.*

1. To create equivalence classes of word types using the morphological rules of a language.

   Often relies on ***part-of-speech*** *tagging to select rules*.

   E.g. if bed *is a noun, then do not remove* -ed *suffix.*

Simple stemmer

In [12]:

```
def stem(word):
    for suffix in ['ies', 's', 'ed', 'ing']: # order matters!
        if word.endswith(suffix):
            return word[:-len(suffix)]
```

What can go wrong?

Stemming Errors

- **over-stemming**: merge types that should not be merged.
- **under-stemming**: fail to merge types that should be merged.

In [13]:

```
types = ['tied', 'ties', 'tis', 'bed', 'cities']
print '\n'.join([stem(w) for w in types])
```

```
ti
t
ti
b
cit
```

How does this affect search?

Porter Stemmer

- Very commonly used stemmer with a complex set of heuristics.

In [14]:

```
from nltk.stem import PorterStemmer # See nltk.org (`pip install nltk`)
porter = PorterStemmer()
print types
print '\n'.join([porter.stem(x) for x in types])
```

```
['tied', 'ties', 'tis', 'bed', 'cities']
tie
tie
ti
bed
citi
```

In [15]:

```
types = ['bed', 'kiss',
         'tied', 'tis',
         'universal', 'university',
         'experiment', 'experience',
         'past', 'paste',
         'alumnus', 'alumni',
         'adhere', 'adhesion',
         'create', 'creation']
```

```
porter_results = [porter.stem(x) for x in types]
print '\n'.join(porter_results)
```

```
bed
kiss
tie
ti
univers
univers
experi
experi
past
past
alumnu
alumni
adher
adhes
creat
creation
```

WordNet Lemmatizer

In [16]:

```
from nltk.stem.wordnet import WordNetLemmatizer
# See description: https://wordnet.princeton.edu/wordnet/man/morphy.7WN.html
lemm = WordNetLemmatizer()
lemm_results = [lemm.lemmatize(x) for x in types]
print 'type, porter, lemmatizer\n'
print '\n'.join([str(t) for t in zip(types, porter_results, lemm_results)])
```

```
type, porter, lemmatizer

('bed', 'bed', 'bed')
('kiss', 'kiss', 'kiss')
('tied', 'tie', 'tied')
('tis', 'ti', 'ti')
('universal', 'univers', 'universal')
('university', 'univers', 'university')
('experiment', 'experi', 'experiment')
('experience', 'experi', 'experience')
('past', 'past', 'past')
('paste', 'past', 'paste')
('alumnus', 'alumnu', 'alumnus')
('alumni', 'alumni', 'alumnus')
('adhere', 'adher', 'adhere')
('adhesion', 'adhes', 'adhesion')
('create', 'creat', 'create')
('creation', 'creation', 'creation')
```

A principled approach?

Given the many number of ways to preprocess text, how do we know which one is best?

Approaches:

- Assume types that appear in similar contexts can be merged.
  - e.g., *universally* and *universal* appear in similar documents, but not *university*.

- Learn from user behavior
  - e.g., users click on very different search results if they search for *universal* vs *university*.

We'll explore both later in the course.

CS 429: Information Retrieval

Lecture 3: Indexing II

Dr. Aron Culotta

Illinois Institute of Technology

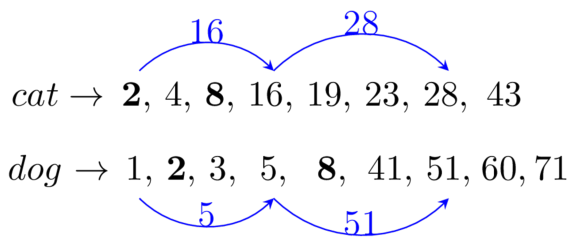
Spring 2014

Recall Inverted Index

$$\begin{aligned} cat &\rightarrow \mathbf{2}, 4, \mathbf{8}, 16, 19, 23, 28, 43 \\ dog &\rightarrow 1, \mathbf{2}, 3, 5, \mathbf{8}, 41, 51, 60, 71 \\ \mathbf{merged} &\rightarrow 2, 8 \end{aligned}$$

Runtime: $O(x + y)$, for postings lists of size x and y

Skip Lists



Worst-case runtime?

$O(x + y)$

Best-case runtime?

$O(k)$, for k matching documents

Merging Skip Lists

```
In [1]:
```

```
# tuple (x,y,z): x=doc_id, y=skip index, z=skip value
index = {'cat': [(2, 3, 16), 4, 8, (16, 6, 28), 19, 23, 28, 43],
         'dog': [(1, 3, 5), 2, 3, (5, 6, 51), 8, 41, 51, 60, 71]}
```

In [2]:

```
# Print postings list containing skip pointers.
def print_skip_list(docs):
    idx = 0
    while idx < len(docs):
        print docs[idx]
        if type(docs[idx]) is tuple: # skip
            idx = docs[idx][1]
        else:
            idx += 1
```

In [3]:

```
print_skip_list(index['cat'])
```

```
(2, 3, 16)
(16, 6, 28)
28
43
```

In [4]:

```
print_skip_list(index['dog'])
```

```
(1, 3, 5)
(5, 6, 51)
51
60
71
```

type

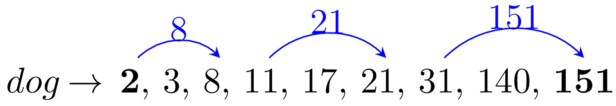
```
INTERSECTWITHSKIPS(p1, p2)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4        then ADD(answer, docID(p1))
 5             p1 ← next(p1)
 6             p2 ← next(p2)
 7        else if docID(p1) < docID(p2)
 8                then if hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
 9                     then while hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
10                          do p1 ← skip(p1)
11                     else p1 ← next(p1)
12                else if hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
13                     then while hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
14                          do p2 ← skip(p2)
15                     else p2 ← next(p2)
16  return answer
```

► **Figure 2.10** Postings lists intersection with skip pointers.

Where to insert skip pointers?

Tradeoff:

- More pointers mean more opportunities to skip
- Fewer pointers means less time wasted comparing to skip values.
- Heuristic: $\sqrt{n}$ evenly-spaced pointers, for list of size n.



Adding to an index with skip pointers

What happens when we have to add a document to a postings list?

If postings list is a ...

- linked list
- dynamic array (e.g., ArrayList)

Phrase queries

"cat dog" **vs** cat AND dog

Phrase Indexing

Two approaches

1. **Biword Index**
2. **Positional Index**

Biword index

"The cat dog jumped."

$$
\begin{aligned}
the &\rightarrow 1, 2, 3, 4, 8, 11, 17, 21, 31, 41, 64, 128, 140, 151 \\
cat &\rightarrow 1, 2, 4, 41, 64, 128, 151 \\
dog &\rightarrow 1, 2, 3, 8, 11, 17, 21, 31, 140, 151 \\
jumped &\rightarrow 2, 151 \\
\text{“}the\ cat\text{”} &\rightarrow 1, 2, 4, 41, 64, 128, 151 \\
\text{“}cat\ dog\text{”} &\rightarrow 1, 2, 151 \\
\text{“}dog\ jumped\text{”} &\rightarrow 2, 151
\end{aligned}
$$

Finding phrases

In [5]:

```
docs = [l.strip() for l in open("documents.txt", 'rt').readlines()]
print 'read', len(docs), 'docs'
```

```
read 62 docs
```

In [6]:

```
def ngrams(n, docs):
    terms = set()
    for d in docs:
        toks = d.split()
        for i in range(len(toks) - n + 1):
            terms.add('_'.join(toks[i:i+n]))
    return terms
```

In [7]:

```
print ngrams(1, ['a b c'])
print ngrams(2, ['a b c'])
print ngrams(3, ['a b c'])
print ngrams(4, ['a b c'])
```

```
set(['a', 'c', 'b'])
set(['b_c', 'a_b'])
set(['a_b_c'])
set([])
```

In [8]:

```
max_n = 10
sizes = [len(ngrams(i, docs)) for i in range(1, max_n)]
print 'number of terms=', zip(range(1, max_n), sizes)
```

```
number of terms= [(1, 380), (2, 585), (3, 599), (4, 567), (5, 511), (6, 452), (
7, 391), (8, 330), (9, 272), (10, 221), (11, 172), (12, 129), (13, 90), (14, 61
), (15, 40), (16, 28), (17, 22), (18, 17), (19, 14), (20, 11), (21, 10), (22, 9
), (23, 8), (24, 7), (25, 6), (26, 5), (27, 4), (28, 3), (29, 2), (30, 1), (31,
0), (32, 0), (33, 0), (34, 0), (35, 0), (36, 0), (37, 0), (38, 0), (39, 0), (40
, 0), (41, 0), (42, 0), (43, 0), (44, 0), (45, 0), (46, 0), (47, 0), (48, 0), (
49, 0), (50, 0), (51, 0), (52, 0), (53, 0), (54, 0), (55, 0), (56, 0), (57, 0),
(58, 0), (59, 0), (60, 0), (61, 0), (62, 0), (63, 0), (64, 0), (65, 0), (66, 0)
, (67, 0), (68, 0), (69, 0), (70, 0), (71, 0), (72, 0), (73, 0), (74, 0), (75,
0), (76, 0), (77, 0), (78, 0), (79, 0), (80, 0), (81, 0), (82, 0), (83, 0), (84
, 0), (85, 0), (86, 0), (87, 0), (88, 0), (89, 0), (90, 0), (91, 0), (92, 0), (
93, 0), (94, 0), (95, 0), (96, 0), (97, 0), (98, 0), (99, 0)]
```

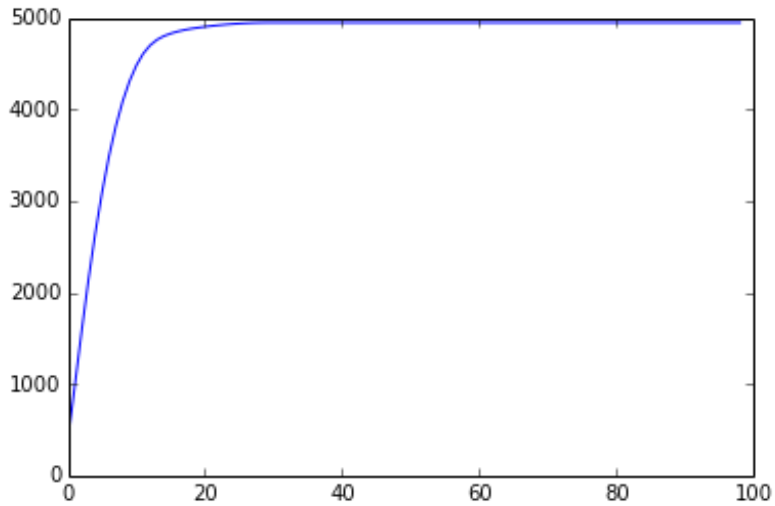
In [9]:

```
%pylab inline
# 1-grams, 1-grams + 2-grams, ...
x = [sum(sizes[:i]) for i in range(1,max_n)]
print x
plot(x)
```

```
Populating the interactive namespace from numpy and matplotlib
[380, 965, 1564, 2131, 2642, 3094, 3485, 3815, 4087, 4308, 4480, 4609, 4699,
4760, 4800, 4828, 4850, 4867, 4881, 4892, 4902, 4911, 4919, 4926, 4932, 4937, 49
41, 4944, 4946, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947
, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947,
4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 49
47, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947
, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947, 4947,
4947, 4947, 4947, 4947, 4947, 4947]
```

Out[9]:

```
[<matplotlib.lines.Line2D at 0x10da61110>]
```



Limits of phrase indices

If we index 5-grams, how can we search for the phrase "to be or not to be"?

- AND of 5-grams "to be or not to" AND "be or not to be"
- Very small possibility of a false match

What if we index 2-grams and we search for the phrase "new york university"

- "new york" AND "york university"
- greater possibility of false match

Positional Index

- Store position of term in original document.
- *term*: [(doc_id1, [pos1, pos2, ...]), (doc_id2, [pos1, pos2, ...]), ...]

In [10]:

```
doc0 = "The cat dog jumped over the dog."
doc1 = "The dog jumped."
index = {
          'the': [(0, [0, 5]), (1, [0])],
          'cat': [(0, [1])],
          'dog': [(0, [2, 6]), (1, 1)],
          'jumped': [(0, [3]), (1, [2])]
         }
```

Positional Index

- Additional space needed?

- One `int` for each time a term occurs in a document.
- Biggest impact on long documents.
- E.g., consider a term that occurs once every thousand words:

document length	# postings	# positional postings
1000	1	1
100,000	1	100

Merging positional postings lists

How can we efficiently merge positional postings lists to find phrases?

In [11]:

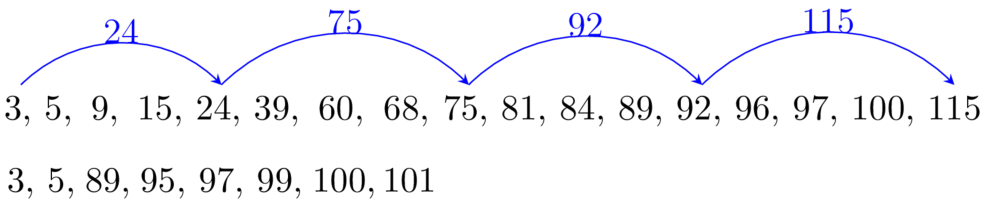
```
index = {'cat': [(0, [1])],
         'dog': [(0, [2, 6]),
                 (1, [1])]}
# [ (doc_id1, [pos1, pos2, ...]),
#   (doc_id2, [pos1, pos2, ...]), ...
# ]

# Search for "cat dog"
# This is inefficient! See Figure 2.12 (from book) and next assignment for more
.
```

```
for cat_doc in index['cat']:
    for dog_doc in index['dog']:
        if cat_doc[0] == dog_doc[0]:  # In same document
            print 'both appear in ', cat_doc[0]
            for cat_pos in cat_doc[1]:
                for dog_pos in dog_doc[1]:
                    if cat_pos == dog_pos - 1: # dog comes right after cat.
                        print 'found "cat dog" at positions', cat_pos, dog_pos
```

```
both appear in  0
found "cat dog" at positions 1 2
```

Combining Biword Index and Positional Index

- Store only phrases that are
  - Commonly queried
  - Individual words are common
- *Britney Spears* vs. *The Who*



$3, 5, 89, 95, 97, 99, 100, 101$

1. How often is a skip pointer followed (i.e., p1 is advanced to skip(p1))?
2. How many postings comparisons will be made by this algorithm while intersecting the two lists?
3. How many postings comparisons would be made if the postings lists are intersected without the use of skip pointers?

CS 429: Information Retrieval

Lecture 4: Dictionaries

Dr. Aron Culotta

Illinois Institute of Technology

Spring 2014

Last time:

- skip lists, phrase search, biword index, positional index

Today:

- Efficient retrieval of postings lists
- Wildcard queries
- Spelling correction

Recall our friend the inverted index:

\begin{eqnarray*} cat & \rightarrow &1,9,62 \\ dog & \rightarrow &1,2,9,31 \\ zebra & \rightarrow &2,62,150 \\ \end{eqnarray*}

Given a query term "dog", how can we efficiently retrieve the matching postings list?

dictionary: data structure to lookup posting list of a term.

What data structure should we use?

- Hash table
- Binary tree
- B-tree

Hash Table

In [134]:

```
index = {'cat': [1, 9, 62],
         'dog': [1, 2, 9, 31],
         'zebra': [2, 62, 150]}
```

In [135]:

```
query = 'dog'
print index['dog']  # hash lookup, O(1)
```

```
[1, 2, 9, 31]
```

How does this work?

In [150]:

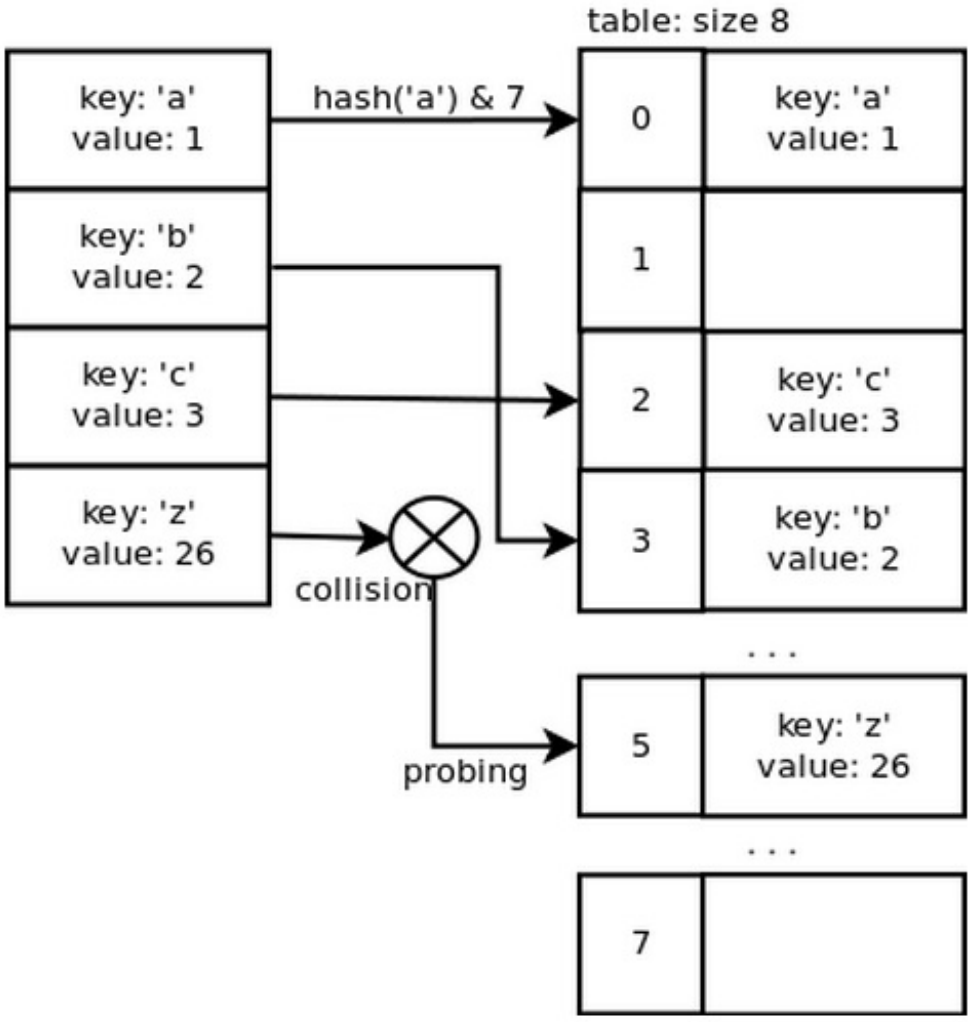
```
print hash('dog')
print hash('cat')
# See the Python's implementation of hash here:
# http://stackoverflow.com/questions/2070276/where-can-i-find-source-or-algorithm-of-pythons-hash-function
```

```
-1925086808205474835
-799031295820617361
```

In [151]:

```
# What happens when two objects return the same hash?
print hash(-799031295820617361)
```

```
-799031295820617361
```



Source: http://www.laurentluce.com/posts/python-dictionary-implementation/

Hash Table for Inverted Index

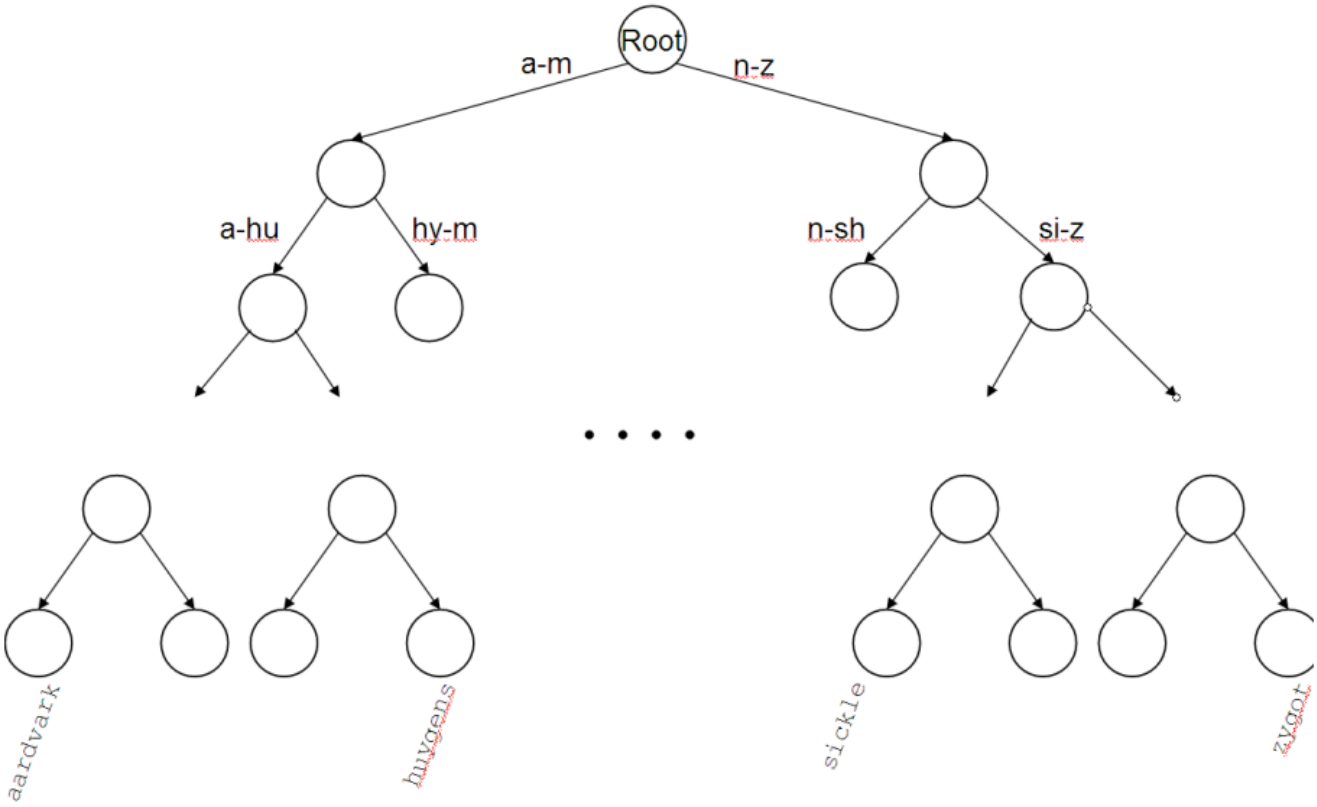
Pros:

- $O(1)$ lookup time
- Simple

Cons:

- Cannot efficiently find minor variants (e.g., zebr*)

Binary Trees



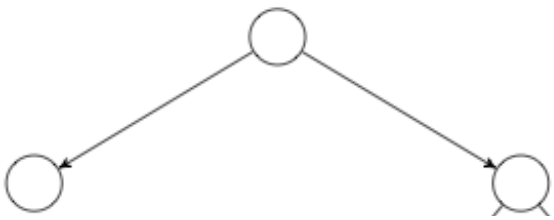
► **Figure 3.1** A binary search tree. In this example the branch at the root partitions vocabulary terms into two subtrees, those whose first letter is between a and m, and the rest.

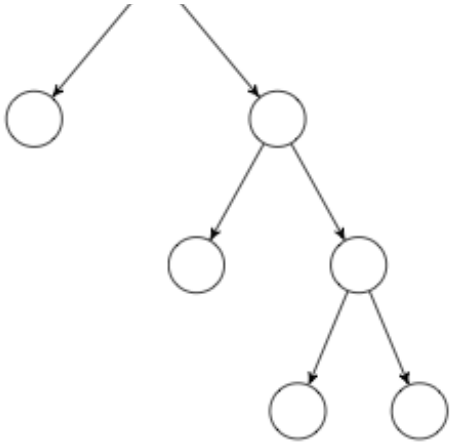
Source: MRS Ch3

Binary Trees

Search time: $O(\log n)$

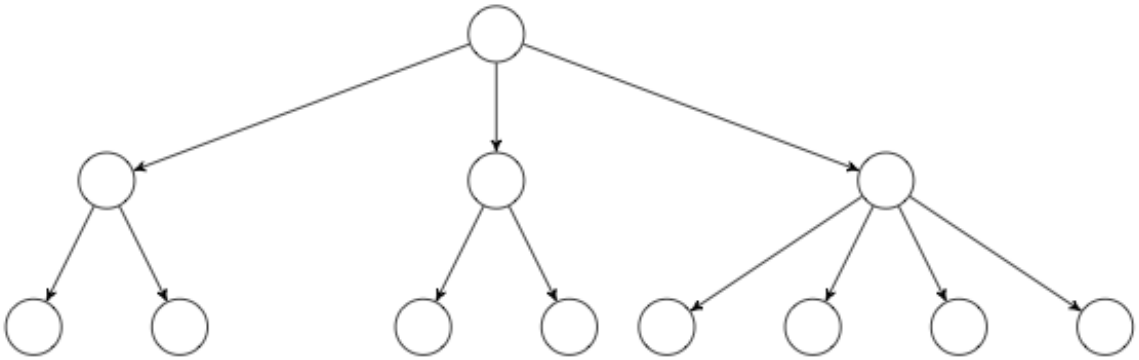
- Assumes a **balanced** tree





B-Trees

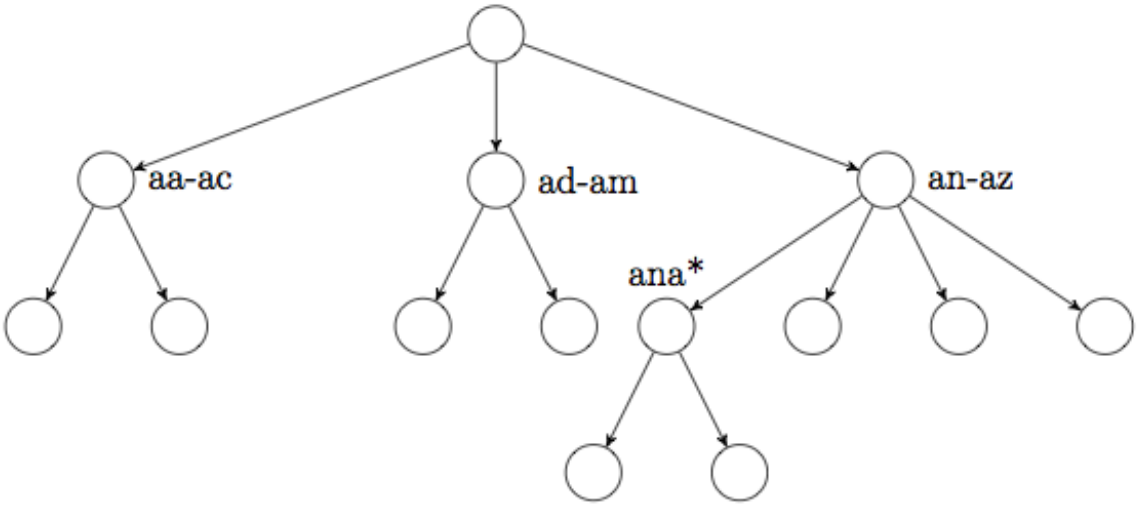
Like a binary tree, but nodes can have between *a* and *b* children, instead of just 2.



B-Tree [2,4]

Wildcard queries with B-Trees

Search for "ana*"



Spelling correction

- k-gram overlap
- Levenshtein
- Middle-ground

Levenshtein distance

How to convert string1 into string2 with the minimum number of operations?

fast $\rightarrow$ cats ?

Operations:

- *insert*: fas $\xrightarrow{insert(t)}$ fas**t**
- *delete*: fast $\xrightarrow{delete(t)}$ fas
- *substitute*: fast $\xrightarrow{substitute(t,x)}$ fas**x**

cats $\xrightarrow{substitute(c, f)}$ **f**ats $\xrightarrow{insert(s)}$ fa**s**ts $\xrightarrow{delete(s)}$ fast (3 operations)

or

cats $\xrightarrow{substitute(c, f)}$ **f**ats $\xrightarrow{substitute(t,s)}$ fa**s**s $\xrightarrow{substitute(s,t)}$ fas**t** (3 operations)

but definitely not:

cats $\xrightarrow{insert(f)}$ **f**cats $\xrightarrow{delete(c)}$ fats $\xrightarrow{delete(t)}$ fas $\xrightarrow{delete(s)}$ fa $\xrightarrow{insert(s)}$ fa**s** $\xrightarrow{insert(t)}$ fas**t** (6 operations)

In [160]:

```
# Slow, recursive Levenshtein implementation (inspired by
<http://en.wikipedia.org/wiki/Levenshtein_distance>)
def leven(s, t):
  # base case: empty strings
  if len(s) == 0:
      return len(t)  # cost of inserting all of t
  if len(t) == 0:
      return len(s)  # cost of inserting all of s

  # test if last characters match
  if s[-1] == t[-1]:
      cost = 0    # match; no cost
  else:
      cost = 1   # no match; cost of substituting one letter.

  # return minimum of (1) delete char from s, (2) delete char from t, and (3) d
elete char from both
  return min(leven(s[:-1], t) + 1,           # e.g., leven(fas, cats) + 1 (for d
eleting 't' from 'fast')
             leven(s, t[:-1]) + 1,           # e.g., leven(fast, cat) + 1 (for
deleting 's' from 'cats')
             leven(s[:-1], t[:-1]) + cost);  # e.g., leven(fas, cat) + cost (for
substituting 't' for 's')
```

In [162]:

```
print leven('fast', 'cats')
```

```
3
```

		f		a		s		t	
	0	**1**	**1**	**2**	**2**	**3**	**3**	**4**	**4**
c	**1** / **1**	*1* / *2*	*2* / *1*	**2** / **2**	3 / **2**	**3** / **3**	4 / **3**	**4** / **4**	5 / **4**
a	**2** / **2**	**2** / 3	**2** / **2**	*1* / *3*	3 / *1*	*3* / *2*	4 / *2*	4 / **3**	5 / **3**
t	**3** / **3**	**3** / 4	**3** / **3**	3 / **2**	4 / **2**	*2* / *3*	*3* / *2*	*2* / *3*	*4* / *2*
s	**4** / **4**	**4** / 5	**4** / **4**	4 / **3**	5 / **3**	**2** / 4	3 / **2**	3 / *3*	3 / *3*

▶ **Figure 3.6** Example Levenshtein distance computation. The 2×2 cell in the $[i, j]$ entry of the table shows the three numbers whose minimum yields the fourth. The cells in italics determine the edit distance in this example.

Source: MRS CH3

Spelling correction with string edit distance

Idea: Find a term in the dictionary that has minimum edit distance to query term

Tie-breaker: term that is most frequent

In [163]:

```
# Fetch a list of word counts.

from collections import defaultdict
import requests

# words: list of terms known to be spelled correctly.
word_counts = defaultdict(lambda: 1)  # Assume all words have been seen once
# Fetch list of word frequencies
words = [line.split() for line in
requests.get('http://norvig.com/ngrams/count_big.txt').text.splitlines()]
# Add to words
for word, count in words:
    word_counts[word] += int(count)
print 'read', len(words), 'words'
print 'count(a)=', word_counts['a']
print 'count(apple)=', word_counts['apple']
print 'count(ajshdlfkjahdlkjh)=', word_counts['ajshdlfkjahdlkjh']
```

```
read 29136 words
count(a)= 21161
```

```
count(apple)= 12
count(ajshdlfkjahdlkjh)= 1
```

In [164]:

```
# Find the element of words that has minimum edit distance to word
# Return word and the distance.
def min_leven(words, word):
    distances = [(w, leven(w, word)) for w in words]
    return min(distances, key=lambda x: x[1])
```

In [168]:

```
print min_leven(['apple', 'banana', 'chair'], 'bannana')
```

```
('banana', 1)
```

In [143]:

```
# Too slow!
# print min_leven(word_counts.keys(), 'accross')
```

Faster but less precise

(See http://norvig.com/spell-correct.html)

70-80% of misspellings are have edit distance of 1

Idea: Efficiently generate all terms that are edit distance of 1 from query term.

In [171]:

```
# Return all single edits to word
alphabet = 'abcdefghijklmnopqrstuvwxyz'
def edits(word):
   splits     = [(word[:i], word[i:]) for i in range(len(word) + 1)]
   deletes    = [a + b[1:] for a, b in splits if b]                          # cat->
ca
   transposes = [a + b[1] + b[0] + b[2:] for a, b in splits if len(b)>1]  # cat -
> act
   replaces   = [a + c + b[1:] for a, b in splits for c in alphabet if b] # cat -
> car
   inserts    = [a + c + b     for a, b in splits for c in alphabet]      # cat -
> cats
   return set(deletes + transposes + replaces + inserts)                   #
union all edits

print len(edits('cat')), 'edits for cat:', edits('cat')
```

```
182 edits for cat: set(['caqt', 'ucat', 'cdt', 'ctat', 'ciat', 'vcat', 'cvat',
'ycat', 'caht', 'cut', 'jat', 'caty', 'clt', 'hat', 'cyat', 'capt', 'icat', 'zc
at', 'fat', 'dat', 'cet', 'caot', 'catz', 'hcat', 'bat', 'crt', 'cayt', 'cakt',
'clat', 'cmt', 'cvt', 'ceat', 'cwat', 'cjat', 'cnat', 'acat', 'cft', 'cabt', 'c
nt', 'cajt', 'aat', 'cwt', 'cast', 'czat', 'csat', 'cqat', 'cit', 'cart', 'jcat
', 'cfat', 'cazt', 'pcat', 'catd', 'caat', 'cgt', 'ctt', 'cati', 'cait', 'cot',
'cawt', 'xcat', 'cta', 'act', 'ncat', 'cxt', 'ckat', 'calt', 'ca', 'dcat', 'cad
t', 'zat', 'cato', 'ct', 'crat', 'cata', 'catb', 'catc', 'tcat', 'cate',
'catf', 'catg', 'cath', 'yat', 'catj', 'catk', 'xat', 'catm', 'catn', 'catl', '
```

```
catp', 'ocat', 'catr', 'cats', 'cht', 'catu', 'catv', 'catw', 'catx', 'iat', 'b
cat', 'wat', 'catq', 'vat', 'cqt', 'cact', 'cyt', 'rcat', 'gat', 'cant', 'cgat'
, 'mcat', 'eat', 'kcat', 'caz', 'cay', 'cax', 'cas', 'car', 'caq', 'cap', 'caw'
, 'cav', 'cau', 'cat', 'cak', 'caj', 'cai', 'cah', 'cao', 'can', 'cam', 'cal',
'cac', 'cab', 'caa', 'cag', 'caf', 'cae', 'cad', 'tat', 'chat', 'fcat', 'caft',
'lcat', 'uat', 'czt', 'rat', 'at', 'cbt', 'catt', 'scat', 'sat', 'qat', 'qcat',
'pat', 'wcat', 'cuat', 'oat', 'nat', 'cst', 'cavt', 'cjt', 'mat', 'cxat', 'caet
', 'cmat', 'ccat', 'cagt', 'cpat', 'kat', 'lat', 'gcat', 'caxt', 'cdat', 'coat'
, 'cct', 'camt', 'ckt', 'caut', 'cpt', 'cbat', 'ecat'])
```

How many edits? *n* deletions, *n-1* transpositions, *26n* substitutions, and *26(n+1)* insertions, for a total of *54n+25.*

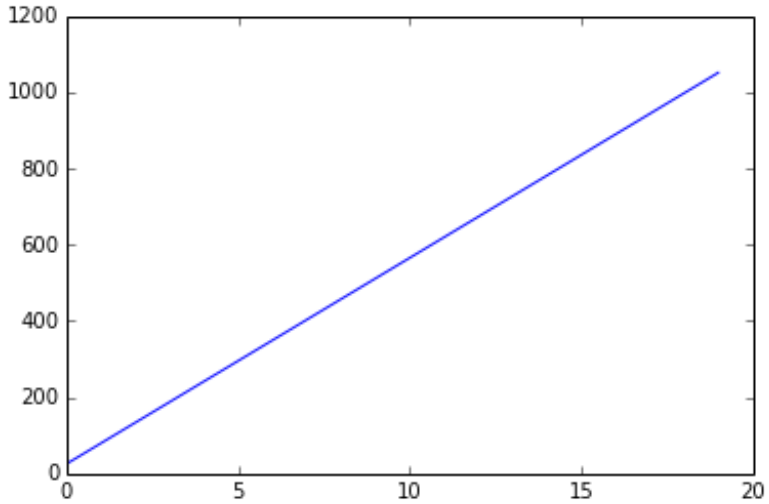
In [172]:

```
%pylab inline
plot([54 * x + 25 for x in range(20)])
```

```
Populating the interactive namespace from numpy and matplotlib
```

Out[172]:

```
[<matplotlib.lines.Line2D at 0x111ce59d0>]
```



In [174]:

```
# Return the subset of words that is in word_counts.
def known(words):
    return set(w for w in words if w in word_counts)

print known(['apple', 'zzzzasdfasdfz'])
```

```
set(['apple'])
```

In [175]:

```
def correct(word):
    candidates = known([word]) or known(edits(word)) or [word] # 'or' returns w
hichever is the first non-empty value
    return max(candidates, key=word_counts.get)
```

In [177]:

```
print correct('apple')   # apple is in word_counts: known([word])
```

```
print correct('accross') # accross is not in word_counts, but across is: known(edits(word))
print correct('zebraa')  # zebra is not in word_counts: [word]
```

```
apple
across
zebraa
```

How to use spelling correction?

- Make suggestions ("Did you mean?")
- Add corrected terms to query
  - only if query term is not in dictionary
  - only if number of matches < N

CS 429: Information Retrieval

Lecture 5: Scalable Indexing

Dr. Aron Culotta

Illinois Institute of Technology

Spring 2014

Last time:

- Efficient retrieval of postings lists
- Wildcard queries
- Spelling correction

Today:

- How do we build an index that does not fit into memory?

Building an index

- Up to now, we've assumed everything fits in memory.
- We'll discuss three ways to scale
  1. Block sort-based indexing (**BSBI**)
  2. Single-pass in-memory indexing (**SPIMI**)
  3. MapReduce

How long does it take to read 100MB from disk?

- **Seek time** (to locate data)
- **Transfer rate** (to copy from disk into memory)
- Contiguous or non-contiguous?

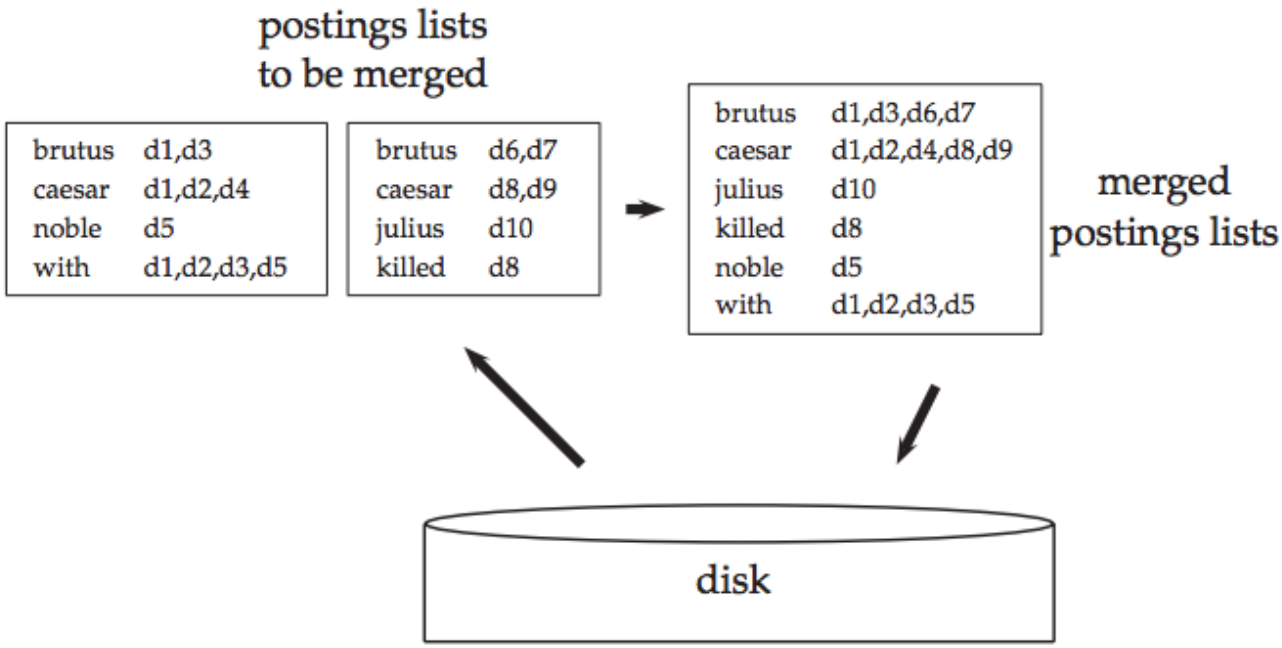
Block sort-based indexing (BSBI)

Assume a single machine.

1. Split documents into **blocks**
2. For each block:
   A. Parse each block into (word_id, doc_id) pairs
   B. Sort pairs and create separate postings lists for each block.

C. Write postings lists to disk

3. Merge the postings lists file for each block



► **Figure 4.3** Merging in blocked sort-based indexing. Two blocks ("postings lists to be merged") are loaded from disk into memory, merged in memory ("merged postings lists") and written back to disk. We show terms instead of termIDs for better readability.

(source: MRS)

In [1]:

```
# BSBI: Create one postings list per block of documents.
from collections import defaultdict
from itertools import groupby

block1 = [(0, ["the", "dog", "jumped"]),
          (1, ["the", "cat", "jumped"])]
block2 = [(2, ["a", "dog", "ran"]),
          (3, ["the", "zebra", "jumped"])]
blocks = [block1, block2]

vocab = defaultdict(lambda: len(vocab))
for block_id, block in enumerate(blocks):
    # A. Collect all individual postings: (word_id, doc_id) pairs.
    postings = []
    for doc_id, doc in block:
        for word in doc:
            postings.append((vocab[word], doc_id))
    print 'block', block_id, 'postings=', postings
    print 'vocab=', vocab.items()

    # B. Sort postings and create postings lists.
    postings = sorted(postings, key=lambda x: x[0])
    print 'block', block_id, 'sorted postings=', postings

    # Group postings for same term together.
    postings = groupby(postings, key=lambda x:x[0])
```

```
    postings = [(word_id, [g[1] for g in group]) for word_id, group in postings]
    print 'block', block_id, 'grouped postings=', postings

    # C. Write to disk
    f = open('bsbi_block' + str(block_id) + '.txt', 'wt')
    f.write('\n'.join(['%s' % str(p) for p in postings]))
    f.close()
    print

# Then, merge blocks in linear time.
```

```
block 0 postings= [(0, 0), (1, 0), (2, 0), (0, 1), (3, 1), (2, 1)]
vocab= [('jumped', 2), ('the', 0), ('dog', 1), ('cat', 3)]
block 0 sorted postings= [(0, 0), (0, 1), (1, 0), (2, 0), (2, 1), (3, 1)]
block 0 grouped postings= [(0, [0, 1]), (1, [0]), (2, [0, 1]), (3, [1])]

block 1 postings= [(4, 2), (1, 2), (5, 2), (0, 3), (6, 3), (2, 3)]
vocab= [('a', 4), ('ran', 5), ('jumped', 2), ('dog', 1), ('cat', 3), ('zebra',
6), ('the', 0)]
block 1 sorted postings= [(0, 3), (1, 2), (2, 3), (4, 2), (5, 2), (6, 3)]
block 1 grouped postings= [(0, [3]), (1, [2]), (2, [3]), (4, [2]), (5, [2]), (6
, [3])]
```

Merging posting list blocks

Since each block is sorted by term, can do a single linear pass through each block.

- Open all postings files simulataneously.
- Read small buffer from each.
- Equivalent to a union of postings lists for same term.

BSBI

Space requirements?

- Number of tokens in each block (T)
- Number of unique terms in vocabulary (V)

Time requirements?

- Sorting (word_id, doc_id) pairs in each block.
- $O(T log T)$

Single-pass in-memory indexing (SPIMI)

- separate dictionary for each block
- create postings lists on the fly, rather than collecting all postings then sorting.
- sort postings lists, rather than individual postings

In [2]:

```
# SPIMI: Create one postings list per block of documents.
from collections import defaultdict
```

```
from itertools import groupby

block1 = [(0, ["the", "dog", "jumped"]),
          (1, ["the", "cat", "jumped"])]
block2 = [(2, ["a", "dog", "ran"]),
          (3, ["the", "zebra", "jumped"])]
blocks = [block1, block2]

for block_id, block in enumerate(blocks):
    # Note that there is a new vocab for each block!
    vocab = defaultdict(lambda: len(vocab))  # maps from term -> term_id
    index = defaultdict(lambda: [])          # from term_id -> postings list

    for doc_id, doc in block:
        # append doc_id to the postings list of each term
        for word in doc:
            index[vocab[word]].append(doc_id)


    # B. Sort terms
    sorted_terms = sorted(vocab.keys())
    print 'Block', block_id, [t + ' ' + str(index[vocab[t]]) for t in
sorted_terms]

    # C. Write to disk
    f = open('spimi_block' + str(block_id) + '.txt', 'wt')
    f.write('\n'.join(['%d, %s' % (vocab[t], str(index[vocab[t]])) for t in sort
ed_terms]))
    f.close()
    print

# Then, merge blocks in linear time.
```

```
Block 0 ['cat [1]', 'dog [0]', 'jumped [0, 1]', 'the [0, 1]']

Block 1 ['a [2]', 'dog [2]', 'jumped [3]', 'ran [2]', 'the [3]', 'zebra [3]']
```

SPIMI

Space requirements?

- Number of tokens in each block (T)
- Number of unique terms in vocabulary *in each block* ($V_b << V$)

Time requirements?

- Sorting unique terms in each block
- $O(V log V)$
  - (compare to $O(T log T)$ for BSBI

MapReduce

- What if we had 100K servers?

- **MapReduce:**

    - A distributed programming framework
    - Breaks large data into smaller data, called **splits**
- Two phases:
    - **Map:**
        - Input: one split
        - Output: (key, value) pairs
    - **Reduce:**
        - Input: (key, list of mapped values)
        - Output: list of output values

MapReduce Counting Example

```
map(key, value):
  for each word in value:
    output word, 1

reduce(key, values):
  output key, sum(values)
```

Framework takes care of grouping keys together to call `reduce` appropriately.

MapReduce Counting Example

- Split 1: "Twinkle, twinkle little star"
- Split 2: "Little by little"

Map

"Twinkle, twinkle little star" $\rightarrow$ **Mapper 1** $\rightarrow$ (twinkle, 1), (twinkle, 1), (little, 1), (star, 1)

"Little by little" $\rightarrow$ **Mapper 2** $\rightarrow$ (little, 1), (by, 1), (little, 1)

Reduce

(twinkle, 1), (twinkle, 1) $\rightarrow$ **Reducer 1** $\rightarrow$ (twinkle, 2)

(little, 1), (little, 1), (little, 1) $\rightarrow$ **Reducer 2** $\rightarrow$ (little, 3)

...

Indexing with MapReduce

- **Map**: Read a document and output (term, doc_id) pairs.
- **Reduce**: Read a list of doc_ids for a term and output a postings list.

"Twinkle, twinkle little star" $\rightarrow$ **Mapper 1** $\rightarrow$ (twinkle, 0), (little, 0), (star, 0)

"Little by little" $\rightarrow$ **Mapper 2** $\rightarrow$ (little, 1), (by, 1)

Reduce

(little, 0), (little, 1) $\rightarrow$ **Reducer 1** $\rightarrow$ (little, [0, 1])

...

In [3]:

```
import re
from mrjob.job import MRJob  # install with `pip install mrjob`

class MRIndexer(MRJob):

    def mapper(self, _, line):
        # Emit word, doc_id pairs from lines that look like:
        # doc_id [document tokens]
        words = re.findall('\w+', line.lower())
        doc_id = int(words[0])
        for word in set(words[1:]):
            yield word, doc_id

    def reducer(self, key, values):
        # key is a term, values is an (unsorted) list of doc_ids
        yield key, sorted(values)

# Run python mr.py to execute this example.
```

CS 429: Information Retrieval

Lecture 6: Index Compression

Dr. Aron Culotta

Illinois Institute of Technology

Spring 2014

Last time:

- How do we build an index that does not fit into memory?

Today:

- How do we compress the contents of an index?

Why compress an inverted index?

- Save disk space
- Fit as much in memory as possible (caching)
- Faster transfer from disk to memory

What will we compress?

- Vocabulary
- Postings lists

To help compression, we need to know

- How large should we expect the vocabulary to be?
  - helpful to compress vocabulary
- How are terms distributed throughout documents?
  - helpful to compress postings lists

Will the vocabulary get *that* big?

- T = number of tokens
- V = number of terms

Can we estimate V as a function of T?

$V = f(T)$

In [1]:

```
# Let's read in some documents.
# This is a dataset of 11K newsgroups posts:
http://qwone.com/~jason/20Newsgroups/
from sklearn.datasets import fetch_20newsgroups
docs = fetch_20newsgroups(subset='train', remove=('headers', 'footers',
'quotes')).data
print 'read %d docs' % len(docs)
```

```
read 11314 docs
```

In [2]:

```
# Let's look at a couple documents.
print docs[0]
```

```
I was wondering if anyone out there could enlighten me on this car I saw
the other day. It was a 2-door sports car, looked to be from the late 60s/
early 70s. It was called a Bricklin. The doors were really small. In addition,
the front bumper was separate from the rest of the body. This is
all I know. If anyone can tellme a model name, engine specs, years
of production, where this car is made, history, or whatever info you
have on this funky looking car, please e-mail.
```

In [3]:

```
print docs[100]
```

```
1.  Software publishing SuperBase 4 windows v.1.3           --->$80

2.  OCR System ReadRight v.3.1 for Windows                  --->$65

3.  OCR System ReadRight  v.2.01 for DOS                    --->$65

4.  Unregistered Zortech 32 bit C++ Compiler v.3.1          --->$ 250
     with Multiscope windows Debugger,
     WhiteWater Resource Toolkit, Library Source Code

5.  Glockenspiel/ImageSoft Commonview 2 Windows
     Applications Framework for Borland C++                 --->$70

6.  Spontaneous Assembly Library With Source Code           --->$50

7.  Microsoft Macro Assembly 6.0                            --->$50

8.  Microsoft Windows v.3.1 SDK Documentation               --->$125

9.  Microsoft FoxPro V.2.0                                  --->$75

10.  WordPerfect 5.0 Developer's Toolkit                    --->$20

11.  Kedwell Software DataBoss v.3.5 C Code Generator       --->$100

12.  Kedwell InstallBoss v.2.0 Installation Generator       --->$35
```

```
13.  Liant Software C++/Views v.2.1
       Windows Application Framework with Source Code       --->$195

14.  IBM OS/2 2.0 & Developer's Toolkit                     --->$95

15.  CBTree DOS/Windows Library with Source Code            --->$120

16.  Symantec TimeLine for Windows                          --->$90

17.  TimeSlip TimeSheet Professional for Windows            --->$30
```

Next, let's count the number of terms and tokens in this dataset.

In [4]:

```
from collections import defaultdict
import re

# Count the number of terms and tokens in a list of documents.
# return
#  1. terms: a dict from term to number of documents it appears in
#  2. n_tokens: the number of tokens in docs
def count_terms_and_toks(docs):
    terms = defaultdict(lambda: 0)  # Map from term to count.
    n_tokens = 0
    for d in docs:
        d_terms = set()  # increment each term once per document.
        for tok in re.findall('[\w]+',d.lower()):
            d_terms.add(tok)
            n_tokens += 1
        for d_term in d_terms:
            terms[d_term] += 1

    return terms, n_tokens
```

In [5]:

```
terms, n_tokens = count_terms_and_toks(docs)
print 'found %d tokens and %d terms' % (n_tokens, len(terms))
```

```
found 2407154 tokens and 101660 terms
```

How does the number of terms vary with the number of tokens?

In [6]:

```
# Compute T/V for different subsets of the documents.
T = []
V = []
for n_docs in [10, 100, 200, 500, 1000, 2000, 3000, 4000, 5000, 10000]:
    terms, n_tokens = count_terms_and_toks(docs[:n_docs])
    print 'found %d tokens and %d terms in %d docs' % (n_tokens, len(terms), n_d
ocs)
    T.append(n_tokens)
    V.append(len(terms))
```

```
found 1270 tokens and 545 terms in 10 docs
```

```
found 20449 tokens and 4941 terms in 100 docs
found 50303 tokens and 8848 terms in 200 docs
found 105452 tokens and 17725 terms in 500 docs
found 209464 tokens and 24469 terms in 1000 docs
found 435254 tokens and 39713 terms in 2000 docs
found 619554 tokens and 46003 terms in 3000 docs
found 858034 tokens and 54854 terms in 4000 docs
found 1123558 tokens and 63355 terms in 5000 docs
found 2155062 tokens and 92689 terms in 10000 docs
```

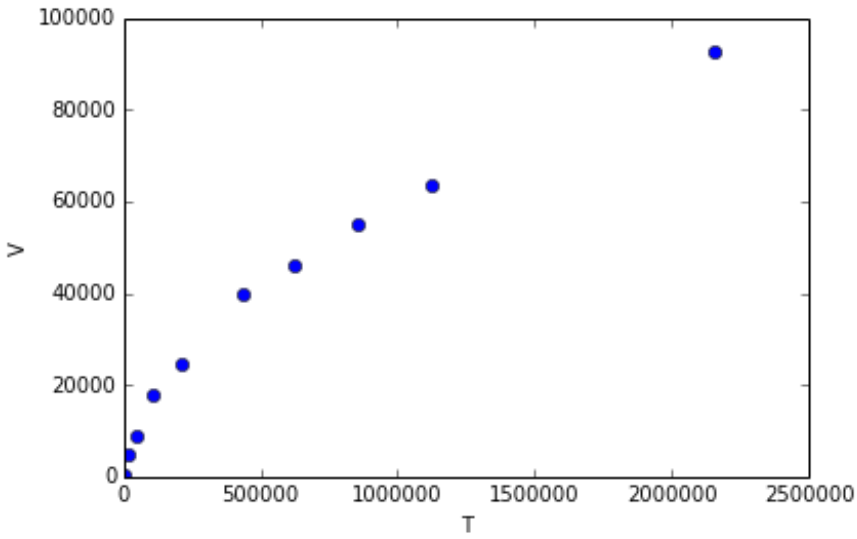
In [7]:

```
# Let's plot the results.
% pylab inline
xlabel('T')
ylabel('V')
plot(T, V, 'bo')
```

```
Populating the interactive namespace from numpy and matplotlib
```

Out[7]:

```
[<matplotlib.lines.Line2D at 0x110a45810>]
```



Is this linear, polynomial, something else?

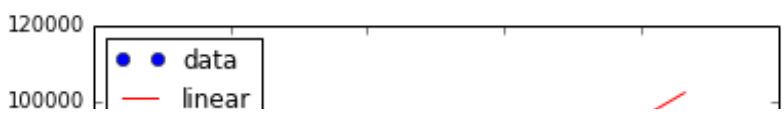
In [8]:

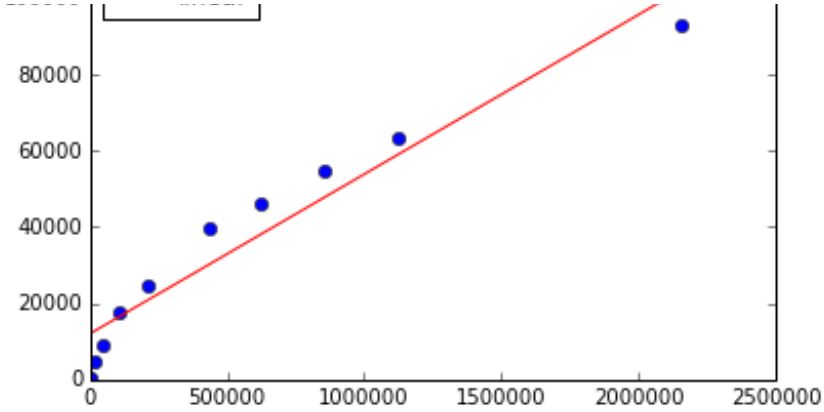
```
# Let's try a linear fit
import numpy as np
linear = np.polyfit(T, V, 1)  # fit slope and intercept
print 'linear fit=%.2f*x + %.2f' % (linear[0], linear[1])
plot(T, V, 'bo', label='data')  # bo = blue circle
plot(T, np.polyval(linear, T), 'r-', label='linear')  # r- = red solid line
legend(loc='best')
```

```
linear fit=0.04*x + 11949.85
```

Out[8]:

```
<matplotlib.legend.Legend at 0x110a7b7d0>
```





Heaps' Law

An observed relation between V and T:

$V = k T^b$

for constants k (typically $30 < k < 100$) and b (typically $b \approx 0.5$)

In [9]:

```
# How do we set k and b in Heaps' Law?
# Minimize mean squared error.
from scipy.optimize import curve_fit
help(curve_fit)
```

```
Help on function curve_fit in module scipy.optimize.minpack:

curve_fit(f, xdata, ydata, p0=None, sigma=None, **kw)
    Use non-linear least squares to fit a function, f, to data.

    Assumes ``ydata = f(xdata, *params) + eps``

    Parameters
    ----------
    f : callable
        The model function, f(x, ...).  It must take the independent
        variable as the first argument and the parameters to fit as
        separate remaining arguments.
    xdata : An N-length sequence or an (k,N)-shaped array
        for functions with k predictors.
        The independent variable where the data is measured.
    ydata : N-length sequence
        The dependent data --- nominally f(xdata, ...)
    p0 : None, scalar, or M-length sequence
        Initial guess for the parameters.  If None, then the initial
        values will all be 1 (if the number of parameters for the function
        can be determined using introspection, otherwise a ValueError
        is raised).
    sigma : None or N-length sequence
        If not None, it represents the standard-deviation of ydata.
        This vector, if given, will be used as weights in the
        least-squares problem.

    Returns
    -------
    popt : array
```

```
        Optimal values for the parameters so that the sum of the squared error
        of ``f(xdata, *popt) - ydata`` is minimized
    pcov : 2d array
        The estimated covariance of popt.  The diagonals provide the variance
        of the parameter estimate.

    See Also
    --------
    leastsq

    Notes
    -----
    The algorithm uses the Levenberg-Marquardt algorithm through `leastsq`.
    Additional keyword arguments are passed directly to that algorithm.

    Examples
    --------
    >>> import numpy as np
    >>> from scipy.optimize import curve_fit
    >>> def func(x, a, b, c):
    ...     return a*np.exp(-b*x) + c

    >>> x = np.linspace(0,4,50)
    >>> y = func(x, 2.5, 1.3, 0.5)
    >>> yn = y + 0.2*np.random.normal(size=len(x))

    >>> popt, pcov = curve_fit(func, x, yn)
```

In [10]:

```
# V = k * T^b
def heaps(T, k, b):
    return k*(T**b)
```

In [11]:

```
# Fit k and b
heap_parms,covar = curve_fit(heaps, T, V)
print 'Heaps fit is %.2f*T^%.2f' % (heap_parms[0], heap_parms[1])
```

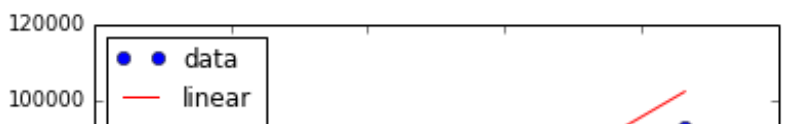
```
Heaps fit is 23.49*T^0.57
```

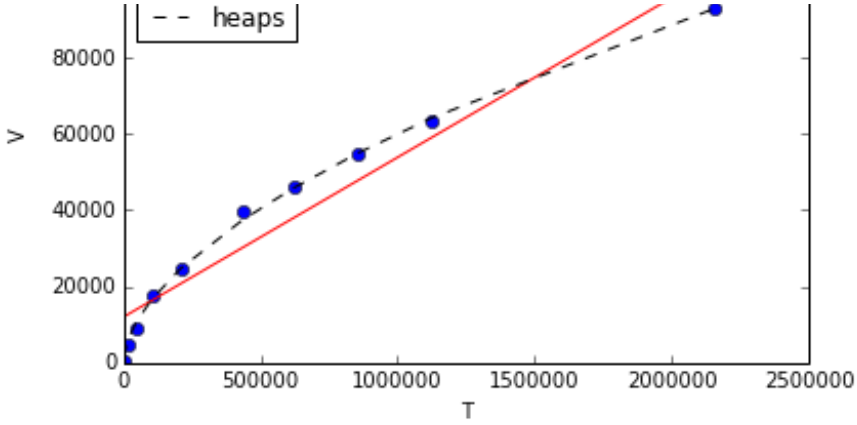
In [12]:

```
# Compare linear fit and Heaps fit
plot(T, V, 'bo', label='data')
plot(T, np.polyval(linear, T), 'r-', label='linear')
plot(T, heaps(T, *heap_parms), 'k--', label='heaps')  # k-- = black dashed line
xlabel('T')
ylabel('V')
legend(loc='best')
```

Out[12]:

```
<matplotlib.legend.Legend at 0x110b1b150>
```





In [13]:

```
print 'Heaps predicts %d terms for %d tokens, truth is %d.' % (heaps(T[-1],
*heap_parms), T[-1], V[-1])
```

```
Heaps predicts 92835 terms for 2155062 tokens, truth is 92689.
```

How are terms distributed across documents?

- How many times does the most frequent term occur? The ith most frequent term?

In [14]:

```
# Let's plot and see.
# Recall that terms is a dict from term to document frequency
print '"the" occurs in %d documents; "honda" occurs in %d documents' % (terms['
the'], terms['honda'])
```

```
"the" occurs in 8373 documents; "honda" occurs in 61 documents
```

In [15]:

```
# Sort frequency values in descending order
freqs = sorted(terms.values(), reverse=True)
print 'top 10 frequencies are', freqs[:10]
```

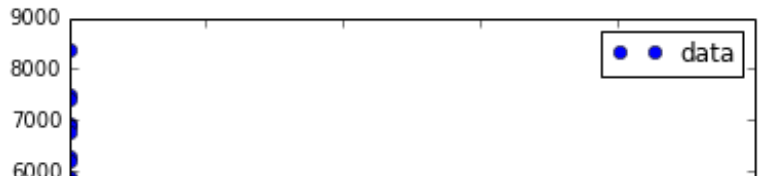
```
top 10 frequencies are [8373, 7479, 7396, 6912, 6870, 6781, 6287, 6206, 5884, 5
814]
```

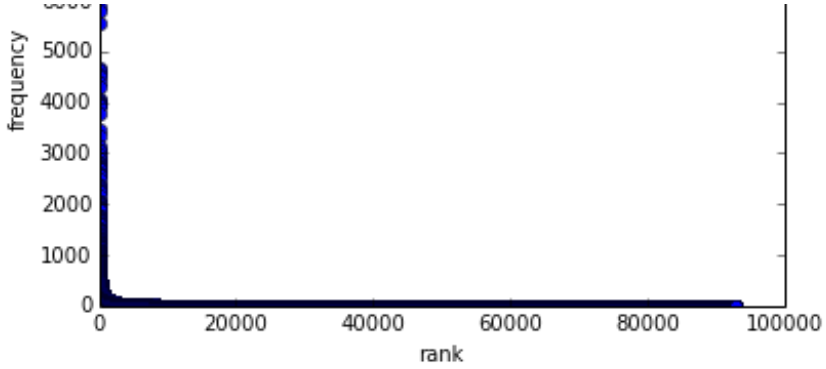
In [16]:

```
ranks = range(1, len(freqs)+1)
plot(ranks, freqs, 'bo', label='data')
legend(loc='best')
xlabel('rank')
ylabel('frequency')
```

Out[16]:

```
<matplotlib.text.Text at 0x110b23d50>
```





In [17]:

```
print '%d/%d terms occur in only one document.' % (len([x for x in freqs if x == 1]), len(freqs))
```

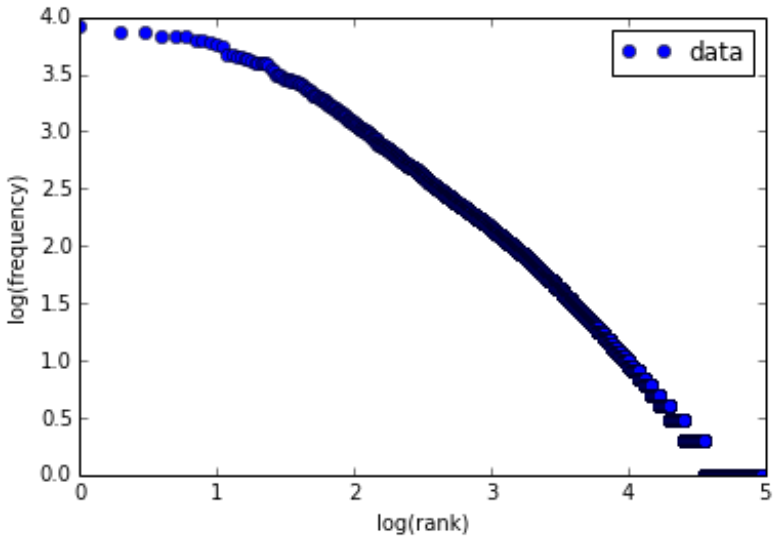
```
56258/92689 terms occur in only one document.
```

In [18]:

```
# That was ugly. The values decrease too rapidly. Let's try a log-log plot.
l_ranks = np.log10(ranks)
l_freqs = np.log10(freqs)
plot(l_ranks, l_freqs, 'bo', label='data')
legend(loc='best')
xlabel('log(rank)')
ylabel('log(frequency)')
```

Out[18]:

```
<matplotlib.text.Text at 0x110a95090>
```



Zipf's Law

Another empirical law that states that the frequency of a term is inversely proportional to its rank.

Let f_i be the frequency of the ith most common term.

$ f_i \propto \frac{1}{i} $

equivalently

$ f_i = k$ $i^b $ for constant k and $b=-1$

(c.f. Heap's law: $V = kT^b$)

In [19]:

```
# Define the Zipf function and fit the k parameter.
def zipfs(i, k):
    return k / i
zipf_parms,covar = curve_fit(zipfs, ranks, freqs)
print 'Zipf fit is %.2f*T^-1' % zipf_parms[0]
```

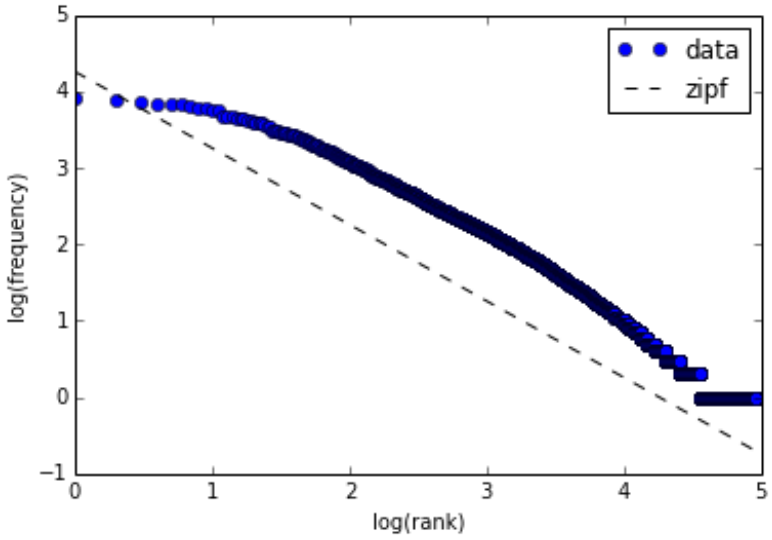
```
Zipf fit is 18138.34*T^-1
```

In [20]:

```
plot(l_ranks, l_freqs, 'bo', label='data')
xlabel('log(rank)')
ylabel('log(frequency)')
plot(l_ranks, log10(zipfs(ranks, *zipf_parms)), 'k--', label='zipf')  # k-- = bl
ack dashed line
legend(loc='best')
```

Out[20]:

```
<matplotlib.legend.Legend at 0x10b3e1b10>
```



In [33]:

```
slope = zipf_parms[0]
print slope / 92689 - slope / 92688

print len([x for x in range(1,92835) if slope / x < 1.5 and x > 1])
```

```
-2.11128021727e-06
80742
```

Dictionary compression

$ \begin{eqnarray*} \mathbf{dog} & \rightarrow & [1, 6, 20] \\ \mathbf{cat} & \rightarrow & [7, 16, 32] \\ \end{eqnarray*} $

1. Fixed-width storage
2. One big string
3. Blocked storage
4. Front encoding

Fixed-width storage

	term	document frequency	pointer to postings list
	a	656,265	⟶
	aachen	65	⟶
	...	...	...
	zulu	221	⟶
space needed:	20 bytes	4 bytes	4 bytes

► **Figure 5.3** Storing the dictionary as an array of fixed-width entries.

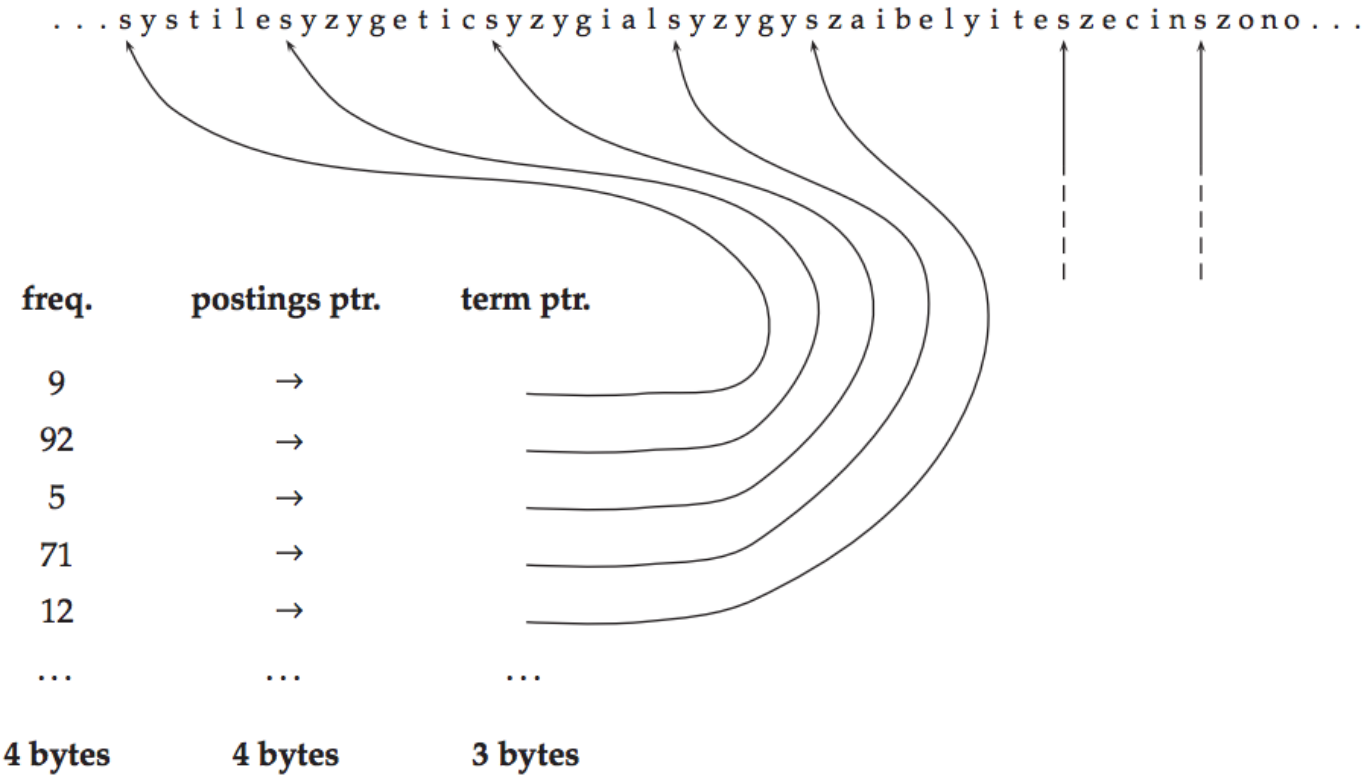
MRS

Space = $(20 + 4 + 4)V = 28V$ bytes

Why is 20 bytes / term wasteful?

Average term $\approx$ 8 bytes

One big string



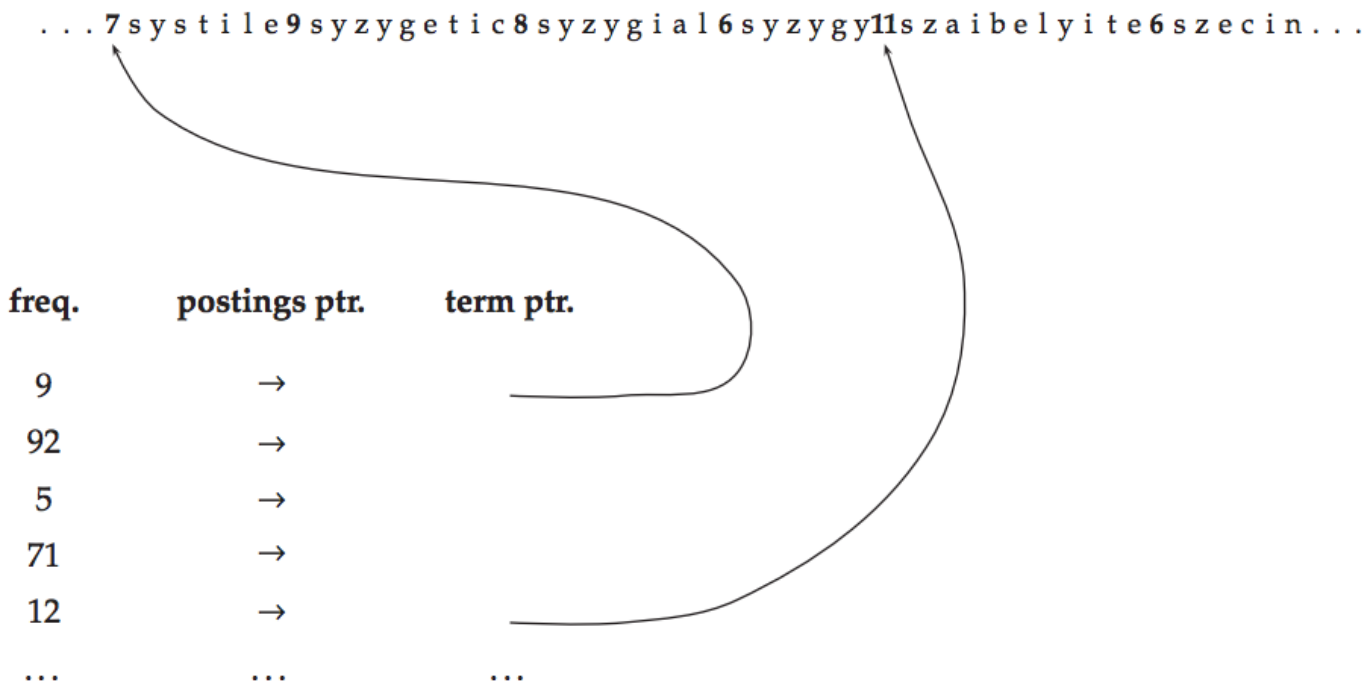
MRS

Assuming average term is 8 bytes,

Space = $(8 + 4 + 4 + 3)V = 19V$ bytes (reduction from $28V$)

Blocked storage

Reduce number of term pointers:



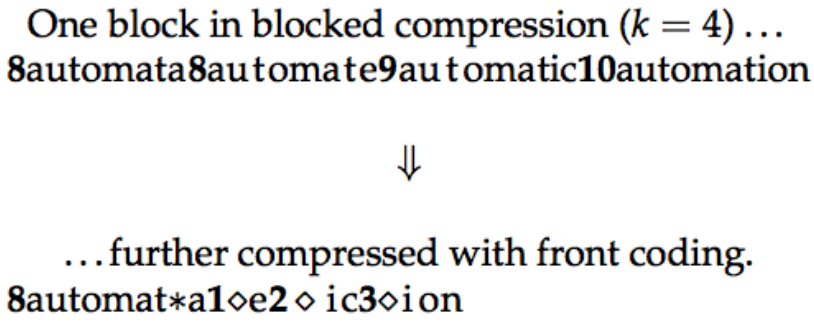
MRS

- Assume k blocks, we store only k term pointers (instead of V)
- But, we also need to add one byte per term for offsets (term length)

Space = $(8 + 4 + 4)V + k + V = 17V + k$ bytes (reduction from $19V$)

Why not use $k=1$?

Front encoding



► **Figure 5.7** Front coding. A sequence of terms with identical prefix ("automat") is encoded by marking the end of the prefix with ∗ and replacing it with ⋄ in subsequent terms. As before, the first byte of each entry encodes the number of characters.

MRS

Savings depends on chosen prefixes.

Dictionary compression results

► **Table 5.2** Dictionary compression for Reuters-RCV1.

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
∼, with blocking, $k = 4$	7.1
∼, with blocking & front coding	5.9

MRS

Note that 28V / 19V $\approx$ 11.2 / 7.6

Compression postings lists

$ \begin{eqnarray*} dog & \rightarrow & \mathbf{[1, 6, 20]} \\ cat & \rightarrow & \mathbf{[7, 16, 32]} \\ \end{eqnarray*} $

How many bits to represent a doc ID?

$log_2(D)$ for D documents in dataset.

So, postings list for a term that appears in n documents requires $n log_2(D)$ bits.

Storing gaps

Idea: Store only gaps between doc IDs. Hope that the range of gaps is less than the number of documents. E.g.

$ dog \rightarrow [101, 102, 104, 107] $

becomes

$ dog \rightarrow [101, 1, 2, 3] $

Instead of $log_2(D)$, each ID is stored in $log_2(M)$, where M is maximum gap.

What about rare words?

$ defenestrate \rightarrow [100, 99999100] $

becomes

$ defenestrate \rightarrow [100, 99999000] $

Variable byte encoding

Idea: Use more bytes for larger numbers.

E.g.,

- doc ID $4 \rightarrow 100$ (3 bits)
- doc ID $4000 \rightarrow 111110100000$ (12 bits)

What's the problem with this?

4 followed by 400 becomes: 100111110100000

Where does one ID end and the next begin?

Idea:

- For each byte, let the final 7 bits be a subsequence of an ID;
- Let the first bit be 1 if this is the final byte for this ID

$4 \rightarrow $ 10000100 (1 = final byte, 0000100 = ID 4)

$4000 \rightarrow$ 00011111 10100000 $\rightarrow$ _0011111_0100000 $\rightarrow$ 00111110100000

► **Table 5.4** VB encoding. Gaps are encoded using an integral number of bytes. The first bit, the continuation bit, of each byte indicates whether the code ends with this byte (1) or not (0).

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

MRS

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
~, with blocking, $k = 4$	7.1
~, with blocking & front coding	5.9
collection (text, xml markup etc)	3600.0
collection (text)	960.0

term incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0
postings, γ encoded	101.0

MRS

How many bytes to store n postings lists?

Need to know how many doc ids in each postings list.

Recall Zipf:

$ f_i = \frac{k}{i} $ for constant k

so, most frequent term has k doc ids, next most frequent has $\frac{k}{2}$, next has $\frac{k}{3}$, etc.

Have to make some assumption about gap sizes: e.g., uniform $ [100, 200, 300, 400, ...] $

$\begin{eqnarray*} term_1 & = & [\frac{D}{k}, 2\frac{D}{k}, 3\frac{D}{k}, \ldots, k\frac{D}{k}]\\ term_2 & = & [\frac{2D}{k}, 2\frac{2D}{k}, 3\frac{2D}{k}, \ldots, k\frac{2D}{k}]\\ \vdots\\ term_i & = & [\frac{iD}{k}, 2\frac{iD}{k}, 3\frac{iD}{k}, \ldots, k\frac{iD}{k}]\\ \end{eqnarray*} $

In terms of D, k, what is

- average gap size?
- total number of gaps?

CS 429: Information Retrieval

Lecture 7: Term weighting and the Vector Space Model

Dr. Aron Culotta

Illinois Institute of Technology

Spring 2014

Last time:

- How do we compress the contents of an index?

Today:

- From boolean search to ranking

Assignment 0 notes

- Why is this inefficient?

In [1]:

```
def intersect(list1, list2):
    list(set(list1) & set(list2))
```

- `create_index`: no duplicate doc IDs

```
>>> create_index([['a', 'a']])
{'a': [0, 0]}
```

- counter-example to optimizing query order by term frequency
  - a=[0, 1, 2], b=[3, 4, 5, 6], c=[0, 2]
  - **shortest first** - (a AND c) AND b. 3 comparisons for (a AND c), yielding [0, 2]; 2 comparisons to AND with b. **Total=** 5 comparisons.
  - **longest first** - (a AND b) AND c. 3 comparisons for (a AND b), yielding [ ]. No need to compare with c. **Total=** 3 comparisons.

- negation does not always come last.
  - a = [0, 1], b = [0, 1], c=[0, 1, 2]
  - a AND (b AND NOT c) requires fewer comparisons than (a AND b) AND NOT c

Why is boolean search bad?

- Prone to user error
- Have to think of all possible ways of expressing information need
- Too many matching results

Why is boolean search good?

- Users get exactly what they ask for
- Good for integrating with other software

Problems with boolean search

- Too few results: "error: null ptr exception in line 2341" $\rightarrow$ 0 results
- Too many results: "error: null ptr exception" $\rightarrow$ 1M results
- User must carefully refine the query

Ranked retrieval

- Order search results by **relevance** to query.
- Too many results is not a problem: user only looks at top 10-20.

Ranked retrieval

Problem: Given a query q and a set of documents D, compute a score $s_i \in [0, 1]$ for each document $d_i \in D$.

- higher scores $\rightarrow$ d_i is more relevant to q.

Ranking one-word queries

- Search index for "dog".
- What should the score for a document be?

Jaccard coefficient

$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$

- $J(A, A) = 1$
- $J(A, B) = 0$ if $A \cap B = 0$
- Example:
  - *q:* dogs with hats
  - d_1: big cats wearing hats
  - d_2: I like hats.
  - $J(q, d_1)$? $J(q, d_2)$?

- $J(q, d_1) = \frac{|\{hats\}|}{|\{dogs, with, hats, big, cats, wearing\}} = \frac{1}{6}$

- $J(q, d_2) = \frac{|\{hats\}|}{|\{dogs, with, hats, I, like\}} = \frac{1}{5}$

Issues

- **term frequency**: documents with more occurrences of query term should be ranked higher than documents with few occurrences.
  - q="dog", score("dog cat dog") > score("dog cat cat")
- **rare terms**: rare terms matter more than common terms
  - q="the dog", score("dog cat") > score("the cat")
- **length normalization**: how can we make the scores comparable for long and short documents?

Term frequency

- Let tf_{t,d} be the frequency of term t in document d.
- What is the functional relationship between tf_{t,d} and relevance?
  - linear, log, something else?

- \log typically used
- The weight of term t in d is:
  - w_{t,d} = 1 + \log tf_{t,d} if tf_{t,d} > 0; otherwise 0.

- score s_i = \sum_{t \in q \cap d_i}(1 + \log tf_{t,d_i})
  - sum of term weights for unique terms in query and document
- If no query terms present? 0 score.

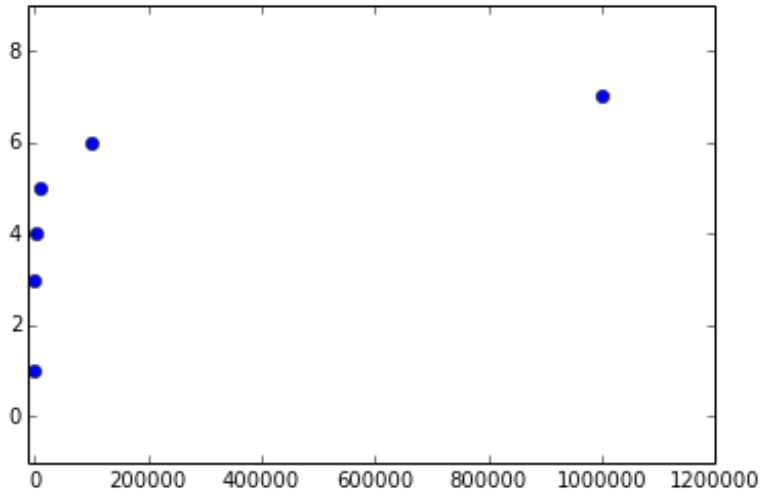
In [2]:

```
%pylab inline
tfs = [1, 100, 1000, 10000, 100000, 1000000]
xlim((-10000,1.2e6))
ylim((-1, 9))
plot(tfs, [1 + math.log10(tf) for tf in tfs], 'bo')
```

```
Populating the interactive namespace from numpy and matplotlib
```

Out[2]:

```
[<matplotlib.lines.Line2D at 0x10c2d52d0>]
```



Document frequency

- We want to weight rare terms more than common terms (e.g., *the* versus *excogitate*).

- **idf weight:**
  - Let N be total number of documents
  - Let df_t be the total number of documents that term t appears in.
  - Define **inverse document freqency** as idf_t = \log (\frac{N}{df_t})

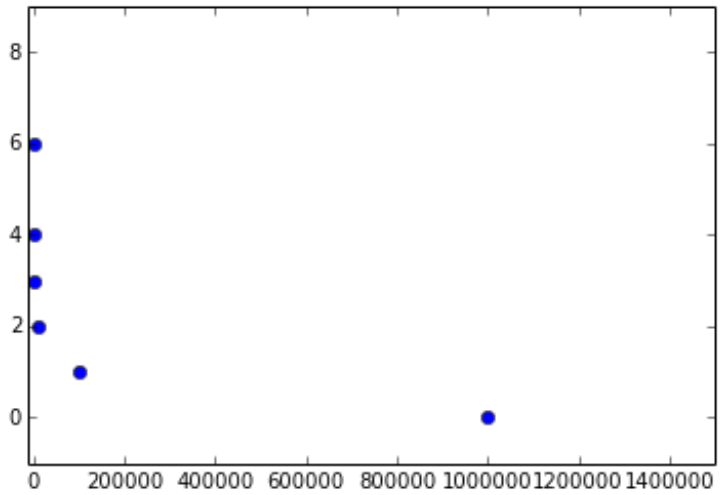
In [3]:

```
%pylab inline
N = 1e6
dfs = [1, 100, 1000, 10000, 100000, 1000000]
xlim((-10000,1.5e6))
ylim((-1, 9))
plot(dfs, [math.log10(1. * N/df) for df in dfs], 'bo')
```

```
Populating the interactive namespace from numpy and matplotlib
```

Out[3]:

```
[<matplotlib.lines.Line2D at 0x10c37a410>]
```



What effect does this have on one term queries?

- **none**

tf-idf weight

- Multiply tf and idf

- w_{t,d} = (1 + \log tf_{t,d}) \times \log (\frac{N}{df_t})
- Perhaps the most common baseline weighting system in IR
- Increases with term frequency in document; increases with rarity of term

Final score:

s_i = \sum_{t \in q \cap d_i} w_{t, d}

Document representation

Previously, each document is a "bag-of-words"

- "I saw the man in the hat in the park." \rightarrow `{i: 1, saw: 1, the: 2, man: 1, in: 2, park: 1}`
  - order doesn't matter; just term frequency

Now, each term is weighted by *tf-idf*.

Document Space

Each document d_i is now a vector in V-dimensional space, where V is the number of terms.

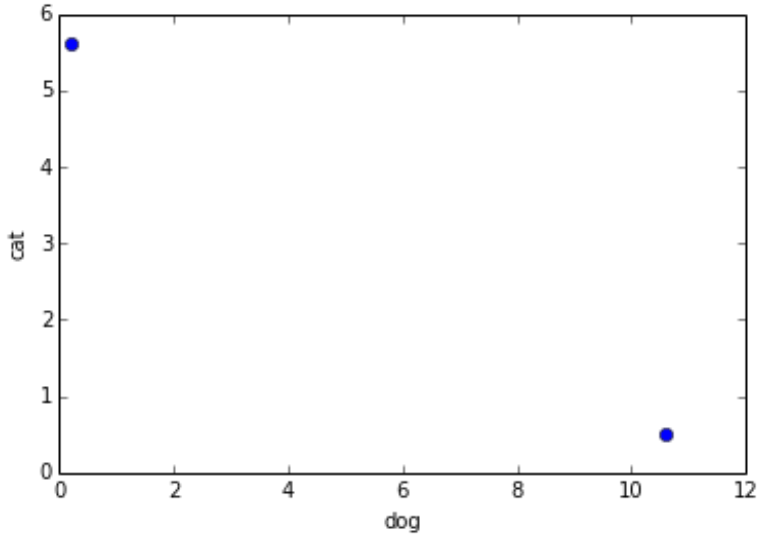
- e.g., assume position0 is "dog" and position1 is "cat", then two documents could be:
  - d_1 = \{10.6, 0.5\} (mostly about dogs)
  - d_2 = \{0.2, 5.6\} (mostly about cats)

In [4]:

```
plot([10.6, 0.2], [.5, 5.6], 'bo')
xlabel('dog')
ylabel('cat')
```

Out[4]:

```
<matplotlib.text.Text at 0x10c37f650>
```

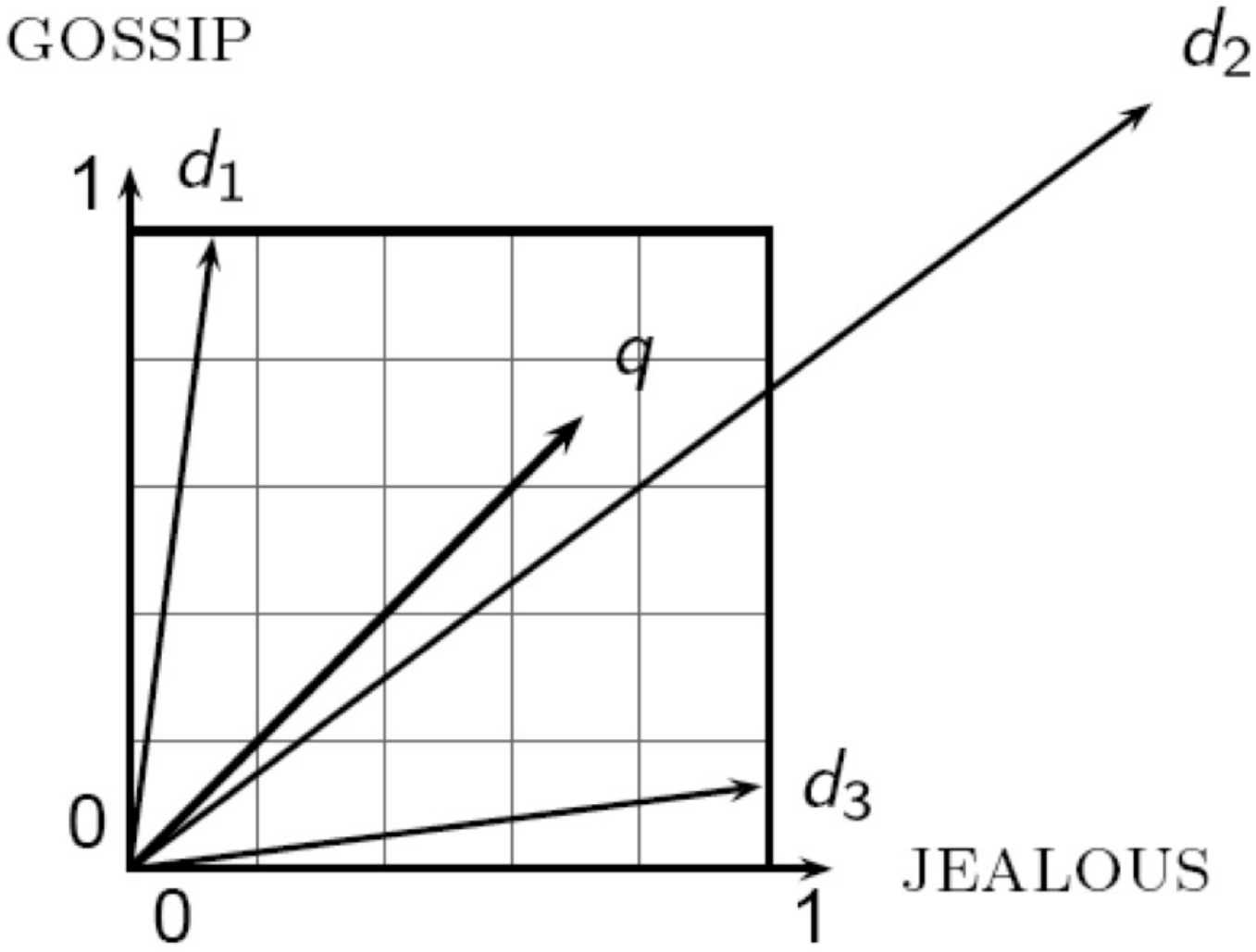


First idea: Euclidean distance

- Score a query by the distance between the query and document in vector space.

dist(a, b) = \sqrt{\sum_i (a_i - b_i)^2}

Problem with Euclidean distance

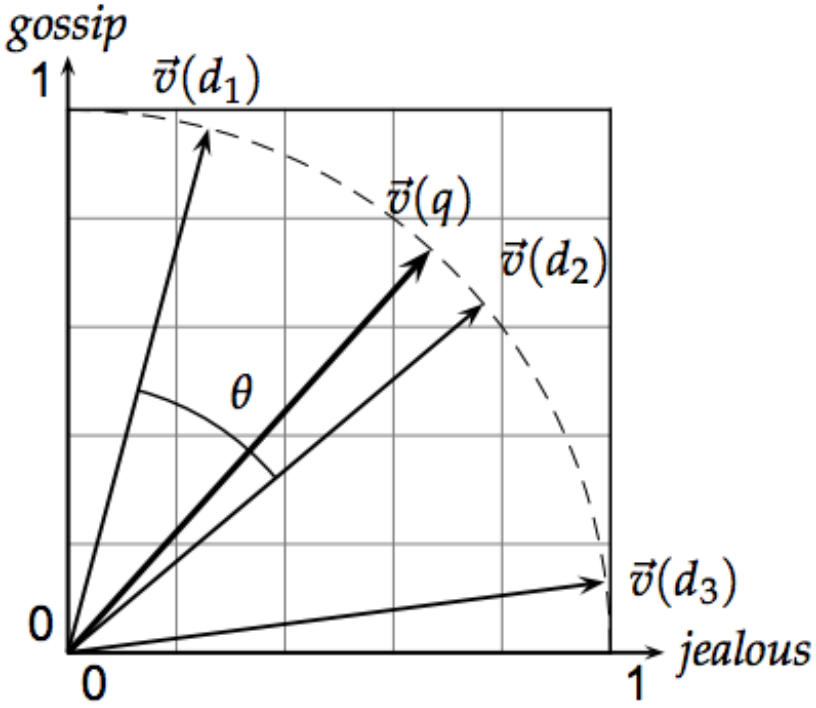


q is similar to d_2, but because d_2 is longer, Euclidean distance is large

- Another way of thinking about it:
  - Let d_i^2 be d_i appended to itself.
  - dist(d_i, d_i^2) should be small, but it is not

Cosine similarity

Idea: score query by angle between query vector and document vector



► **Figure 6.10** Cosine similarity illustrated. $\text{sim}(d_1, d_2) = \cos\theta$.

Cosine similarity

sim(a, b) = \frac{a \cdot b}{||a||\hbox{ } ||b||}

- a \cdot b is dot product: \sum_i a_i \times b_i
- ||a|| is norm: \sqrt{\sum_i a_i^2}

Exercise

Given three documents:

- d_1: `['the', 'dog', 'barked']`
- d_2: `['the', 'dog', 'jumped']`
- d_3: `['a', 'cat', 'jumped']`

1. Compute tf-idf vectors for d_1, d_2
2. Compute the cosine similarity between d_1 and d_2

CS 429: Information Retrieval

Lecture 8: Scalable scoring and system integration

Dr. Aron Culotta

Illinois Institute of Technology

Spring 2014

Last time:

- tf-idf weights for each document
- Vector Space Model
- Cosine Similarity

Today:

- How to efficiently retrieve top ranked documents
- Full search pipeline
- Grab bag

tf-idf weight:

- w_{t,d} = (1 + \log tf_{t,d}) \times \log (\frac{N}{df_t})

cosine similarity:

sim(a, b) = \frac{a \cdot b}{||a||\hbox{ } ||b||}

- a \cdot b is dot product: \sum_i a_i \times b_i
- ||a|| is norm: \sqrt{\sum_i a_i^2}

search:

- convert each query and document into *tf-idf* vectors q and d
- sort documents by sim(q, d)

In [ ]:

```
# Score documents by cosine similarity
from collections import defaultdict

# tf-idf weighted query
query = {'the': 0.01, 'zygote': 14.2}
```

```
# index is list of (doc_id, tf-idf weight) pairs
index = {'the': [(0, 44), (1, 100)],
         'zygote': [(0, 100), (1, 44)]}

# document lengths, for normalization
doc_lengths = {0: 12, 1: 12}

def cosine(query, index, doc_lengths):
  scores = defaultdict(lambda: 0)
  # For each search term
  for query_term, query_weight in query.items():
      # For each matching doc
      for doc_id, doc_weight in index[query_term]:
          scores[doc_id] += query_weight * doc_weight  # part of dot product

  # normalize by doc length (why not also by query length?)
  for doc_id in scores:
    scores[doc_id] /= doc_lengths[doc_id]
  return sorted(scores.items(), key=lambda x: x[1], reverse=True)

results = cosine(query, index, doc_lengths)
print results
```

What is runtime?

O(QN) where Q is number of query terms and N is number of documents containing each query term.

Faster Cosine Search

If only retrieving top k, how can we do better than

In []:

```
sorted(scores.items(), key=lambda x: x[1], reverse=True)
```

No need to sort all scores: use priority queue of size k

- O(2J) to construct heap, where J is number of docs with non-zero score
- O(k \log J) to find top k

Approximate k-best

- How can we find *almost* the top k documents?

- **Idea:**
  - Find a set A of *contenders* K < |A| << N
  - Only compute cosine similarity between query and A

We'll consider a number of approaches.

Only use high idf terms

- Similar to pruning stop words
- Since low idf terms occur in many documents, this prunes a lot

- What's the downside?

Soft conjunction

- If query has 4 terms
  - Retrieve all docs that match at least 3 terms
  - Compute cosine similarity for this subset
- How to find matches efficiently?

Champion lists

At index time:

- For each term t
  - compute r documents that have highest weight for t ("champion lists")

At query time:

- Take union of champion lists of all query terms
- Sort them by cosine similarity

- How can we use an inverted index for a champion lists?

Static quality scores

- Assign a score g(d) to each document at index time indicating how good it is
- Based on what?

- List of known good pages (Wikipedia, CNN, ...)
- Pages with many in-links, bookmarks
- PageRank
- Is it spam?

Static quality scores

- How to combine static score with cosine score?
  - **addition**: `netscore`(q,d) = g(d) + sim(q,d)
- How to efficiently find top by `netscore`?

- One speedup: Order postings lists by g(d)
  - Thus, we'll find top documents earlier
  - Good if have small time budget

Impact ordering

- Sort postings list in decreasing order of tfidf

In [31]:

```
index = {'the': [(0, 44), (1, 100)],
         'zygote': [(0, 100), (1, 44)]}
```

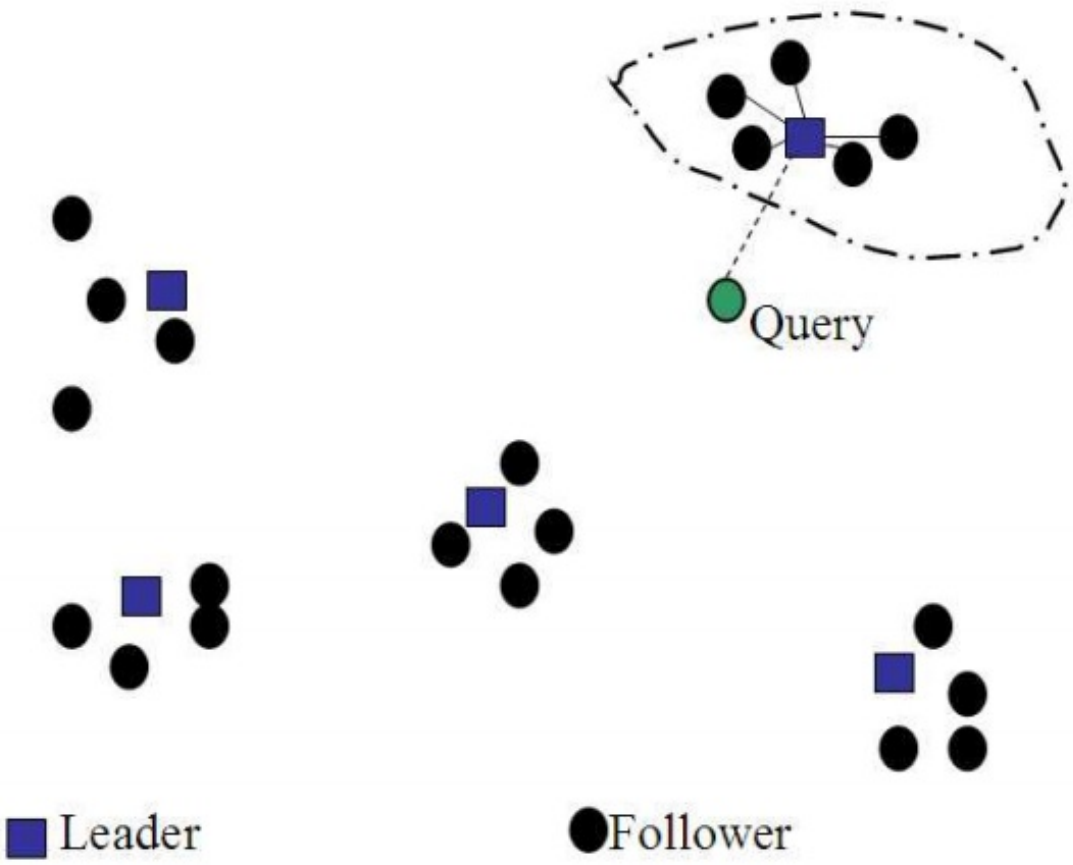
```
# becomes

index = {'the': [(1, 100), (0, 44)],
         'zygote': [(0, 100), (1, 44)]}
```

How does this affect our algorithm to merge postings lists?

We'll instead use our initial algorithm that accumulates scores one term at a time.

- Approximations:
  - **Early termination**: stop traversing postings after r docs or when weight drops below a threshold
  - idf **ordered search terms**: Sort query terms by idf and process in order. Stop if score doesn't change much with additional term.

Cluster pruning



▶ **Figure 7.3** Cluster pruning.

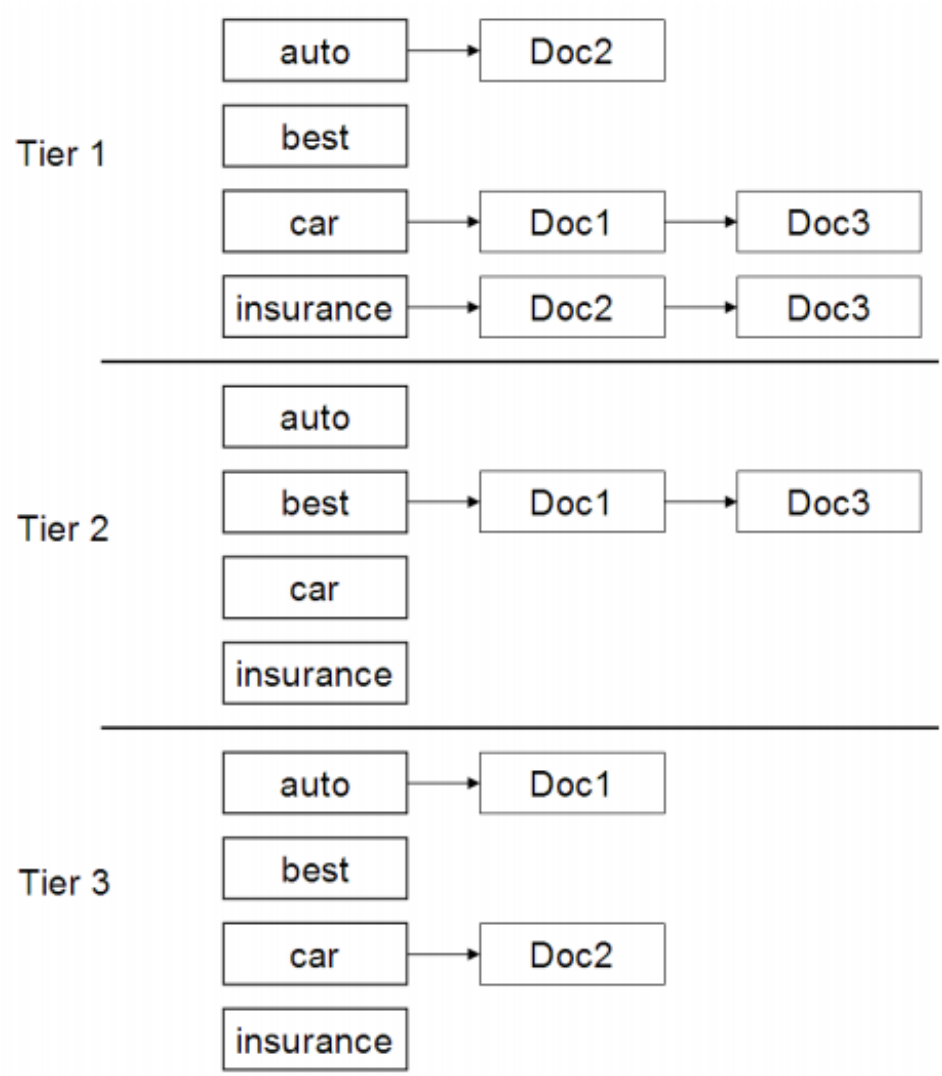
- pick \sqrt(N) docs at random (**leaders**)
- assign all other docs to nearest leader
- **follower**: doc attached to a leader
- for query Q
  - find nearest leader
  - find K-best followers
  - rank by cosine similarity

Improvements?

improvements?

- Select $b > 1$ leaders; attach each follower to $c > 1$ leaders
- when will cluster pruning fail?

Tiered indices



Field and Zone Search

- *Field:* year, name, etc (limited values)
- *Zone:* subsection of document (abstract, footer)

How to search these efficiently?

In [ ]:

```
index = {'the': [(0, 44), (1, 100)],
         'zygote': [(0, 100), (1, 44)],
         'the-title': [(0, 44), (1, 100)],
         'zygote-title': [(0, 100), (1, 44)],
         'the-abstract': [(0, 44), (1, 100)],
         'zygote-abstract': [(0, 100), (1, 44)],
         }
```

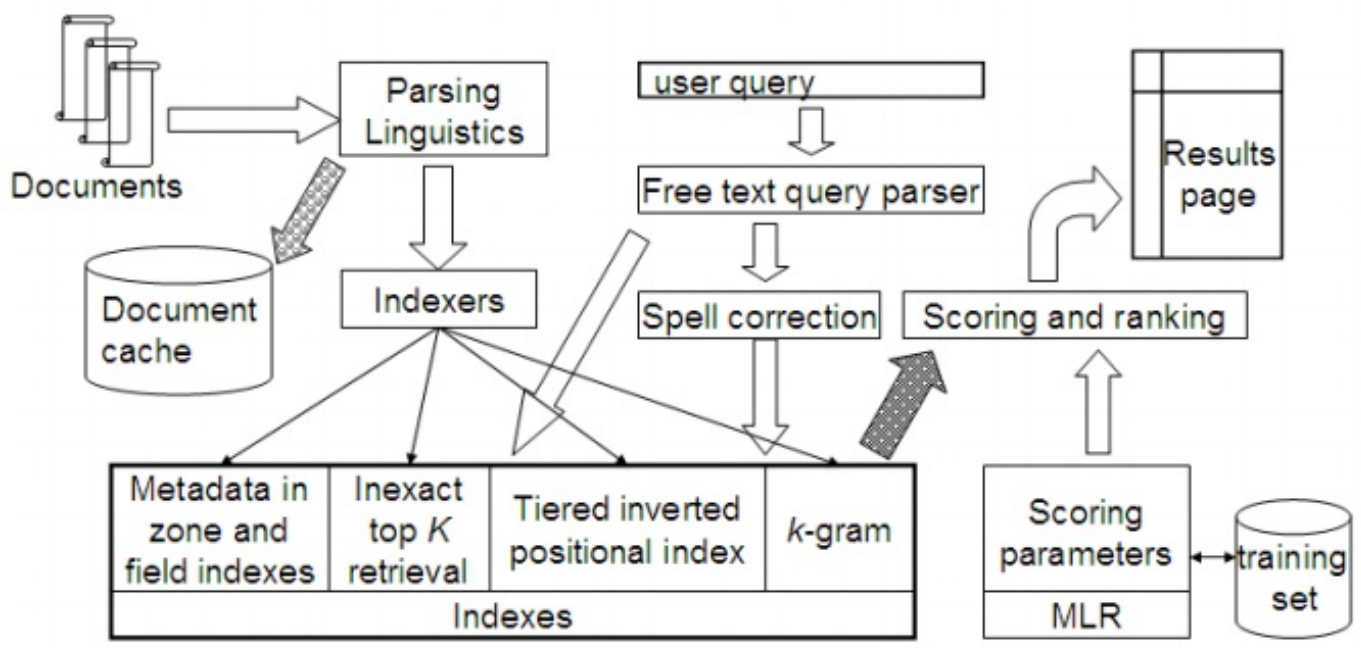
Query term proximity

- If query is: *dog catcher van*, how can we prefer documents where the three words occur in proximity?

Query parser

- Consider phrase query "pitchfork music festival"
- Submit various queries until get at least k results
  - "pitchfork music festival"
  - "pitchform music" AND "music festival"
  - pitchform AND music AND festival
  - ...

All together now...



CS 429: Information Retrieval

Lecture 9: Evaluation

Dr. Aron Culotta

Illinois Institute of Technology

Spring 2014

Last time

- Scalable cosine similarity ranking

Today

- Is our search engine any good?

Evaluation

- Your first day at Bing, your boss says "make our search engine better than Google's"
- How do you know when you're done?
- Maybe it's already better?

How to measure goodness?

1. Relevant results
2. Speed (indexing and search)
3. Query capabilities
4. Pleasing user interface
5. User is *happy*?

How to tell when the user is happy?

- They come back.
- They come back frequently.
- They take surveys and tell you they like you.
- We'll settle for measuring **relevance**.

What is relevance?

What is relevance?

A document is **relevant** if it addresses the user's information need.

What is **information need**?

- The user's information need is the explicit definition of what they are looking for.
- The query is only a proxy for that need.

- E.g. the **information need** may be "find a flight from Chicago to Detroit leaving tomorrow"
  - The **query** may be "ORD to DTW"

How relevant are the results of your search engine?

- An **Information Retrieval Benchmark** consists of:

  1. A documents collection.
  2. A set of queries (and corresponding information needs).
  3. A set of **relevance judgements** for each query-document pair (usually binary)

Document collection:

- Which documents?
- How many?

Queries:

- Which queries?
- How many?

Generally, want a diverse set of queries that people actually use, over a diverse set of documents that people actually read.

Relevance judgements

- Who is making these judgements?
- How many do we need?
- How do we know they are right?

- Suppose we have 10M documents and 1K queries:
  - 10B relevance judgements??

- Crowd-sourcing can help, but not solve.
- Either consider small datasets (<10K docs)
- Or only judge a subset
  - E.g., only consider those documents in top K by some reasonable baseline system

Are the humans right?

- How do we know we can trust the human relevance judgements?

- Make multiple people label the same document.

- **Inter-annotator agreement:** How often do two humans agree on the label?
- E.g., consider two humans labeling 100 documents:

		Person 1	
		Relevant	Not Relevant
Person 2	Relevant	50	20
	Not Relevant	10	20

- Simple **agreement**: fraction of documents with matching labels. $\frac{70}{100} = 70\%$
- But, how much agreement would we expect by chance?
- Person 1 says Relevant 60% of the time.
- Person 2 says Relevant 70% of the time.
- Chance that they both say relevant at the same time? $60\% \times 70\% = 42\%$.
- Person 1 says Not Relevant 40% of the time.
- Person 2 says Not Relevant 30% of the time.
- Chance that they both say not relevant at the same time? $40\% \times 30\% = 12\%$.
- Chance that they agree on any document (both say yes or both say no): $42\% + 12\% = 54\%$

Cohen's Kappa κ

- Percent agreement beyond that expected by chance

$ \kappa = \frac{P(A) - P(E)}{1 - P(E)}$

- $P(A)$ = simple agreement proportion
- $P(E)$ = agreement proportion expected by chance

E.g., $\kappa = \frac{.7 - .54}{1 - .54} = .3478$

- $k=0$ if no better than chance, $k=1$ if perfect agreement

Assuming humans are right...

Given document collection, a query, and relevance judgements, what score do we give to our search engine?

- Assume search engine returns K results

	Relevant	**Nonrelevant**
Retrieved	true pos (**tp**)	false pos (**fp**)
Not Retrieved	false neg (**fn**)	true neg (**tn**)

Example:

```
Rank    DocID    Relevant
----    -----    --------
```

```
1       123     Y
2       456     N
3       789     N
4       321     Y
5       654     N
```

Assume there are 10 relevant documents total and 100 documents in the collection.

	Relevant	**Nonrelevant**
Retrieved	2 (**tp**)	3 (**fp**)
Not Retrieved	8 (**fn**)	87 (**tn**)

Accuracy: $\frac{tp + tn}{tp + fn + fp + tn}$

e.g., $\frac{2 + 87}{2 + 8 + 3 + 87} = 89\%$

- Why is this a mostly useless measure?

Very few documents are relevant, so returning 0 documents has nearly perfect true negatives.

	Relevant	**Nonrelevant**
Retrieved	0 (**tp**)	0 (**fp**)
Not Retrieved	10 (**fn**)	90 (**tn**)

Accuracy: $\frac{tp + tn}{tp + fn + fp + tn}$

e.g., $\frac{0 + 90}{0 + 10 + 0 + 90} = 90\%$

Precision and Recall

	Relevant	**Nonrelevant**
Retrieved	true pos (**tp**)	false pos (**fp**)
Not Retrieved	false neg (**fn**)	true neg (**tn**)

- **Precision:** $P = \frac{tp}{tp + fp}$, The fraction of *returned* documents that are *relevant*.
- **Recall:** $R = \frac{tp}{tp + fn}$, The fraction of *relevant* documents that are *returned*.

	Relevant	**Nonrelevant**
Retrieved	2 (**tp**)	3 (**fp**)
Not Retrieved	8 (**fn**)	87 (**tn**)

$P = \frac{tp}{tp + fp} = \frac{2}{2 + 3} = 40\%$

$R = \frac{tp}{tp + fn} = \frac{2}{2 + 8} = 20\%$

	Relevant	**Nonrelevant**

Retrieved	0 (**tp**)	0 (**fp**)
Not Retrieved	10 (**fn**)	90 (**tn**)

$P = \frac{tp}{tp + fp} = \frac{0}{0} = NaN \rightarrow 0\%$

$R = \frac{tp}{tp + fn} = \frac{0}{10} = 0\%$

How to combine precision and recall?

Harmonic mean of P and R is $\frac{1}{\frac{1}{P} + \frac{1}{R}}$

Weighted harmonic mean: $\frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}}$

- Greater $\alpha \rightarrow$ precision is more important than recall

F1: Setting $\alpha=\frac{1}{2}$ leads to: $\frac{2\cdot P \cdot R}{P + R}$

- **F1** commonly used; weights precision and recall equally

Why didn't we just use the arithmetic mean? ($\frac{P + R}{2}$)

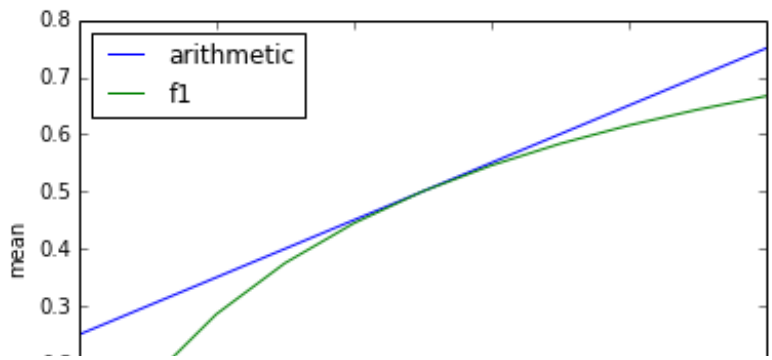
In [38]:

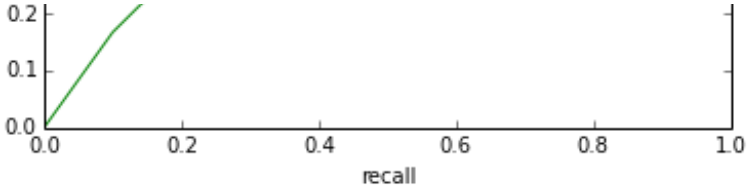
```
%pylab inline

# Compare arithmetic mean of precision and recall to F1 for fixed precision of
50%.

def arith_mean(p, r):
    return (p + r) / 2.0

def f1(p, r):
    return (2.0 * p * r) / (p + r)

p = .5
r = [0, .1, .2, .3, .4, .5, .6, .7, .8, .9, 1.0]

xlabel('recall')
ylabel('mean')
plot(r, [arith_mean(p, ri) for ri in r], label='arithmetic')
plot(r, [f1(p, ri) for ri in r], label='f1')
legend(loc='best')
```

```
Populating the interactive namespace from numpy and matplotlib
```

Out[38]:

```
<matplotlib.legend.Legend at 0x109cc56d0>
```





Precision-Recall Curves

- Depending on use-case, precision or recall may be more important
  - Lawyer who cannot miss a single "smoking-gun" document needs 100% recall.
  - Average web user doesn't want **all** documents with the song lyrics she's searching for.
- Precision-recall curves displays results varying the number of returned documents

```
Rank    DocID    Relevant
----    -----    --------
1       123      Y
2       456      N
3       789      N
4       321      Y
5       654      N
```

Assume there are 10 relevant documents total and 100 documents in the collection.

$P = \frac{tp}{tp + fp} = \frac{2}{2 + 3} = 40\%$

$R = \frac{tp}{tp + fn} = \frac{2}{2 + 8} = 20\%$

```
Rank    DocID    Relevant
----    -----    --------
1       123      Y
2       456      N
3       789      N
4       321      Y
5       654      N
--
6       987      Y
7       135      N
8       246      N
9       357      N
10      468      N
```

	Relevant	**Nonrelevant**
Retrieved	~~2~~ 3 (**tp**)	~~3~~ 7 (**fp**)
Not Retrieved	~~8~~ 7 (**fn**)	~~87~~ 83 (**tn**)

$P = \frac{tp}{tp + fp} = \frac{3}{3 + 7} = 30\%$

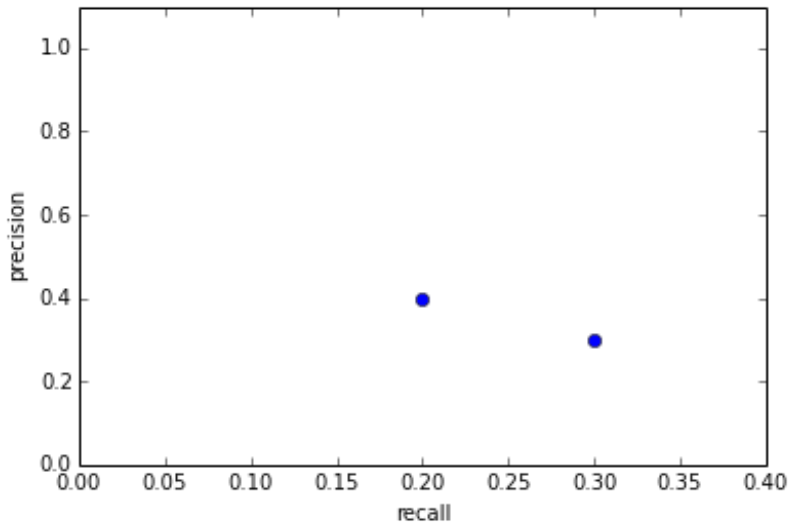
$R = \frac{tp}{tp + fn} = \frac{3}{3 + 7} = 30\%$

In [39]:

```
xlabel('recall')
ylabel('precision')
plot([.2, .3], [.4, .3], 'bo')
xlim((0, .4))
ylim((0, 1.1))
```

Out[39]:

```
(0, 1.1)
```



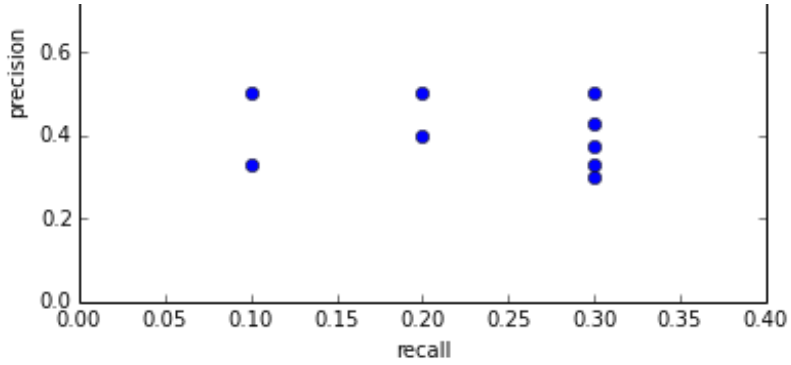
Rank	DocID	Relevant	P	R
1	123	Y	1.0	0.1
2	456	N	0.5	0.1
3	789	N	0.33	0.1
4	321	Y	0.5	0.2
5	654	N	0.4	0.2
6	987	Y	0.5	0.3
7	135	N	0.43	0.3
8	246	N	0.375	0.3
9	357	N	0.33	0.3
10	468	N	0.3	0.3

In [40]:

```
# compute precision/recall at each sublist of size 1 to 10
xlabel('recall')
ylabel('precision')
precisions = [1, .5, .33, .5, .4, .5, .43, .375, .33, .3]
recalls = [.1, .1, .1, .2, .2, .3, .3, .3, .3, .3]
plot(recalls, precisions, 'bo')
xlim((0, .4))
ylim((0, 1.1))
```

Out[40]:

```
(0, 1.1)
```



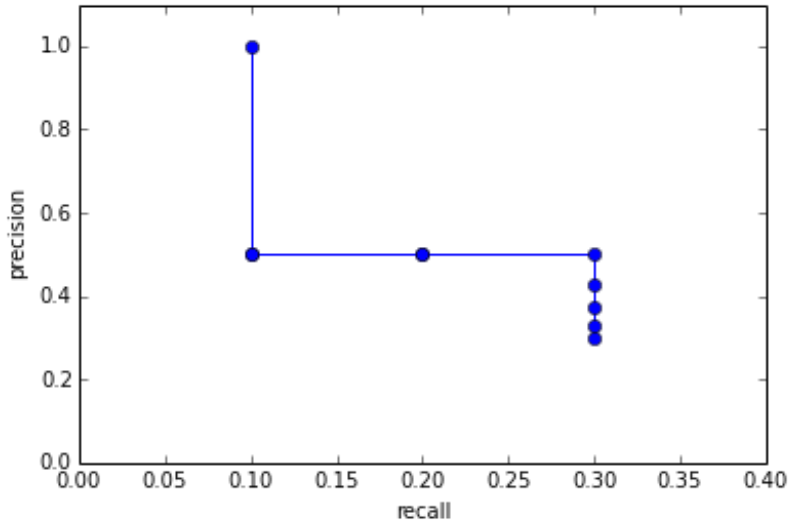
In [41]:

```
# Interpolated precision: max of precisions to right of value
xlabel('recall')
ylabel('precision')
interpolated_pre = [max(precisions[i:]) for i in range(len(precisions))]
print interpolated_pre
step(recalls, interpolated_pre, 'bo')
xlim((0, .4))
ylim((0, 1.1))
```

```
[1, 0.5, 0.5, 0.5, 0.5, 0.5, 0.43, 0.375, 0.33, 0.3]
```

Out[41]:

```
(0, 1.1)
```



Mean Average Precision (MAP)

- Average the precision values for top k documents, considering only those elements where a relevant document is found.

Rank	DocID	Relevant	P	R
1	123	Y	*1.0*	0.1
2	456	N	0.5	0.1
3	789	N	0.33	0.1
4	321	Y	*0.5*	0.2
5	654	N	0.4	0.2
6	987	Y	*0.5*	0.3

7	135	N	0.43	0.3
8	246	N	0.375	0.3
9	357	N	0.33	0.3
10	468	N	0.3	0.3

If relevant document not returned, assume 0 precision for those.

$MAP=\frac{1.0 + 0.5 + 0.5 + 0 + \ldots + 0}{10} = .2$

R-Precision

Precision considering the top R documents, where R is the number of relevant documents.

In our example, precision "at 10" (P@10) is 0.3

Perfect system has R-precision of 1.0 (e.g., top R results are all relevant)

Labeling data is hard...

Can we get it without trying?

For query q, the system returns

Rank	DocID
1	123
2	456
3	789

Click-through data:

DocID	Clicks
123	1,000
456	500
789	100

Does this mean that users think document 123 is more relevant than document 456?

Positional bias

Users are *a-priori* more likely to click a higher ranked document.

How can we compensate for this?

Pairwise preferences:

DocID	Clicks
123	1,000
456	500
789	5,000

Can probably conclude that 789 > 456 and 789 > 123

How can we compute a score from this?

Let L^* be the set of **true** pairwise preferences $\{d_i > d_j\}$

Let L be the ranking produced by our system: $d_{123} > d_{456} > d_{789} \ldots$

Let $r_L(i, j)$ be 1 if ranking L ranks i before j.

Kendall tau distance is the number of pairwise disagreements:

$\tau(L, L^*) = {\Big|}{\big \{}(i,j) $ s.t. $r_L(i, j) \ne r_{L^*}(i, j){\big \}}{\Big|}$

If perfectly matched: $\tau=0$; if perfectly mismatched: $\tau=\Big| L^* \Big|$

Interleaving rankings

Given two ranking systems A and B, which is better?

Interleave rankings from each, showing half with A first, half with B first.

Group 1: A_1, B_1, A_2, $B_2 \ldots$

Group 2: B_1, A_1, B_2, $A_2 \ldots$

E.g., if A_1 is clicked more than B_1 in Group 2, then A might be better than B.

A/B Testing

A method of measuring the impact of a system parameter.

- E.g., Should I stem? What should the size of the champion list be? How should I tokenize?

To do A/B testing:

1. Create a search engine with million of users.
2. Divert a small sample (1%) to the *experimental* search engine with that feature modified.
3. Measure difference in user happiness between groups.

- Happiness:
  - proportion of time first result is clicked
  - proportion of time top k result is clicked

CS 429: Information Retrieval

Lecture 10: Query Expansion

Dr. Aron Culotta

Illinois Institute of Technology

Spring 2014

Last time:

- Evaluation
  - accuracy, precision, recall, MAP

This time:

- How can we incoporate user feedback to improve search?
- How can we alter the user's query to improve search?

Relevance Feedback

- An *interactive* IR system in which

1. The user enters a query.
2. The system returns results.
3. The user indicates which results are relevant.
4. GoTo 2.

How should we incorporate user feedback?

- Create a new query that is similar to relevant documents but dissimilar to irrelevant documents.

Rocchio

$ \DeclareMathOperator*{\argmax}{arg\,max}$ $\vec{q}^* \leftarrow \argmax_{\vec{q}} sim(\vec{q}, C_r) - sim(\vec{q}, C_{nr})$

- where q is a query
- C_r is a set of relevant documents
- C_{nr} is a set of irrelevant documents
- sim is cosine similarity

Document Centroid

Recall that we represent each document as a vector of tf-idf values.

Given a collection of documents $D = \{d_1 \ldots d_N\}$, the centroid vector is:

$ \frac{1}{N} \sum_{\vec{d_j} \in D}\vec{d}_j $

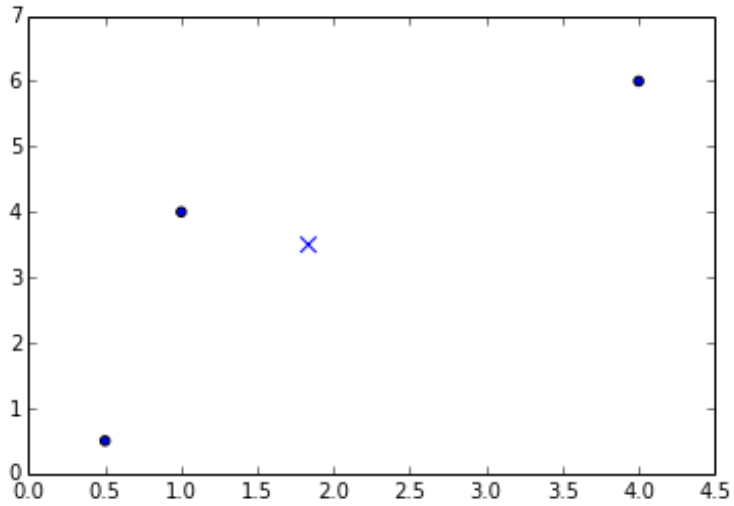
In [157]:

```
%pylab inline
import numpy as np

points = [[1, 4],
          [.5, .5],
          [4, 6]]

centroid = np.sum(points, axis=0) / len(points)
scatter([p[0] for p in points], [p[1] for p in points])
scatter([centroid[0]], [centroid[1]], marker='x', s=60)
```

```
Populating the interactive namespace from numpy and matplotlib
```

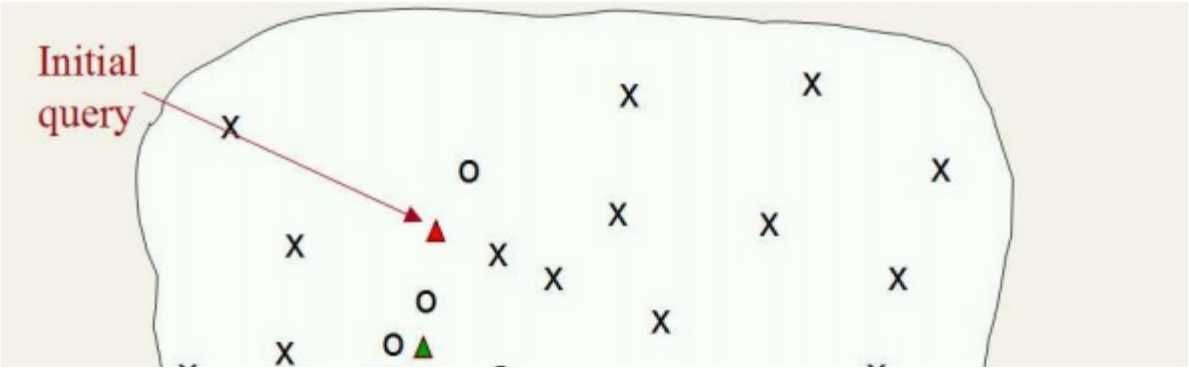
Out[157]:

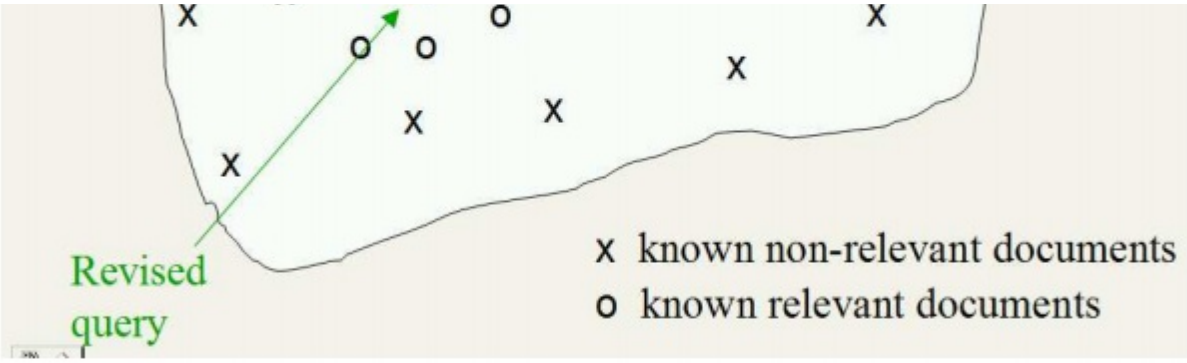
```
<matplotlib.collections.PathCollection at 0x10f690dd0>
```



Want a query that is closest to relevant documents, but far from irrelevant documents.

$\vec{q}^* = \frac{1}{|C_r|} \sum_{\vec{d_j} \in C_r}\vec{d}_j - \frac{1}{|C_{nr}|} \sum_{\vec{d}_j \in C_{nr}} \vec{d}_j$





► **Figure 9.4** An application of Rocchio's algorithm. Some documents have been labeled as relevant and nonrelevant and the initial query vector is moved in response to this feedback.

Source: MRS

But, we don't know the set of all relevant and irrelevant documents.

$\vec{q}_m = \alpha \vec{q}_0 + \beta\frac{1}{D_r} \sum_{\vec{d}_j \in D_r} \vec{d}_j - \gamma\frac{1}{|D_{nr}|} \sum_{\vec{d_j} \in D_{nr}} \vec{d}_j$

- $\vec{q}_0$ is original query vector
- α, β, γ are tunable parameters.

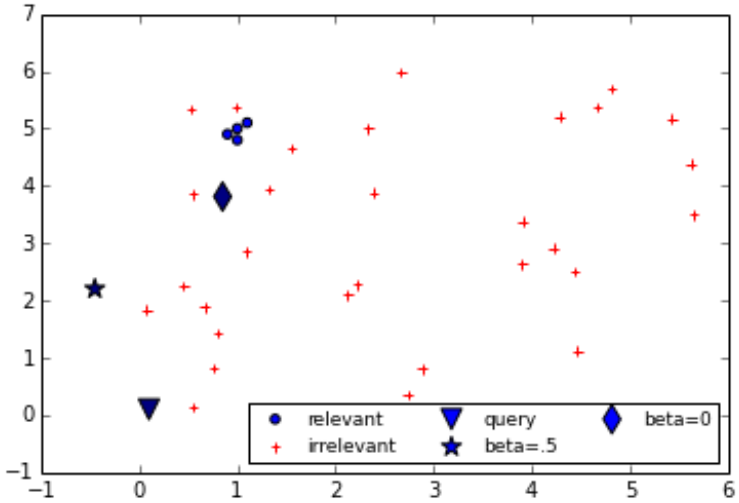
In [162]:

```
# Plot effect of relevance feedback as we change parameters.
import numpy as np
from numpy import array as npa
import random as rnd

def centroid(docs):
    return np.sum(docs, axis=0) / len(docs)

def rocchio(query, relevant, irrelevant, alpha, beta, gamma):
    return alpha * query + beta * centroid(relevant) - gamma * centroid(irreleva
nt)

# Create some documents
relevant = npa([[1, 5], [1.1, 5.1], [0.9, 4.9], [1.0, 4.8]])
irrelevant = npa([[rnd.random()*6, rnd.random()*6] for i in range(30)])

# Create a query
query = npa([.1, .1])

# Compute two different Rocchio updates (beta=0.5, beta=0)
new_query_b5 = rocchio(query, relevant, irrelevant, 1., .75, .5)
new_query_b0 = rocchio(query, relevant, irrelevant, 1., .75, 0.)

# Plot them.
pos = scatter([p[0] for p in relevant], [p[1] for p in relevant])
neg = scatter([p[0] for p in irrelevant], [p[1] for p in irrelevant], marker='+'
, edgecolor='red')
q = scatter(query[0], query[1], marker='v', c=0.1, s=100)
newq_b5 = scatter([new_query_b5[0]], [new_query_b5[1]], marker='*', s=100, c=.9)
newq_b0 = scatter([new_query_b0[0]], [new_query_b0[1]], marker='d', s=100, c=.8)
plt.legend((pos, neg, q, newq_b5, newq_b0),
```

```
            ('relevant', 'irrelevant', 'query', 'beta=.5', 'beta=0'),
            scatterpoints=1,
            loc='lower right',
            ncol=3,
            fontsize=9)
```

Out[162]:

```
<matplotlib.legend.Legend at 0x10dc936d0>
```



- $\gamma=0$ Often used, since we're more confident in relevant annotations than irrelevant.
- One might decrease α as the number of relevant documents increase.

Does relevance feedback help precision or recall?

- Mostly recall: "adding" similar terms to query vector from relevant documents.
- When would it not help?

- Spelling correction?
- Different language?
- Synonyms?

- Assumption 1: query is "close" to relevant documents
  - feedback makes the query closer
- Assumption 2: relevant documents form one cluster.

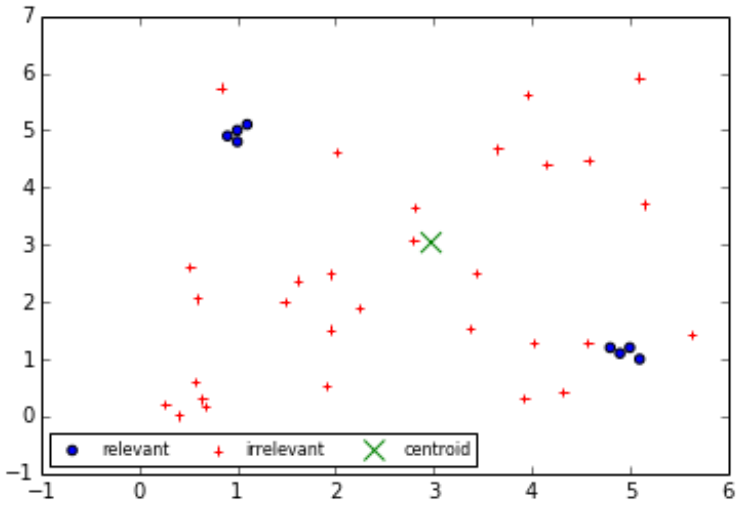
In [163]:

```
# What happens if there are two clusters of relevant examples?

import numpy as np
import random as rnd

points = [[1, 5], [1.1, 5.1], [0.9, 4.9], [1.0, 4.8],
          [5, 1.2], [4.9, 1.1], [5.1, 1.0], [4.8,1.2]]

centroid = np.sum(points, axis=0) / len(points)
pos = scatter([p[0] for p in points], [p[1] for p in points])
neg = scatter([rnd.random()*6 for i in range(30)], [rnd.random() * 6 for i in ra
nge(30)], marker='+', edgecolor='red')
centroid = scatter([centroid[0]], [centroid[1]], marker='x', s=100, edgecolor='g
reen')
```

```
plt.legend((pos, neg, centroid),
           ('relevant', 'irrelevant', 'centroid'),
           scatterpoints=1,
           loc='lower left',
           ncol=3,
           fontsize=8)
```

Out[163]:

```
<matplotlib.legend.Legend at 0x10f7ec950>
```



Does relevance feedback affect search time?

- Much longer queries
- How to approximate?
  - Use top k most informative terms from relevant set.

Variants of relevance feedback

- Pseudo-relevance: Assume top k documents are relevant.
- Indirect relevance: Mine click logs.

Pseudo-relevance feedback

1. Rank documents
2. Let V be the top k documents. We pretend these are all relevant.
3. Update q according to Rocchio

We can iterate steps $2-4$ until ranking stops changing.

When would this work? When would this not work?

Explicit query expansion

- Thesaurus
- Word co-occurrences
- Mine reformulations from query log

WordNet

WordNet

http://wordnetweb.princeton.edu/perl/webwn

Thesaurus discovery

Idea: Look for words that occur in same context.

- "He put the mug on the _____"

In [164]:

```
from collections import Counter, defaultdict
from sklearn.datasets import fetch_20newsgroups
import re
docs = fetch_20newsgroups(subset='train', remove=('headers', 'footers',
'quotes'),
                          categories=['comp.graphics', 'comp.sys.mac.hardware',
'comp.sys.ibm.pc.hardware',
                                      'comp.os.ms-windows.misc']).data

# Count words that occur within a window of -n to +n of each word.
def term2contexts(docs, n):
    contexts = defaultdict(lambda: Counter())
    for d in docs:
        toks = re.findall('[\w]+', d.lower())
        for i in range(len(toks)):
            contexts[toks[i]].update(toks[i-n:i] + toks[i+1:i+n+1])
    return contexts

contexts = term2contexts(docs, 1)
print contexts['email']
```

```
Counter({u'please': 31, u'me': 26, u'or': 21, u'to': 18, u'via': 15, u'address'
: 14, u'by': 13, u'and': 10, u'as': 7, u'send': 6, u'the': 6, u'addresses': 5,
u'it': 5, u'if': 5, u'i': 5, u'replies': 4, u's': 4, u'thanks': 3, u'an': 3, u'
sp': 3, u'at': 3, u'also': 2, u'your': 2, u'responses': 2, u'my': 2, u'with': 2
, u'can': 2, u'have': 2, u'any': 2, u'you': 2, u'response': 2, u'request': 2, u
'time': 2, u'canberra': 1, u'soon': 1, u'03051': 1, u'through': 1, u'edu': 1, u
'anyoneand': 1, u'query': 1, u'mikael_fredriksson': 1, u'whose': 1, u'vatti': 1
, u'mail': 1, u'weeks': 1, u'7197': 1, u'karlth': 1, u'fax': 1, u'his': 1, u'me
ans': 1, u'very': 1, u'utah': 1, u'possible': 1, u'87131': 1, u'whichever': 1,
u'every': 1, u'not': 1, u'advanced': 1, u'server': 1, u'try': 1, u'9570': 1, u'
2393': 1, u'x': 1, u'tp923021': 1, u'gmbh': 1, u'schaefer': 1, u'gloege': 1, u'
id': 1, u'80': 1, u'robert': 1, u'thank': 1, u'current': 1, u'enough': 1, u'rep
ly': 1, u'khoros': 1, u'available': 1, u'ken': 1, u'we': 1, u'hansch': 1,
u'300': 1, u'sole': 1, u'1933': 1, u'joe': 1, u'haston': 1, u'care': 1, u'advan
ce': 1, u'c': 1, u'limits': 1, u'could': 1, u'etc': 1, u'corrections': 1,
u'rodney': 1, u'com': 1, u'facility': 1, u'imagine': 1, u'prefer': 1,
u'fredriksson': 1, u'directly': 1, u'johne': 1, u'7795966': 1, u'there': 1, u'2
': 1, u'leehian': 1, u'that': 1, u'happily': 1, u'but': 1, u'capabilities': 1,
u'ma': 1, u'submissions': 1, u'ihno': 1, u'growing': 1, u'similar': 1, u'6695':
1, u'508336': 1, u'documentation': 1, u'is': 1, u'good': 1, u'something': 1, u'
in': 1, u'mouse': 1, u'karl': 1, u'information': 1, u'no': 1, u'get': 1, u'when
': 1, u'd2002': 1, u'1': 1, u'tim': 1, u'valid': 1, u'wes1574': 1, u'faxes': 1,
u'singapore': 1, u'66s': 1, u'dicta93': 1, u'after': 1, u'wnkretz': 1,
u'annee': 1, u'a': 1, u'markus': 1, u'marchesf': 1, u'contact': 1,
u'corporate': 1})
```

In [165]:

```
import math
# Compute inverse document frequency values for each term.
def compute_idfs(docs):
    idfs = Counter()
    for d in docs:
        toks = set(re.findall('[\w]+', d.lower()))
        idfs.update(toks)
    for d in dfs:
        idfs[d] = math.log(1.0 * len(docs) / idfs[d])
    return idfs

idfs = compute_idfs(docs)
print 'idf of the=', idfs['the'], ' of monitor=', idfs['monitor']
```

```
idf of the= 0.207475223156  of monitor= 2.86134763856
```

In [166]:

```
def cosine(term1, term2):
    context1 = contexts[term1]
    context2 = contexts[term2]
    sim = sum(idfs[term] * context1[term] * context2[term] for term in context1)
    return 1.0 * sim / (sum(context1.values()) + sum(context2.values()))

def find_closest_term(term, contexts):
    context1 = contexts[term]
    cosines = [(term2, cosine(term, term2)) for term2 in contexts]
    return sorted(cosines, key=lambda x: x[1], reverse=True)

print '\n'.join('%s  %.2f' % (w, v) for w, v in find_closest_term('email', conte
xts)[:10])
```

```
email  9.18
tell  6.66
let  6.53
reply  6.18
send  5.26
mail  4.70
help  4.38
give  3.86
respond  3.24
post  2.83
```

Google's n-gram data:

http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html

How do we decide when to expand the query?

- Few results returned.
- Query log data
  - Searches where few results are clicked.

CS 429: Information Retrieval

Lecture 11: Probabilistic IR, Part I

Dr. Aron Culotta

Illinois Institute of Technology

Spring 2014

Last time:

- query expansion

This time:

- We'll derive something similar to tfidf using probability theory.
- In the next lectures, we'll extend this to more sophisticated probabilistic ranking models.

Goal: Model the probability that a document is relevant.

$P(R=1| d, q)$

- R: binary variable; 1 if document d is relevant to query q

Probability review

- **Chain rule**

$P(A, B) \equiv P(A \cap B) = P(A|B)P(B) = P(B|A)P(A)$

- **Bayes' rule** (application of the chain rule)

$P(A|B) = \frac{P(A,B)}{P(B)} = \frac{P(B|A)P(A)}{P(B)}$

Using Bayes' Rule:

$P(R=1| d,q) = \frac{P(d|R=1, q)P(R=1|q)}{P(d|q)}$

- Why did we complicate things by turning one term into three?
- It will turn out to be easier to come up with estimates of those three simpler values.

Let's assume for now we can compute $P(R=1|d, q)$.

- How do we use it?

- Rank all documents by probability of relevance: $P(R=1|d,q)$.

Equivalently, we can rank documents by the **odds** of relevance:

$$\frac{P(R=1|d,q)}{P(R=0|d,q)}$$

This will produce the same rankings (since $P(R=1|d,q) + P(R=0|d,q) = 1$).

Using odds, the denominator of Bayes' rule cancels out:

$$\frac{P(d|R=1, q)P(R=1|q)}{P(d|q)} / \frac{P(d|R=0, q)P(R=0|q)}{P(d|q)}$$

$$= \frac{P(d|R=1, q)P(R=1|q)}{ P(d|R=0, q)P(R=0|q) }$$

Furthermore, the ratio $\frac{P(R=1|q)}{P(R=0|q)}$ is constant for every document

- So, it does not affect ranking.

The only terms remaining in our scoring function, then, are:

$$\frac{P(d|R=1, q)}{ P(d|R=0, q)}$$

$P(d|R=1,q)$ is the probability of seeing a relevant document d given query q.

What is this, and how do we estimate it?

Binary Independence Model

Each document represent by a binary term vector.

$$d = \vec{x} = \{x_1, x_2, \ldots, x_n\}$$

where $x_i=1$ if term i appears in d at least once.

- Frequency of term is ignored (for now).
- Word order ignored.

$$P(d|R=1,q) = P(\vec{x}|R=1,q)$$

Conditional independence assumption:

We assume that x_i is conditionally independent of x_j given R,q.

Recall notion of probabilistic independence:

If $A \perp\!\!\!\perp B$, then $P(A,B) = P(A)P(B)$.

- E.g., two flips of a coin

Conditional independence:

If $A \perp\!\!\!\perp B \mid C$, then $P(A,B|C) = P(A|C)P(B|C)$.

- E.g., if a coin might be biased towards heads, then knowing the outcome of the first flip may influence my estimate for the second flip.
- But, knowing for sure the bias of the coin renders the two flips independent

Assuming conditional independence of x given R,q:

$$P(\vec{x}|R=1,q) = P(x_1|R=1,q)P(x_2|R=1,q) \ldots P(x_n|R=1,q)$$

$$= \prod_i P(x_i | R=1, q)$$

We've reduced the problem from "what's the probability of a relevant document" to "what's the probability of a relevant term"

Since each x_i is either 0 or 1, we can re-write this as:

$$P(\vec{x}|R=1,q) = \prod_{i:x_i=1}P(x_i=1|R=1,q) \prod_{i:x_i=0}P(x_i=0|R=1,q)$$

- where $i:x_i=1$ means loop over terms that are in the document.
- and $i:x_i=0$ means loop over terms that are not in the document.

We can make a table of the various probabilities we need to estimate:

		R	
		R=1	R=0
x_i	$x_i=1$	p_i	u_i
	$x_i=0$	$1-p_i$	$1 - u_i$

- p_i: probability of seeing word x_i in a document that is relevant to the query.
- u_i: probability of seeing word x_i in a document that is **not** relevant to the query.
- $1-p_i$: probability of **not** seeing word x_i in a document that is relevant to the query.
- $1-u_i$: probability of **not** seeing word x_i in a document that is **not** relevant to the query.

We can substitute in the values from the table in our calculations:

$$P(\vec{x}|R=1,q) = \prod_{i:x_i=1}P(x_i=1|R=1,q) \prod_{i:x_i=0}P(x_i=0|R=1,q)$$

$$=\prod_{i:x_i=1}p_i \prod_{i:x_i=0}(1-p_i)$$

and for non-relevance:

$$P(\vec{x}|R=0,q) = \prod_{i:x_i=1}P(x_i=1|R=0,q) \prod_{i:x_i=0}P(x_i=0|R=0,q)$$

$$=\prod_{i:x_i=1}u_i \prod_{i:x_i=0}(1-u_i)$$

Our odds ratio can now be written as:

$$\frac{P(d|R=1, q)}{ P(d|R=0, q)} = \prod_{i:x_i=1}\frac{p_i}{u_i} \prod_{i:x_i=0}\frac{(1-p_i)}{(1-u_i)}$$

These products currently loop over all words in a document, even those not in the query.

We make the assumption that the words in the query are the only ones that matter:

$$\frac{P(d|R=1, q)}{ P(d|R=0, q)} = \prod_{i:x_i=q_i=1}\frac{p_i}{u_i} \prod_{i:x_i=0,q_i=1}\frac{(1-p_i)}{(1-u_i)}$$

- where $i:x_i=q_i=1$ means that both the query and document contain term x_i
- and $i:x_i=0,q_i=1$ means that the query contains the term, but the document doesn't

We can cleverly re-arrange these terms to:

$$\frac{P(d|R=1, q)}{ P(d|R=0, q)} = \prod_{i:x_i=q_i=1}\frac{p_i(1-u_i)}{u_i(1-p_i)} \prod_{i:q_i=1}\frac{(1-p_i)}{(1-u_i)}$$

where now the second product is document independent, so we can ignore it from the ranking score.

Finally, we are left as our ranking function:

$$\frac{P(d|R=1, q)}{ P(d|R=0, q)} = \prod_{i:x_i=q_i=1}\frac{p_i(1-u_i)}{u_i(1-p_i)}$$

as a reminder:

- p_i: probability of seeing word x_i in a document that is relevant to the query.
- u_i: probability of seeing word x_i in a document that is **not** relevant to the query.
- $1-p_i$: probability of **not** seeing word x_i in a document that is relevant to the query.
- $1-u_i$: probability of **not** seeing word x_i in a document that is **not** relevant to the query.

For numerical reasons, we typically take the log of this, which is called the *Retrieval Status Value (RSV)*:

$$RSV(d) = \log \prod_{i:x_i=q_i=1}\frac{p_i(1-u_i)}{u_i(1-p_i)} = \sum_{i:x_i=q_i=1}\log\frac{p_i(1-u_i)}{u_i(1-p_i)}$$

$$= \sum_{i:x_i=q_i=1} \log \frac{p_i}{1-p_i} + \log\frac{1-u_i}{u_i}$$

$RSV(d)$ will be used to rank each document d

Estimation

We still need to estimate

- p_i: probability of seeing word x_i in a document that is relevant to the query.
- u_i: probability of seeing word x_i in a document that is **not** relevant to the query.

Given a set of documents $D=\{d_1 \ldots d_N\}$, what is the probability of seeing word x_i?

Just count:

(number of documents containing x_i) / N

To estimate u_i (prob. of x_i in a non-relevant document)

- Given a very large collection, most documents are not relevant to a query.
- So, we can estimate u_i as simply the probability of seeing x_i in *any* document...
- Which is just *document frequency*!

$$u_i = \frac{df_i}{N}$$

Thus

$$\log\frac{1-u_i}{u_i} = \log\frac{1-\frac{df_i}{N}}{\frac{df_i}{N}} = \log \frac{N-df_i}{df_i} \approx \log \frac{N}{df_i}$$

Plugging this back into the RSV equation:

$$RSV(d) = \sum_{i:x_i=q_i=1} \log \frac{p_i}{1-p_i} + \log\frac{1-u_i}{u_i}$$

$$= \sum_{i:x_i=q_i=1} \log \frac{p_i}{1-p_i} + \log\frac{N}{df_i}$$

How should we estimate p_i? (probability of seeing word x_i in a relevant document)

1. Ignore it! Then, we rank documents only by idf
2. Ask a human for relevance judgements. (*expensive*)
3. Use click log data.
4. Assume the top k retrieved results are relevant, then estimate from that (psuedo-relevance feedback)

Psuedo-relevance Feedback

1. Initialize $p_i \leftarrow 0.5$.
2. Rank documents by $RSV(d)$.
3. Let V be the top k documents. We pretend these are all relevant.
4. We then update p_i to be the proportion of elements of V that contain term x_i

    - $p_i =$ (number of documents containing x_i in V) / $|V|$

We can iterate steps 2-4 until ranking stops changing.

CS 429: Information Retrieval

Lecture 12: Probabilistic IR, Part II

Dr. Aron Culotta

Illinois Institute of Technology

Spring 2014

Last time:

- Probabilistic justification for idf weights

This time:

- Include term frequency
- Normalize by document length
  - BM25 (Okapi)
- Extensions to fielded search

RSV (Retrieval Status Value)

$$RSV(d) = \sum_{i:x_i=q_i=1} \log \frac{N}{df_i}$$

- sum of (log of) inverse document frequencies for each matching query term

Assumptions of RSV:

- a Boolean representation of documents/queries/relevance
- term independence
- terms not in the query don't affect the outcome
- document relevance values are independent

Multiplying by term frequency

$$RSV_{tf}(d) = \sum_{i:x_i=q_i=1} \log \Big[\frac{N}{df_i}\Big] \times tf_i$$

- but, want tf scores to be bounded, so

$$RSV_{tf}(d) = \sum_{i:x_i=q_i=1} \log \Big[\frac{N}{df_i} \Big] \times \frac{(k+1)tf_i}{k + tf_i}$$

What is the effect of k?

In [89]:

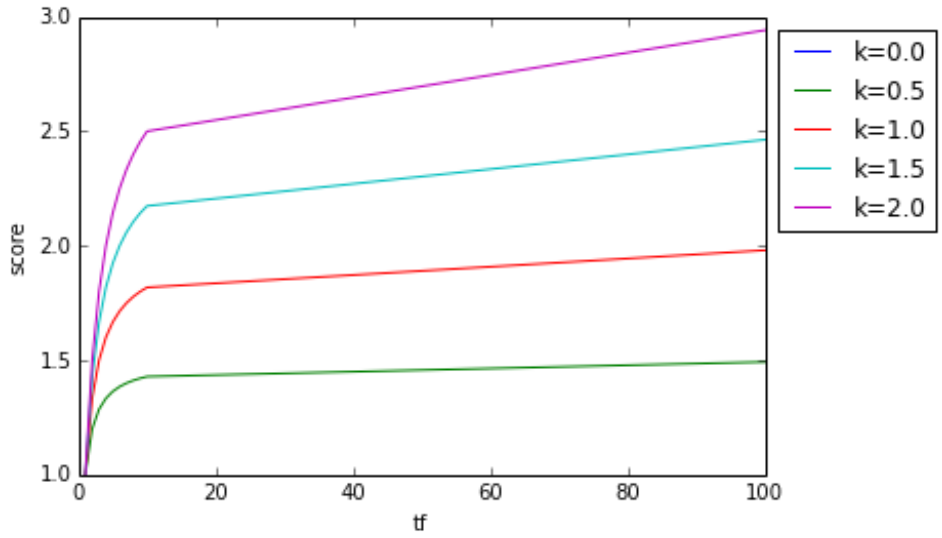
```
# Plot tf score as k varies
```

```
# plot tf score as k varies.

def tfscore(k, tf):
    return (k+1)*tf / (k+tf)

% pylab inline
ks = [0., .5, 1., 1.5, 2.0]
tfs = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 100]
for k in ks:
    plot(tfs, [tfscore(k, tf) for tf in tfs], label='k=' + str(k))
xlabel('tf')
ylabel('score')
legend(loc='upper left', bbox_to_anchor=(1,1))
#ylim((0.9,2.6))
```

```
Populating the interactive namespace from numpy and matplotlib
```

Out[89]:

```
<matplotlib.legend.Legend at 0x114a89d50>
```



larger k \rightarrow slower "saturation" of score.

- like normal tf, but bounded in (1,2)

Length normalization

Longer document \rightarrow larger values of tf_i

- How can we level the playing field between long and short documents?
- Perhaps we should prefer longer documents?

Document length

l(d) = \sum_{i \in V} tf_i

- Average document length

\mu_l = \frac{1}{N}\sum_d l(d)

Want to allow different levels of length normalization:

$$B = (1-b) + b \frac{l(d)}{\mu_l}$$

$$0 \le b \le 1$$

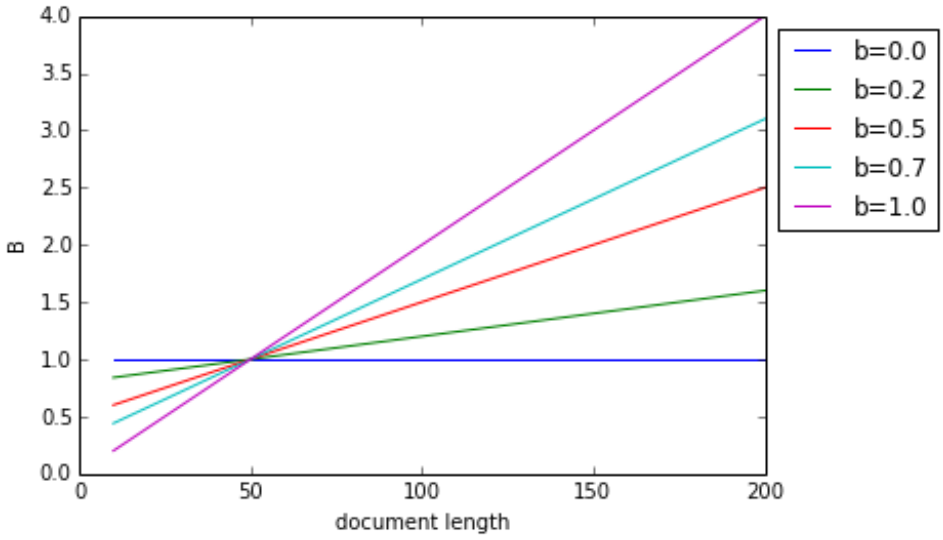
0= no normalization 1= full normalization

In [87]:

```
# What is the effect of b on the length normalization term B?
def len_norm(b, length, avg_length):
    return (1-b) + b * length / avg_length

bs = [0., 0.2, 0.5, 0.7, 1.]
lengths = [10, 30, 50, 100, 150, 200]
avg_length = 50
for b in bs:
    plot(lengths,
         [len_norm(b, length, avg_length) for length in lengths],
         label='b=' + str(b))
xlabel('document length')
ylabel('B')
legend(loc='upper left', bbox_to_anchor=(1,1))
```

Out[87]:

```
<matplotlib.legend.Legend at 0x114b52cd0>
```



- large $l(d) \rightarrow$ large B.
- large $b \rightarrow$ large B
- $l(d) < \mu_l \rightarrow 0 < B < 1$
- $l(d) > \mu_l \rightarrow B > 1$

To adjust tf using l(d):

$$tf_i' = \frac{tf_i}{B}$$

Replacing in RSV definition:

$$\begin{align} BM25(d) & = & \sum_{i:x_i=q_i=1} \log \Big[\frac{N}{df_i}\Big] \times \frac{(k+1)tf_i'}{k + tf_i'}\\ & = & \sum_{i:x_i=q_i=1} \log \Big[\frac{N}{df_i}\Big] \times \frac{(k+1)tf_i}{k((1-b) + b \frac{l(d)}{\mu_l}) + tf_i}\\ & = & \sum_{i:x_i=q_i=1} \log \Big[\frac{N}{df_i}\Big] \times \frac{(k+1)tf_i}{Bk + tf_i} \end{align}$$

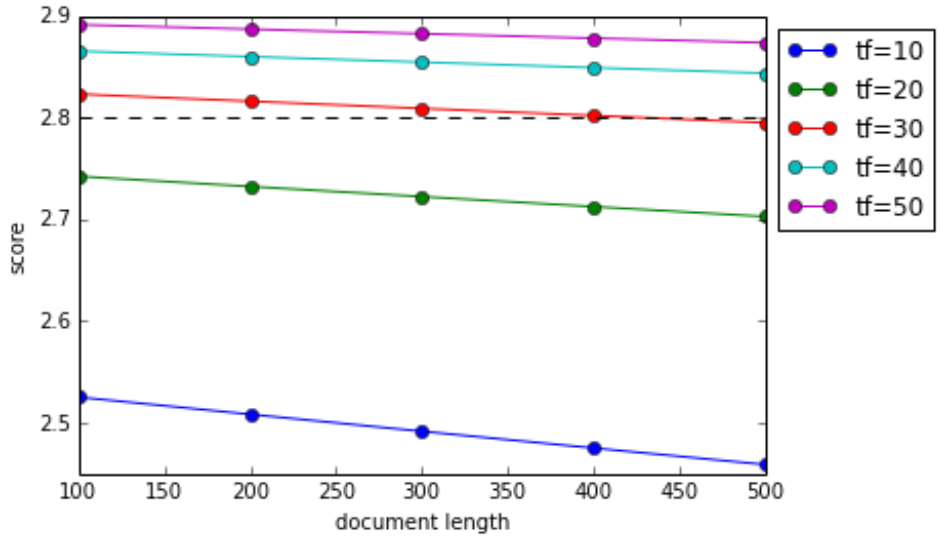
BM25: "Best Match" 25 system (aka "Okapi")

In [93]:

```
# How does score vary with tf and document length?
def score(k, b, tf, length, m_length):
    return (k + 1) * tf / (k * ((1 - b) + b * length / m_length) + tf)

tfs = [10, 20, 30, 40, 50]
lengths = [100, 200, 300, 400, 500]
m_length = 250
k = 2.0  # tf strength
b = 0.1  # length normalization strength

for tf in tfs:
    plot(lengths,
        [score(k, b, tf, length, m_length) for length in lengths],
        'o-', label='tf=' + str(tf))
plot (lengths, [2.8] * len(lengths), 'k--')
legend(loc='upper left', bbox_to_anchor=(1,1))
xlabel('document length')
ylabel('score')
```

Out[93]:

```
<matplotlib.text.Text at 0x114949690>
```



Compare:

- two documents with same tf but different lengths
- two documents with same length but different tf

Fielded search

How can we combine scores from multiple fields? (title, author, etc.)

Simple way: Score each field separately and take the average.

- Assumes each field the same

Instead, take weighted average:

tf^F_i = \sum_{z=1}^Z v_z tf_{zi}

$l^F(d) = \sum_{z=1}^Z v_z l(d_z)$

$\mu^F_l = \frac{1}{N} \sum_d l^F(d)$

- Z: number of fields
- tf_{zi}: tf of term i in field z
- $l(d_z)$: length of field z
- v_z: weight for field z

$$\begin{align} BM25F(d) = \sum_{i:x_i=q_i=1} \log \Big[\frac{N}{df_i}\Big] \times \frac{(k+1)tf^F_i}{k((1-b) + b \frac{l^F(d)}{\mu^F_l}) + tf^F_i} \end{align}$$

Could also have different b for each field. Why?

Exercise

There are four documents in the world, with the following term frequencies:

- d_1: {dog: 4, cat: 6, the: 10}
- d_2: {dog: 2, the: 8}
- d_3: {zebra: 20}
- d_4: {zebra: 30}

Consider the query **(the dog)**. Let $b=1$ and $k=2$. Fill in the values in this table (showing your work):

	RSV	BM25
d_1		
d_2		

$$\begin{align} BM25(d) & = & \sum_{i:x_i=q_i=1} \log \Big[\frac{N}{df_i}\Big] \times \frac{(k+1)tf_i}{k((1-b) + b \frac{l(d)}{\mu_l}) + tf_i}\\ \end{align}$$

$$\begin{align} RSV(d) & = & \sum_{i:x_i=q_i=1} \log \frac{N}{df_i} \end{align}$$

CS 429: Information Retrieval

Lecture 13: Language Models, Part I

Dr. Aron Culotta

Illinois Institute of Technology

Spring 2014

Last week...

We ranked documents by:

$P(R=1|d, q)$

where $R=1$ means document d is relevant to query q.

Problem: hard to estimate this value without d,q pairs labeled with relevance.

Language Models

Idea:

Rank documents by:

$P(q|d)$

The probability that the process that generated d would also generate q.

No variable for relevance.

Generative models

- Each document is a list of strings from a language.
- Consider all the possible documents the author could have written
  - How many of them would contain the term "zebra"?
- Consider the query q
  - What is the probability that the author of document d would have written down q?
  - $P(q|M_d)$

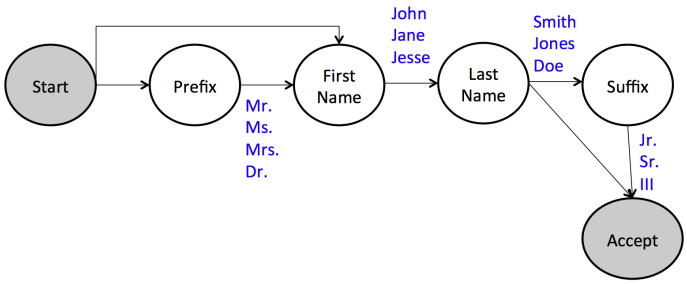
Finite State Machine

Let a *language* L be a set of documents $\{d_1 \ldots d_n\}$.

A finite-state machine M_L accepts a document d as input and outputs "yes" if $d \in L$; otherwise it outputs "no."

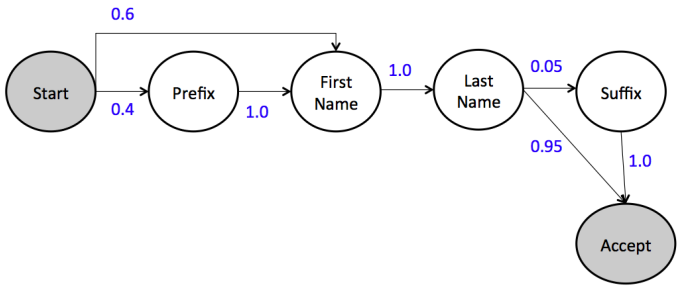
M_L consists of:

- a set of **states** $S = \{s_1 \ldots s_m\}$
- an input **vocabulary** V, a finite set of acceptable terms
- a **transition function** $\delta : V \times S \mapsto S$
  - When in state s_i, if term $w \in V$ is read, the state changes to s_j



- Mr. John Smith Jr. start $\rightarrow$ prefix $\xrightarrow{Mr.}$ first $\xrightarrow{John}$ last $\xrightarrow{Smith}$ suffix $\xrightarrow{Jr.}$ accept
- Jane Doe
- Mr. Jr.

Weighted Finite State Machine



- $P($Mr. John Smith Jr.$)= 0.4 * 1.0 * 1.0 * .05 * 1.0 = 0.02$
- $P($Jane Doe$) = 0.6 * 1.0 * 0.95 = 0.57$
- $P($Mr. Jr.$) = 0.0$

Generative Model

Rather than simply assigning probabilities to documents, we can use a weighted finite state machine to **generate** documents.

In [119]:

```
# Generate names.
# Assume all words are equally likely, but state transitions follow previous FS
M.

prefixes = ['Mr. ', 'Ms. ', 'Mrs. ', 'Dr. ']
firsts = ['John ', 'Jane ', 'Jesse ']
lasts = ['Smith ', 'Jones ', 'Doe ']
suffixes = ['Jr. ', 'Sr. ', 'III ']
```

```
def sample(alist):
    """ Sample an element of a list. """
    return alist[random.randint(0, len(alist) - 1)]

import random
num_documents = 20
for i in range(num_documents):
    doc = ''
    if random.random() <= 0.4:  # prefix
        doc += sample(prefixes)
    doc += sample(firsts) + sample(lasts)
    if random.random() <= .05:  # suffix
        doc += sample(suffixes)
    print doc
```

```
John Jones
Jesse Doe
Ms. Jane Smith
Mrs. Jane Doe
Jane Smith
Jesse Doe
John Smith
Jane Jones
John Doe
Mrs. Jane Doe
John Doe
Mrs. John Smith
Mr. Jane Doe
Jesse Doe
Jesse Smith
Mrs. Jesse Doe
Mr. John Doe
Jane Doe
Jane Doe
Jane Jones
```

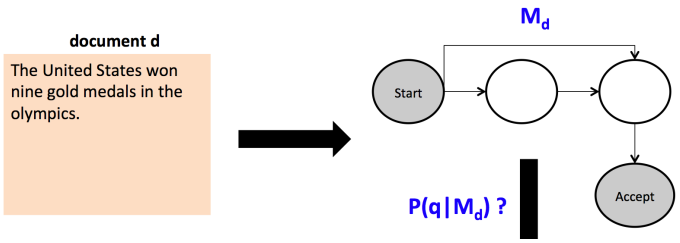
Language Model

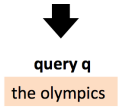
A weighted finite state machine that can

- generate documents
- generate queries
- assign probabilities to documents/queries

Idea:

- Construct a language model M_d for each document d.
- For each query q, compute the probability that M_d generated q: $P(q|M_d)$
- Rank documents by $P(q|M_d)$.

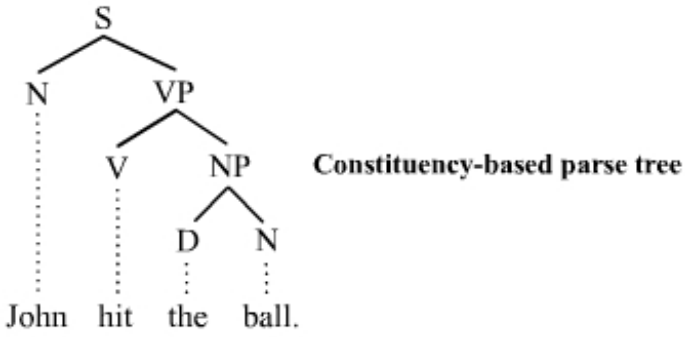




How can we construct a language model from a document?

Long history in natural language processing:

- parse trees



Source: Wikipedia

- sentence generators

But, grammar has little effect on information retrieval.

- queries are rarely grammatical

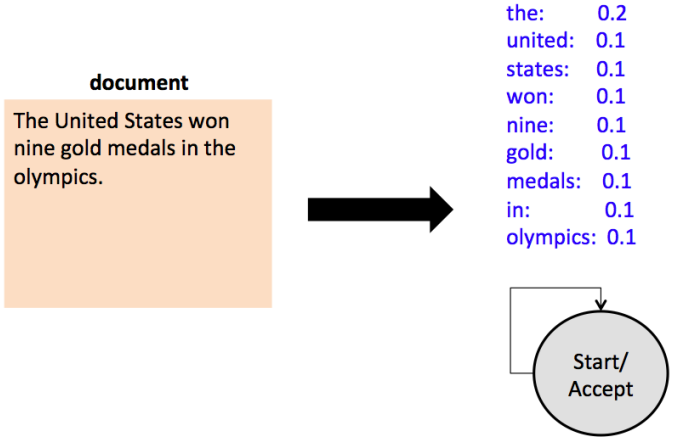
Unigram Language Models

- Ignore word order.
- Generate each word independently.

$\begin{align} P(q|M_d) = \prod_{t \in q} P(t|M_d) = \prod_{t \in q} \frac{tf_{t, d}}{L_d} \end{align}$

- $q:$ query consisting of terms t
- $M_d:$ language model for document d
- $tf_{t, d}:$ frequency of term t in document d
- $L_d:$ number of tokens in d

Unigram Language Models



In [120]:

```
from collections import Counter

def doc2model(doc):
    """ Convert a document d into a language model M_d. """
    counts = Counter(doc)
    for term in counts:
        counts[term] /= 1. * len(doc)
    return counts

m_d = doc2model(['the', 'united', 'states', 'won', 'nine', 'gold', 'medals', 'in
', 'the', 'olympics'])
print m_d
```

```
Counter({'the': 0.2, 'united': 0.1, 'gold': 0.1, 'states': 0.1, 'won': 0.1, 'ni
ne': 0.1, 'in': 0.1, 'olympics': 0.1, 'medals': 0.1})
```

In [125]:

```
import numpy as np

def sample_from_model(m_d, length):
    """ Sample length words from language model m_d. """
    counts = np.random.multinomial(length, m_d.values(), size=1)[0]
    words = []
    for i, count in enumerate(counts):
        words.extend(count * [m_d.keys()[i]])
    return words

print sample_from_model(m_d, 10)
```

```
['united', 'gold', 'won', 'won', 'olympics', 'olympics', 'the', 'medals', 'meda
ls', 'medals']
```

In [127]:

```
def pr_q_given_m(q, m_d):
    """ Compute P(q|M_d), the probability of language model M_d generating quer
y q. """
    product = 1.
    for qi in q:
        product *= m_d[qi]
    return product

print 'Pr([the, olympics] | d)=', pr_q_given_m(['the', 'olympics'], m_d)
print 'Pr([united, states] | d)=', pr_q_given_m(['united', 'states'], m_d)
print 'Pr([olympics, united, states] | d)=', pr_q_given_m(['olympics', 'united'
, 'states'], m_d)
```

```
Pr([the, olympics] | d)= 0.02
Pr([united, states] | d)= 0.01
Pr([olympics, united, states] | d)= 0.001
```

In [128]:

```
def doc2ngram_model(doc, n):
    """ Convert a document d into a language model M_d. """
```

```
    counts = Counter()
    for i in range(len(doc) - 1):
        counts.update([' '.join(doc[i:i+n]) for i in range(len(doc) - n + 1)])
    length = sum(counts.values())
    for term in counts:
        counts[term] /= 1. * length
    return counts

m_d2 = doc2ngram_model(['the', 'united', 'states', 'won', 'nine', 'gold', 'medal
s', 'in', 'the', 'olympics'], 2)
m_d3 = doc2ngram_model(['the', 'united', 'states', 'won', 'nine', 'gold', 'medal
s', 'in', 'the', 'slalom', 'in', 'the', 'olympics'], 2)

print 'm_d2:', m_d2

print '\nm_d3', m_d3
```

```
m_d2: Counter({'states won': 0.1111111111111111, 'won nine':
0.1111111111111111, 'the olympics': 0.1111111111111111, 'united states': 0.1111
111111111111, 'gold medals': 0.1111111111111111, 'the united':
0.1111111111111111, 'medals in': 0.1111111111111111, 'in the':
0.1111111111111111, 'nine gold': 0.1111111111111111})

m_d3 Counter({'in the': 0.16666666666666666, 'states won': 0.08333333333333333,
'the slalom': 0.08333333333333333, 'won nine': 0.08333333333333333, 'the olympi
cs': 0.08333333333333333, 'united states': 0.08333333333333333, 'gold medals':
0.08333333333333333, 'the united': 0.08333333333333333, 'medals in':
0.08333333333333333, 'nine gold': 0.08333333333333333, 'slalom in':
0.08333333333333333})
```

In [129]:

```
sample_from_model(m_d3, 10)
```

Out[129]:

```
['states won',
 'the slalom',
 'gold medals',
 'the united',
 'the united',
 'medals in',
 'in the',
 'in the',
 'in the',
 'slalom in']
```

In [130]:

```
m_d4 = doc2ngram_model(['the', 'united', 'states', 'won', 'nine', 'gold', 'medal
s', 'in', 'the', 'olympics'], 4)
sample_from_model(m_d4, 10)
```

Out[130]:

```
['nine gold medals in',
 'nine gold medals in',
 'gold medals in the',
 'gold medals in the',
 'won nine gold medals',
 'won nine gold medals',
```

```
 'the united states won',
 'medals in the olympics',
 'states won nine gold',
 'states won nine gold']
```

Why not just set $n=10000$?

In [110]:

```
# 4-gram model:
print 'Pr([the olympics] | m_d4)=', pr_q_given_m(['the olympics'], m_d4)
```

```
Pr([the olympics] | m_d4)= 0.0
```

In [111]:

```
# 2-gram model
print 'Pr([the olympics] | m_d2)=', pr_q_given_m(['the olympics'], m_d2)
```

```
Pr([the olympics] | m_d2)= 0.111111111111
```

In [112]:

```
# Even for unigram model
print 'Pr([the, olympics, zebra] | m_d)=', pr_q_given_m(['the', 'olympics', 'ze
bra'], m_d)
```

```
Pr([the, olympics, zebra] | m_d)= 0.0
```

If a query does not appear in document d, then $P(q|M_d)=0$.

- Want to allow some chance that a word not in d will appear.

In [115]:

```
def doc2model_smooth(doc, smooth_term, vocab):
    """ Convert a document d into a language model M_d. """
    counts = Counter(doc)
    for term in vocab:
        counts[term] = (counts[term] + smooth_term) / (1. * len(doc) +
smooth_term * len(vocab))
    return counts

vocab = ['the', 'united', 'states', 'won', 'nine', 'gold', 'medals', 'in', 'olym
pics', 'zebra']
m_d_smooth1 = doc2model_smooth(['the', 'united', 'states', 'won', 'nine', 'gold'
, 'medals', 'in', 'the', 'olympics'], 1, vocab)
m_d_smooth10 = doc2model_smooth(['the', 'united', 'states', 'won', 'nine',
'gold', 'medals', 'in', 'the', 'olympics'], 10, vocab)
print 'unsmoothed model:', m_d
print '\nsmoothed model1:', m_d_smooth1
print '\nsmoothed model10:', m_d_smooth10
```

```
unsmoothed model: Counter({'the': 0.2, 'united': 0.1, 'gold': 0.1, 'states': 0.
1, 'won': 0.1, 'nine': 0.1, 'in': 0.1, 'olympics': 0.1, 'medals': 0.1})

smoothed model1: Counter({'the': 0.15, 'united': 0.1, 'gold': 0.1, 'states': 0.
1, 'won': 0.1, 'nine': 0.1, 'in': 0.1, 'olympics': 0.1, 'medals': 0.1, 'zebra':
0.05})
```

```
smoothed model10: Counter({'the': 0.10909090909090909, 'united': 0.1, 'gold': 0
.1, 'states': 0.1, 'won': 0.1, 'nine': 0.1, 'in': 0.1, 'olympics': 0.1, 'medals
': 0.1, 'zebra': 0.09090909090909091})
```

In [114]:

```
print 'Pr([the, olympics, zebra] | m_d_smooth1)=', pr_q_given_m(['the', 'olympi
cs', 'zebra'], m_d_smooth1)
print 'Pr([the, olympics, zebra] | m_d_smooth10)=', pr_q_given_m(['the', 'olymp
ics', 'zebra'], m_d_smooth10)
```

```
Pr([the, olympics, zebra] | m_d_smooth1)= 0.00075
Pr([the, olympics, zebra] | m_d_smooth10)= 0.00099173553719
```

Smoothed Language Model

(Laplace smoothing)

$\begin{align} P_{smooth}(q|M_d) = \prod_{t \in q} P(t|M_d) = \prod_{t \in q} \frac{tf_{t, d} + \epsilon}{L_d + V\epsilon} \end{align}$

- $q:$ query consisting of terms t
- $M_d:$ language model for document d
- $tf_{t, d}:$ frequency of term t in document d
- $L_d:$ number of tokens in d
- $\epsilon:$ amount to smooth
- V: vocabulary size

CS 429: Information Retrieval

Lecture 14: Language Models, Part II

Dr. Aron Culotta

Illinois Institute of Technology

Spring 2014

Language Models

Idea:

Rank documents by:

$P(q|d)$

The probability that the process that generated d would also generate q.

No variable for relevance.

Generative models

- Each document is a list of strings from a language.
- Consider all the possible documents the author could have written
  - How many of them would contain the term "zebra"?
- Consider the query q
  - What is the probability that the author of document d would have written down q?
  - $P(q|M_d)$

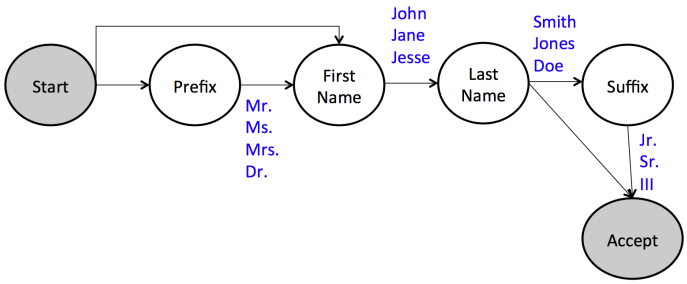
Finite State Machine

Let a *language* L be a set of documents $\{d_1 \ldots d_n\}$.

A finite-state machine M_L accepts a document d as input and outputs "yes" if $d \in L$; otherwise it outputs "no."

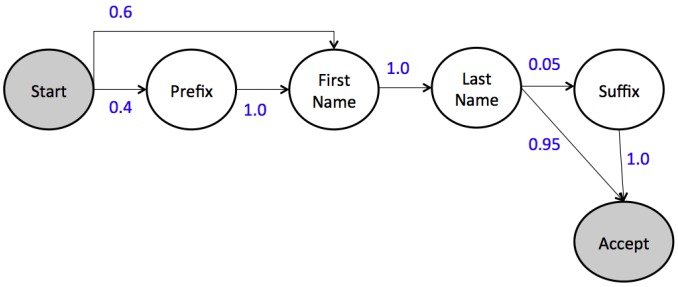
M_L consists of:

- a set of **states** $S = \{s_1 \ldots s_m\}$
- an input **vocabulary** V, a finite set of acceptable terms
- a **transition function** $\delta : V \times S \mapsto S$
  - When in state s_i, if term $w \in V$ is read, the state changes to s_j



- Mr. John Smith Jr. start $\rightarrow$ prefix $\xrightarrow{Mr.}$ first $\xrightarrow{John}$ last $\xrightarrow{Smith}$ suffix $\xrightarrow{Jr.}$ accept
- Jane Doe
- Mr. Jr.

Weighted Finite State Machine



- $P($Mr. John Smith Jr.$)= 0.4 * 1.0 * 1.0 * .05 * 1.0 = 0.02$
- $P($Jane Doe$) = 0.6 * 1.0 * 0.95 = 0.57$
- $P($Mr. Jr.$) = 0.0$

Generative Model

Rather than simply assigning probabilities to documents, we can use a weighted finite state machine to **generate** documents.

In [2]:

```
# Generate names.
# Assume all words are equally likely, but state transitions follow previous FS
M.

prefixes = ['Mr. ', 'Ms. ', 'Mrs. ', 'Dr. ']
firsts = ['John ', 'Jane ', 'Jesse ']
lasts = ['Smith ', 'Jones ', 'Doe ']
suffixes = ['Jr. ', 'Sr. ', 'III ']

def sample(alist):
    """ Sample an element of a list. """
    return alist[random.randint(0, len(alist) - 1)]

import random
num_documents = 20
for i in range(num_documents):
    doc = ''
    if random.random() <= 0.4:  # prefix
        doc += sample(prefixes)
```

```
    doc += sample(firsts) + sample(lasts)
    if random.random() <= .05:  # suffix
        doc += sample(suffixes)
    print doc
```

```
Jesse Doe
Mrs. Jesse Jones
Dr. Jane Doe
Jesse Jones
Ms. Jane Jones
Jane Doe
Mr. Jesse Jones
Jesse Doe
Mr. John Smith
Jesse Jones
Jesse Smith
Dr. Jesse Jones
Jesse Smith
Jane Smith
Ms. Jesse Smith
Mr. John Doe
Dr. John Doe
Mrs. John Smith
Dr. Jane Doe Sr.
Jesse Jones
```

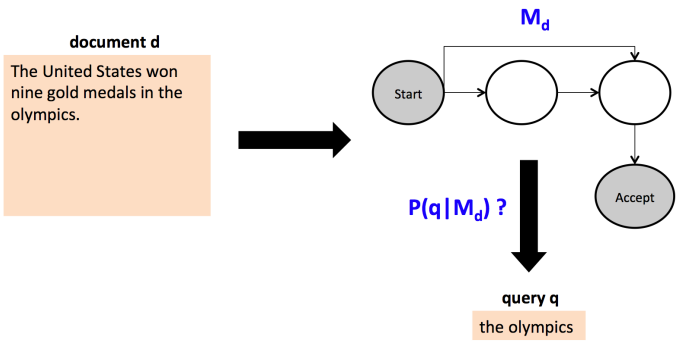
Language Model

A weighted finite state machine that can

- generate documents
- generate queries
- assign probabilities to documents/queries

Idea:

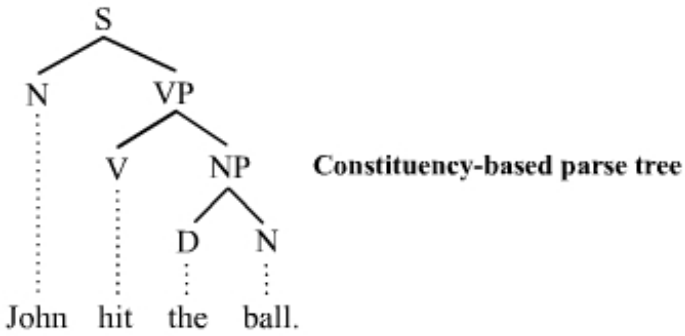
- Construct a language model M_d for each document d.
- For each query q, compute the probability that M_d generated q: $P(q|M_d)$
- Rank documents by $P(q|M_d)$.



How can we construct a language model from a document?

Long history in natural language processing:

- parse trees



Source: Wikipedia

- sentence generators

But, grammar has little effect on information retrieval.

- queries are rarely grammatical

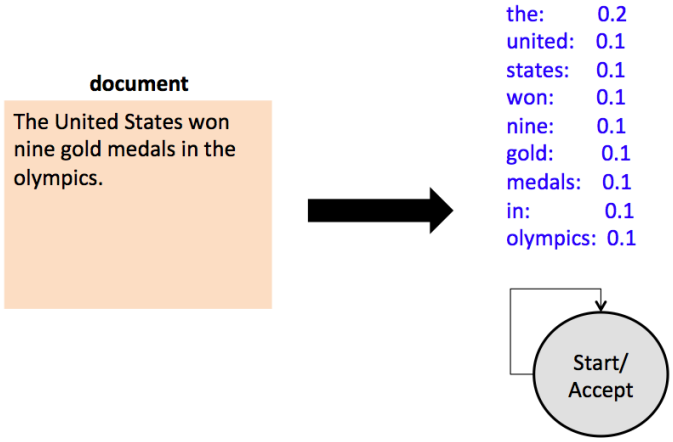
Unigram Language Models

- Ignore word order.
- Generate each word independently.

$\begin{align} P(q|M_d) = \prod_{t \in q} P(t|M_d) = \prod_{t \in q} \frac{tf_{t, d}}{L_d} \end{align}$

- $q:$ query consisting of terms t
- $M_d:$ language model for document d
- $tf_{t, d}:$ frequency of term t in document d
- $L_d:$ number of tokens in d

Unigram Language Models



In [3]:

```
from collections import Counter

def doc2model(doc):
    """ Convert a document d into a language model M_d. """
    counts = Counter(doc)
    for term in counts:
        counts[term] /= 1. * len(doc)
    return counts
```

```
m_d = doc2model(['the', 'united', 'states', 'won', 'nine', 'gold', 'medals', 'in
', 'the', 'olympics'])
print m_d
```

```
Counter({'the': 0.2, 'united': 0.1, 'gold': 0.1, 'states': 0.1, 'won': 0.1, 'ni
ne': 0.1, 'in': 0.1, 'olympics': 0.1, 'medals': 0.1})
```

In [4]:

```
import numpy as np

def sample_from_model(m_d, length):
    """ Sample length words from language model m_d. """
    counts = np.random.multinomial(length, m_d.values(), size=1)[0]
    words = []
    for i, count in enumerate(counts):
        words.extend(count * [m_d.keys()[i]])
    return words

print sample_from_model(m_d, 10)
```

```
['united', 'united', 'united', 'united', 'states', 'won', 'nine', 'nine', 'the'
, 'medals']
```

In [5]:

```
def pr_q_given_m(q, m_d):
    """ Compute P(q|M_d), the probability of language model M_d generating quer
y q. """
    product = 1.
    for qi in q:
        product *= m_d[qi]
    return product

print 'Pr([the, olympics] | d)=', pr_q_given_m(['the', 'olympics'], m_d)
print 'Pr([united, states] | d)=', pr_q_given_m(['united', 'states'], m_d)
print 'Pr([olympics, united, states] | d)=', pr_q_given_m(['olympics', 'united'
, 'states'], m_d)
```

```
Pr([the, olympics] | d)= 0.02
Pr([united, states] | d)= 0.01
Pr([olympics, united, states] | d)= 0.001
```

In [6]:

```
def doc2ngram_model(doc, n):
    """ Convert a document d into a language model M_d. """
    counts = Counter()
    for i in range(len(doc) - 1):
        counts.update([' '.join(doc[i:i+n]) for i in range(len(doc) - n + 1)])
    length = sum(counts.values())
    for term in counts:
        counts[term] /= 1. * length
    return counts

m_d2 = doc2ngram_model(['the', 'united', 'states', 'won', 'nine', 'gold', 'medal
s', 'in', 'the', 'olympics'], 2)
```

```
m_d3 = doc2ngram_model(['the', 'united', 'states', 'won', 'nine', 'gold', 'medal
s', 'in', 'the', 'slalom', 'in', 'the', 'olympics'], 2)

print 'm_d2:', m_d2

print '\nm_d3', m_d3
```

```
m_d2: Counter({'states won': 0.1111111111111111, 'won nine':
0.1111111111111111, 'the olympics': 0.1111111111111111, 'united states': 0.1111
111111111111, 'gold medals': 0.1111111111111111, 'the united':
0.1111111111111111, 'medals in': 0.1111111111111111, 'in the':
0.1111111111111111, 'nine gold': 0.1111111111111111})

m_d3 Counter({'in the': 0.16666666666666666, 'states won': 0.08333333333333333,
'the slalom': 0.08333333333333333, 'won nine': 0.08333333333333333, 'the olympi
cs': 0.08333333333333333, 'united states': 0.08333333333333333, 'gold medals':
0.08333333333333333, 'the united': 0.08333333333333333, 'medals in':
0.08333333333333333, 'nine gold': 0.08333333333333333, 'slalom in':
0.08333333333333333})
```

In [7]:

```
sample_from_model(m_d3, 10)
```

Out[7]:

```
['states won',
 'states won',
 'the slalom',
 'the slalom',
 'won nine',
 'the olympics',
 'the olympics',
 'the united',
 'medals in',
 'in the']
```

In [8]:

```
m_d4 = doc2ngram_model(['the', 'united', 'states', 'won', 'nine', 'gold', 'medal
s', 'in', 'the', 'olympics'], 4)
sample_from_model(m_d4, 10)
```

Out[8]:

```
['nine gold medals in',
 'nine gold medals in',
 'won nine gold medals',
 'the united states won',
 'medals in the olympics',
 'states won nine gold',
 'states won nine gold',
 'states won nine gold',
 'united states won nine',
 'united states won nine']
```

Why not just set $n=10000$?

In [9]:

```
# 4-gram model:
```

```
print 'Pr([the olympics] | m_d4)=', pr_q_given_m(['the olympics'], m_d4)
```

```
Pr([the olympics] | m_d4)= 0.0
```

In [10]:

```
# 2-gram model
print 'Pr([the olympics] | m_d2)=', pr_q_given_m(['the olympics'], m_d2)
```

```
Pr([the olympics] | m_d2)= 0.111111111111
```

In [11]:

```
# Even for unigram model
print 'Pr([the, olympics, zebra] | m_d)=', pr_q_given_m(['the', 'olympics', 'ze
bra'], m_d)
```

```
Pr([the, olympics, zebra] | m_d)= 0.0
```

If a query does not appear in document d, then $P(q|M_d)=0$.

- Want to allow some chance that a word not in d will appear.

Smoothed Language Model

(Laplace smoothing)

$\begin{align} P_{smooth}(q|M_d) = \prod_{t \in q} P(t|M_d) = \prod_{t \in q} \frac{tf_{t, d} + \epsilon}{L_d + V\epsilon} \end{align}$

- $q:$ query consisting of terms t
- $M_d:$ language model for document d
- $tf_{t, d}:$ frequency of term t in document d
- $L_d:$ number of tokens in d
- $\epsilon:$ amount to smooth
- V: vocabulary size

In [18]:

```
def doc2model_smooth(doc, smooth_term, vocab):
    """ Convert a document d into a language model M_d. """
    counts = Counter(doc)
    for term in vocab:
        counts[term] = (counts[term] + smooth_term) / (1. * len(doc) +
smooth_term * len(vocab))
    return counts

vocab = ['the', 'united', 'states', 'won', 'nine', 'gold', 'medals', 'in', 'olym
pics', 'zebra', 'a']
m_d_smooth1 = doc2model_smooth(['the', 'united', 'states', 'won', 'nine', 'gold'
, 'medals', 'in', 'the', 'olympics'], 1, vocab)
m_d_smooth10 = doc2model_smooth(['the', 'united', 'states', 'won', 'nine',
'gold', 'medals', 'in', 'the', 'olympics'], 10, vocab)
print 'unsmoothed model:', m_d
print '\nsmoothed model1:', m_d_smooth1
print '\nsmoothed model10:', m_d_smooth10
```

```
unsmoothed model: Counter({'the': 0.2, 'united': 0.1, 'gold': 0.1, 'states': 0.
1, 'won': 0.1, 'nine': 0.1, 'in': 0.1, 'olympics': 0.1, 'medals': 0.1})

smoothed model1: Counter({'the': 0.14285714285714285, 'united':
0.09523809523809523, 'gold': 0.09523809523809523, 'states':
0.09523809523809523, 'won': 0.09523809523809523, 'nine': 0.09523809523809523, '
in': 0.09523809523809523, 'olympics': 0.09523809523809523, 'medals':
0.09523809523809523, 'a': 0.047619047619047616, 'zebra': 0.047619047619047616})

smoothed model10: Counter({'the': 0.1, 'united': 0.09166666666666666, 'gold': 0
.09166666666666666, 'states': 0.09166666666666666, 'won': 0.09166666666666666,
'nine': 0.09166666666666666, 'in': 0.09166666666666666, 'olympics':
0.09166666666666666, 'medals': 0.09166666666666666, 'a': 0.08333333333333333, '
zebra': 0.08333333333333333})
```

In [19]:

```
print 'Pr([the, olympics, zebra] | m_d_smooth1)=', pr_q_given_m(['the', 'olympi
cs', 'zebra'], m_d_smooth1)
print 'Pr([the, olympics, zebra] | m_d_smooth10)=', pr_q_given_m(['the', 'olymp
ics', 'zebra'], m_d_smooth10)
```

```
Pr([the, olympics, zebra] | m_d_smooth1)= 0.000647878198899
Pr([the, olympics, zebra] | m_d_smooth10)= 0.000763888888889
```

Problem with Laplace smoothing:

- Assumes that all unseen words are equally likely.
  - Effectively adds ϵ occurrences to every document.

In [21]:

```
print 'Pr([the, olympics, zebra] | m_d_smooth10)=', pr_q_given_m(['the', 'olymp
ics', 'zebra'], m_d_smooth10)
print 'Pr([the, olympics, a] | m_d_smooth10)=', pr_q_given_m(['the', 'olympics'
, 'a'], m_d_smooth10)
```

```
Pr([the, olympics, zebra] | m_d_smooth10)= 0.000763888888889
Pr([the, olympics, a] | m_d_smooth10)= 0.000763888888889
```

- $d_1:$ the, cat
- $d_2:$ dog, cat

- $q:$ dog, the

Should return d_2.

But, Laplace smoothing means missing the word "dog" is just as bad as missing the word "the".

$\begin{align} P_{smooth}(q|M_d) = \prod_{t \in q} P(t|M_d) = \prod_{t \in q} \frac{tf_{t, d} + \epsilon}{L_d + V\epsilon} \end{align}$

$\begin{align} P_{smooth}(q|M_{d_1}) = P(dog|M_{d_1}) * P(the|M_{d_1}) = \frac{\epsilon}{2 + V\epsilon} * \frac{1 + \epsilon}{2 + V\epsilon} \end{align}$

$\begin{align} P_{smooth}(q|M_{d_2}) = P(dog|M_{d_2}) * P(the|M_{d_2}) = \frac{1 + \epsilon}{2 + V\epsilon} * \frac{\epsilon}{2 + V\epsilon} \end{align}$

Smoothing with collection frequency

Let cf_t be the collection frequency of term t

- That is, the total number of times it occurs (as opposed to df_t).

Then if term t does not appear in document d.

- We want $P(t|M_d) < \frac{cf_t}{T}$
- $T=$ total number of tokens in all documents.

Let M_c be the language model for the entire document collection:

$\begin{align} P(t|M_c) = \frac{cf_{t}}{T} \end{align}$

Dirichlet Smoothing

$\begin{align} P_{dir}(t|M_d) = \frac{tf_{t, d} + \alpha P(t|M_c)}{L_d + \alpha} \end{align}$

- $\alpha:$ tunable parameter
- Larger $\alpha \rightarrow$ more smoothing.

Interpolation Smoothing

Alternatively, we can *interpolate* between the document probability and the collection probability:

$\begin{align}P_{interp}(t|M_d) & = & \lambda P(t|M_d) + (1-\lambda) P(t|M_c)\\ & = & \lambda \frac{tf_{t, d}}{L_d} + (1-\lambda) \frac{cf_{t}}{T} \end{align}$

- λ is a tunable parameter.
- Smaller $\lambda \rightarrow$ more smoothing.
- This is also called *Jelinek-Mercer* smoothing.

Thus, the new query likelihood is:

$\begin{align} P_{interp}(q|M_d) = \prod_{t \in q} P_{interp}(t|M_d) = \prod_{t \in q} \lambda \frac{tf_{t, d}}{L_d} + (1-\lambda) \frac{cf_{t}}{T} \end{align}$

Interpolation Example

(from MRS p. 246)

- $d_1:$ Xyzzy reports a profit but revenue is down
- $d_2:$ Quorus narrows quarter loss but revenue decreases further
- $\lambda=.5$

Suppose the query is **revenue down**. Then:

$P_{interp}(q|d_1) = $

$P_{interp}(q|d_2) = $

$\begin{align}P_{interp}(t|M_d) & = & \lambda P(t|M_d) + (1-\lambda) P(t|M_c)\\ & = & \lambda \frac{tf_{t, d}}{L_d} + (1-\lambda) \frac{cf_{t}}{T} \end{align}$

Interpolation Example

(from MRS p. 246)

- $d_1:$ Xyzzy reports a profit but revenue is down
- $d_2:$ Quorus narrows quarter loss but revenue decreases further
- $\lambda=.5$

Suppose the query is revenue down. Then:

$P_{interp}(q|d_1) = [(1/8 + 2/16)/2] * [(1/8 + 1/16)/2] = 1/8 * 3/32 = 3/256$

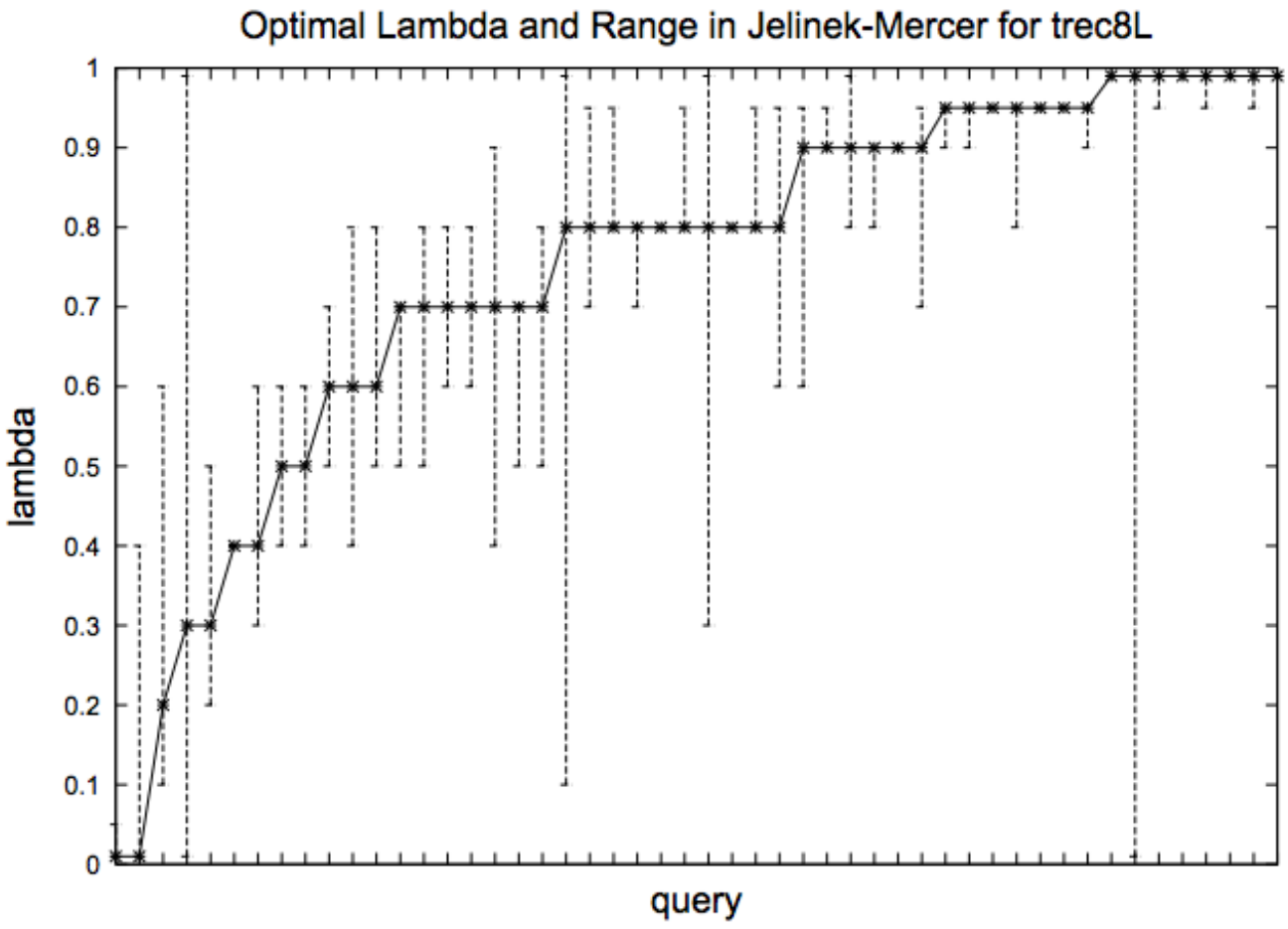
$P_{interp}(q|d_2) = [(1/8 + 2/16)/2] * [(0/8 + 1/16)/2] = 1/8 * 1/32 = 1/256$

Where are the following quantities used, if at all?

- Term frequency in a document
- Collection frequency of a term
- Document frequency of a term
- Length normalization of a term

$\begin{align} P_{interp}(q|M_d) = \prod_{t \in q} P_{interp}(t|M_d) = \prod_{t \in q} \lambda \frac{tf_{t, d}}{L_d} + (1-\lambda) \frac{cf_{t}}{T} \end{align}$

Should amount of smoothing (λ) depend on query length?



Source: Zhai & Lafferty, 2004

Language Model vs. tf-idf

	Precision			
Rec.	tf-idf	LM	%chg	
0.0	0.7439	0.7590	+2.0	
0.1	0.4521	0.4910	+8.6	
0.2	0.3514	0.4045	+15.1	*
0.3	0.2761	0.3342	+21.0	*
0.4	0.2093	0.2572	+22.9	*
0.5	0.1558	0.2061	+32.3	*
0.6	0.1024	0.1405	+37.1	*
0.7	0.0451	0.0760	+68.7	*
0.8	0.0160	0.0432	+169.6	*
0.9	0.0033	0.0063	+89.3	
1.0	0.0028	0.0050	+76.9	
Ave	0.1868	0.2233	+19.55	*

Source: MRS

$\begin{align} P_{interp}(q|M_d) = \prod_{t \in q} P_{interp}(t|M_d) = \prod_{t \in q} \lambda \frac{tf_{t, d}}{L_d} + (1-\lambda) \frac{cf_{t}}{T} \end{align}$

vs.

$ S_{tfidf}(q, d) = \sum_{t \in q} \begin{align} (1 + \log(tf_{t, d})) * \log(\frac{N}{df_t}) \end{align}$

docID	Document text
1	click go the shears boys click click click
2	click click
3	metal here
4	metal shears click here

Query	Doc 1	Doc 2	Doc 3	Doc 4
click				
shears				
click shears				

Let $\lambda=0.5$.

$\begin{align} P_{interp}(q|M_d) = \prod_{t \in q} P_{interp}(t|M_d) = \prod_{t \in q} \lambda \frac{tf_{t, d}}{L_d} + (1-\lambda) \frac{cf_{t}}{T} \end{align}$

(Source: MRS)

CS 429: Information Retrieval

Lecture 22: Expectation Maximization for Clustering

Dr. Aron Culotta

Illinois Institute of Technology

Spring 2014

Below, we use expectation maximization to find the means of two clusters for two-dimensional data.

We assume diagonal covariance matrices.

The bivariate normal density in this case is:

$$\begin{align} N(\vec{\mu}, \vec{\sigma}, \vec{x}) = \frac{1}{2\pi\sigma_1\sigma_2} \exp\left(- \frac{(x_1-\mu_1)^2}{2\sigma_1} + \frac{(x_2-\mu_2)^2}{2\sigma_2}\right) \end{align}$$

Let $\vec{\mu^j}, \vec{\sigma^j}$ be the mean and variance for cluster j. We update at each iteration with:

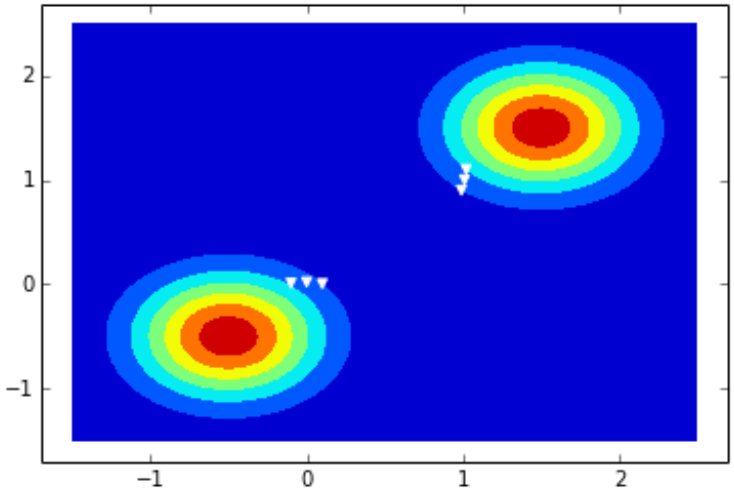
$$\begin{align} \mu_j' = \frac{\sum_i N(\vec{\mu^j}, \vec{\sigma^j}, \vec{x_i})\vec{x_i}}{\sum_i N(\vec{\mu^j}, \vec{\sigma^j}, \vec{x_i})} \end{align}$$

In [274]:

```
% pylab inline
import numpy as np
from numpy import array as npa

def gauss(mean, covar, x):
    """
    Bivariate Gaussian distribution, assuming diagonal covariance.
    1/(2*pi*v1*v2) * exp(- 1/2 * (x1-mean1)**2/v1 + (x2-mean2)**2/v2)
    """
    Xmu = x[0]-mean[0]
    Ymu = x[1]-mean[1]
    z = Xmu**2 / (covar[0]**2) + Ymu**2 / (covar[1]**2)
    denom = 2 * np.pi * covar[0] * covar[1]
    return np.exp(-z / 2.) / denom

data = np.array([npa([0.,0.02]), npa([.1,0.005]), npa([-0.1,.01]),
                 npa([1.01,1.]), npa([.99,0.9]), npa([1.02,1.1])])

mean1 = npa([-.5, -.5])
covar1 = npa([.4, .4])
mean2 = npa([1.5, 1.5])
covar2 = npa([.4, .4])

def plotme(mean1, covar1, mean2, covar2, data):
```

```
    x, y = np.mgrid[-1.5:2.5:.01, -1.5:2.5:.01]
    pos = np.empty(x.shape + (2,))
    pos[:, :, 0] = x; pos[:, :, 1] = y
    contourf(x, y,
             [gauss(mean1, covar1, [xi, yi]) + gauss(mean2, covar2, [xi, yi])
              for (xi, yi) in zip(x, y)])
    scatter([d[0] for d in data], [d[1] for d in data], marker='v', color='w')

plotme(mean1, covar1, mean2, covar2, data)
```

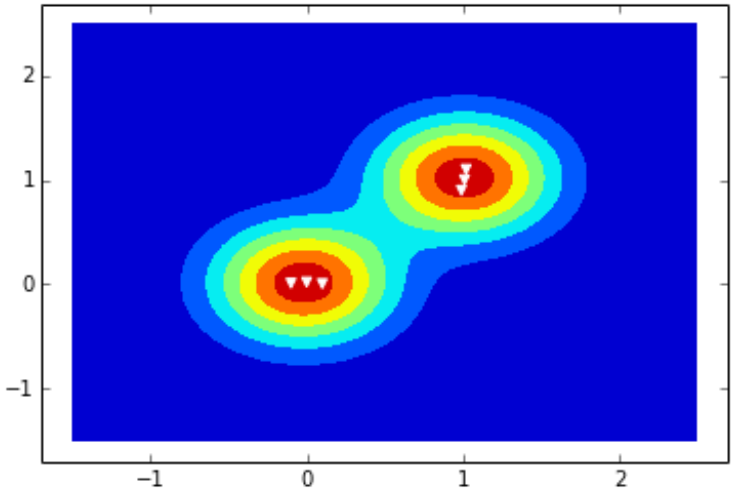
```
Populating the interactive namespace from numpy and matplotlib
```



In [275]:

```
def e_step(data, mean1, covar1, mean2, covar2):
    results = []
    for point in data:
        results.append([gauss(mean1, covar1, point),
                        gauss(mean2, covar2, point)])
    return results

def m_step(data, mean1, covar1, mean2, covar2, probs):
    new_mean1 = npa([0., 0.])
    new_mean2 = npa([0., 0.])
    for point, prob in zip(data, probs):
        new_mean1 += prob[0] * point
        new_mean2 += prob[1] * point
    new_mean1 /= sum(p[0] for p in probs)
    new_mean2 /= sum(p[1] for p in probs)
    return new_mean1, new_mean2

def iterate(data, mean1, covar1, mean2, covar2):
    probs = e_step(data, mean1, covar1, mean2, covar2)
    new_mean1, new_mean2 = m_step(data, mean1, covar1, mean2, covar2, probs)
    print 'new mean1=', new_mean1, '\nnew mean2=', new_mean2
    print 'difference in means=', np.sum(np.abs(new_mean1 - mean1) + np.abs(new_
mean2 - mean2))
    plotme(new_mean1, covar1, new_mean2, covar2, data)
    return new_mean1, new_mean2

mean1, mean2 = iterate(data, mean1, covar1, mean2, covar2)
```

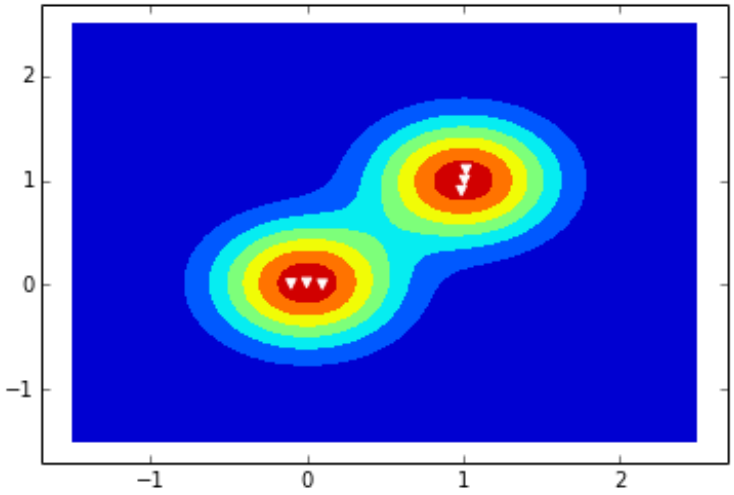
```
new mean1= [-0.02004906  0.01202256]
new mean2= [ 1.01010493  1.02306492]
difference in means= 1.9588036624
```



In [276]:

```
mean1, mean2 = iterate(data, mean1, covar1, mean2, covar2)
```

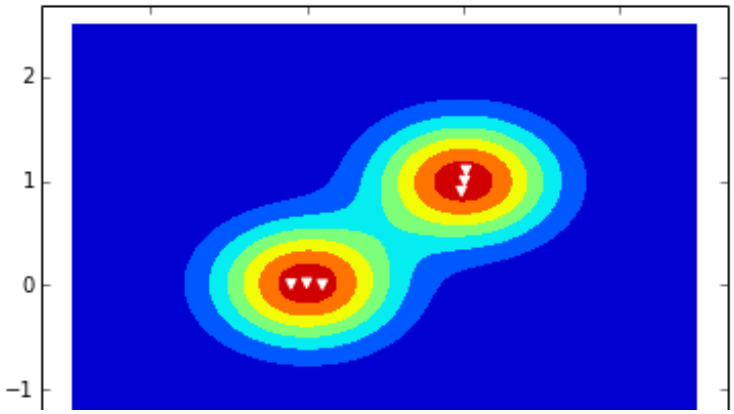
```
new mean1= [ 0.00123465  0.01372373]
new mean2= [ 1.0050071   0.99909989]
difference in means= 0.0520477245206
```



In [277]:

```
mean1, mean2 = iterate(data, mean1, covar1, mean2, covar2)
```

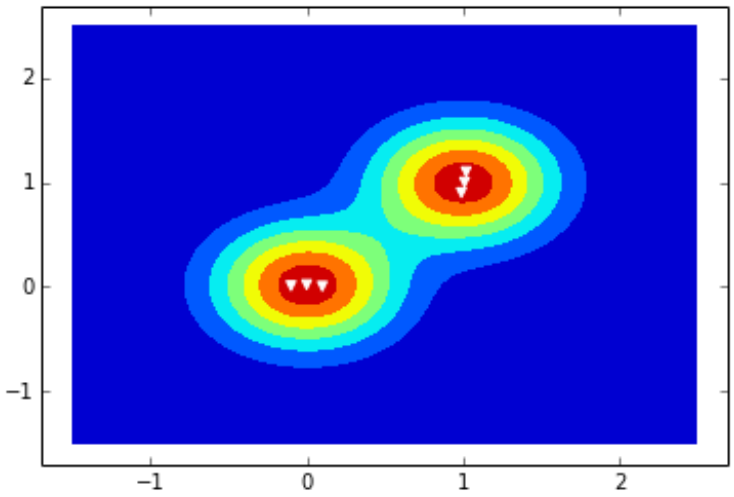
```
new mean1= [ 0.00242549  0.01400209]
new mean2= [ 1.00449348  0.99771439]
difference in means= 0.00336831060153
```



In [278]:

```
mean1, mean2 = iterate(data, mean1, covar1, mean2, covar2)
```

```
new mean1= [ 0.00249586  0.01402144]
new mean2= [ 1.00445858  0.99762789]
difference in means= 0.000211134878354
```

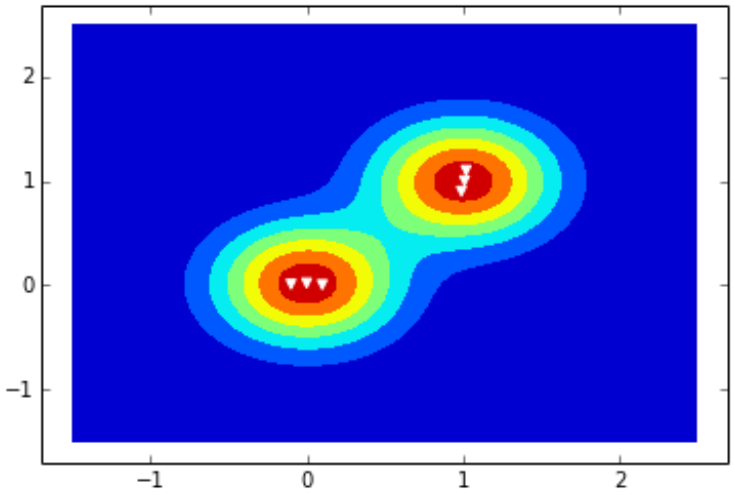


In [279]:

```
for i in range(10):
    mean1, mean2 = iterate(data, mean1, covar1, mean2, covar2)
```

```
new mean1= [ 0.00250007  0.01402263]
new mean2= [ 1.00445635  0.99762242]
difference in means= 1.30901486128e-05
new mean1= [ 0.00250032  0.0140227 ]
new mean2= [ 1.00445621  0.99762208]
difference in means= 8.10454092567e-07
new mean1= [ 0.00250033  0.01402271]
new mean2= [ 1.0044562   0.99762205]
difference in means= 5.0198829414e-08
new mean1= [ 0.00250033  0.01402271]
new mean2= [ 1.0044562   0.99762205]
difference in means= 3.11143275554e-09
new mean1= [ 0.00250033  0.01402271]
new mean2= [ 1.0044562   0.99762205]
difference in means= 1.92993729938e-10
new mean1= [ 0.00250033  0.01402271]
new mean2= [ 1.0044562   0.99762205]
difference in means= 1.19795852925e-11
new mean1= [ 0.00250033  0.01402271]
new mean2= [ 1.0044562   0.99762205]
difference in means= 7.43958714078e-13
new mean1= [ 0.00250033  0.01402271]
new mean2= [ 1.0044562   0.99762205]
difference in means= 4.62785192112e-14
new mean1= [ 0.00250033  0.01402271]
new mean2= [ 1.0044562   0.99762205]
difference in means= 2.99760216649e-15
new mean1= [ 0.00250033  0.01402271]
```

```
new mean2= [ 1.0044562   0.99762205]
difference in means= 2.71917904859e-16
```



In [ ]:

CS 429: Information Retrieval

Lecture 23: Clustering Words

Dr. Aron Culotta

Illinois Institute of Technology

Spring 2014

Motivation

Often, we want to know which features appear together.

- If you liked *Twilight* you might like *Nosferatu*.
- "happy" is a synonym of "glad."

We'll use k-means to cluster together related words from Twitter.

Caution: This uses live Twitter data, which often contains profanity.

In [21]:

```
# Get some tweets containing the word 'i'.

import os
from TwitterAPI import TwitterAPI

# Read Twitter credentials from environmental variables.
api = TwitterAPI(os.environ.get('TW_CONSUMER_KEY'),
                 os.environ.get('TW_CONSUMER_SECRET'),
                 os.environ.get('TW_ACCESS_TOKEN'),
                 os.environ.get('TW_ACCESS_TOKEN_SECRET'))

# Collect tweets until we hit rate limit.
tweets = []
while True:
    r = api.request('search/tweets', {'q':'i',
                                      'language':'en',
                                      'count':'100'})
    if r.status_code != 200: # error
        break
    else:
        for item in r.get_iterator():
            tweets.append(item)
```

In [416]:

```
print len(tweets)
```

```
17300
```

In [417]:

```
# Each tweet is a Python dict.
print 'text', tweets[10]['text']
print 'description:', tweets[10]['user']['description']
print 'name:', tweets[10]['user']['name']
print 'location:', tweets[10]['user']['location']
```

```
text "You are amazing." - Silas says to Sam. I have to agree. #GH
description: I am a former editor.  A soon to be grad student and educator.  I currently have a book about my life in editing mode.
name: Sean Greeley
location: Oakland Gardens, New York
```

In [418]:

```
# Tokenize each tweet text.
import re
tokens = []
for tweet in tweets:
    text = tweet['text'].lower()
    text = re.sub('@\S+', ' ', text)  # Remove mentions.
    text = re.sub('http\S+', ' ', text)  # Remove urls.
    tokens.append(re.findall('[A-Za-z]+', text)) # Retain words.
print tokens[1]
```

```
[u'i', u'm', u'strangely', u'ok', u'with', u'being', u'known', u'as', u'a', u'day', u'drinker']
```

In [419]:

```
# Count words.
from collections import Counter

word_counts = Counter()
for tweet in tokens:
    word_counts.update(tweet)
```

In [420]:

```
# Inspect word counts.
import math

print len(word_counts), 'unique terms'
sorted_counts = sorted(word_counts.items(),
                       key=lambda x: x[1],
                       reverse=True)
print '\n'.join('%s\t%d' % (w, c) for w, c in sorted_counts[:10])
```

```
18713 unique terms
i 21239
rt 5812
to 5137
you 4381
the 4192
a 4121
```

```
m 3303
my 3070
and 2891
me 2768
```

In [421]:

```
# Retain in vocabulary words occurring more than twice.
vocab = set([w for w, c in word_counts.iteritems() if c > 2])
print '%d words occur at least three times.' % len(vocab)
```

```
4915 words occur at least three times.
```

In [422]:

```
# Prune tokens.
newtoks = []
for i, tweet in enumerate(tokens):
    newtok = [token for token in tweet if token in vocab]
    if len(newtok) > 0:
        newtoks.append(newtok)
tokens = newtoks
```

In [423]:

```
# A sample pruned tweet.
print tokens[1]
```

```
[u'i', u'm', u'ok', u'with', u'being', u'known', u'as', u'a', u'day']
```

In [426]:

```
# For each term, create a context vector, indicating how often
# each word occurs to the left or right of it.
from collections import defaultdict

# dict from term to context vector.
contexts = defaultdict(lambda: Counter())
window = 2
for tweet in tokens:
    for i, token in enumerate(tweet):
        features = []
        for j in range(np.amax([0, i-window]), i):
            features.append(tweet[j] + "@" + str(j-i))
        for j in range(i+1, min(i + window, len(tweet))):
            features.append(tweet[j] + "@" + str(j-i))
        contexts[token].update(features)
        # Optionally: ignore word order
        # contexts[token].update(tweet[:i] + tweet[i+1:])
```

In [428]:

```
print contexts['you'].items()[:5]
```

```
[(u'talk@-2', 13), (u'touch@-2', 1), (u'please@1', 32), (u'man@-1', 1), (u'date
@-1', 2)]
```

In [429]:

```
# Compute the number of different contexts each term appears in.
tweet_freq = Counter()
for context in contexts.itervalues():
    for term in context:
        tweet_freq[term] += 1.
print tweet_freq.items()[:5]
```

```
[(u'please@1', 90.0), (u'history@-1', 7.0), (u'sum@1', 11.0), (u'date@-2', 22.0
), (u'history@-2', 8.0)]
```

In [430]:

```
# Transform each context vector to be term freq / tweet frequency.
# Also then normalize by length.
for term, context in contexts.iteritems():
    for term2, frequency in context.iteritems():
        context[term2] = frequency / (1. + math.log(tweet_freq[term2]))
    length = math.sqrt(sum([v*v for v in context.itervalues()]))
    for term2, frequency in context.iteritems():
        context[term2] = 1. * frequency / length

print contexts['being'].items()[:5]
```

```
[(u'comes@-2', 0.021457572258183157), (u'or@1', 0.013519637819578057), (u'as@-1
', 0.030798751481017773), (u'time@-1', 0.030514456587444547), (u'wow@1', 0.0210
1070082357469)]
```

At this point we have ~5k dictionaries, one per term, indicating the terms that co-occur (weighted by inverse tweet frequency).

Next, we have to cluster these vectors. To do this, we'll need to be able to compute the euclidean distance between two vectors.

In [431]:

```
def distance(c1, c2):
    if len(c1.keys()) == 0 or len(c2.keys()) == 0:
        return 1e9
    keys = set(c1.keys()) | set(c2.keys())
    distance = 0.
    for k in keys:
        distance += (c1[k] - c2[k]) ** 2
    return math.sqrt(distance)

print distance({'hi':10, 'bye': 5}, {'hi': 9, 'bye': 4})
print distance({'hi':10, 'bye': 5}, {'hi': 8, 'bye': 4})
```

```
1.41421356237
2.2360679775
```

In [432]:

```
def find_closest(term, n=5):
    terms = np.array(contexts.keys())
    context = contexts[term]
    distances = []
    for term2, context2 in contexts.iteritems():
```

```
        distances.append(distance(context, context2))
    return terms[np.argsort(distances)][:n]

print find_closest('sad', n=10)
```

```
[u'sad' u'sleepy' u'excited' u'glad' u'sorry' u'scared' u'lazy' u'bad'
 u'hungry' u'nervous']
```

In [436]:

```
nz_contexts = [t for t, context in contexts.items()
               if len(context) > 1]
contexts = dict([(term, contexts[term]) for term in nz_contexts])
print len(nz_contexts), 'nonzero contexts'
```

```
4910 nonzero contexts
```

In [437]:

```
# Transform context dicts to a sparse vector
# for sklearn.
from sklearn.feature_extraction import DictVectorizer

vec = DictVectorizer()
X = vec.fit_transform(contexts.values())
names = np.array(vec.get_feature_names())
print names[:10]
print X[0]
```

```
[u'a@-1' u'a@-2' u'a@1' u'aap@-1' u'aap@-2' u'aap@1' u'aapl@-1' u'aapl@-2'
 u'aapl@1' u'aaron@-1']
  (0, 4941) 0.237473316099
  (0, 7742) 0.374375711285
  (0, 6262) 0.146969238152
  (0, 2252) 0.374375711285
  (0, 13400) 0.166342996686
  (0, 10740) 0.143485404568
  (0, 8758) 0.163703930656
  (0, 7285) 0.339401454089
  (0, 2191) 0.229437338908
  (0, 8608) 0.61729242976
  (0, 1) 0.133230276709
```

In [438]:

```
# Let's cluster!
from sklearn.cluster import KMeans
num_clusters = 50
kmeans = KMeans(num_clusters)
kmeans.fit(X)
```

Out[438]:

```
KMeans(copy_x=True, init='k-means++', max_iter=300, n_clusters=50, n_init=10,
    n_jobs=1, precompute_distances=True, random_state=None, tol=0.0001,
    verbose=0)
```

In [439]:

```
# Let's print out the top features for each mean vector.
```

```
# This is swamped by common terms
for i in range(num_clusters):
    print i, ' '.join(names[np.argsort(kmeans.cluster_centers_[i])[::-1][:5]])
```

```
0 i@-2 i@-1 just@-1 ve@-1 a@1
1 the@-1 i@1 in@-2 to@-2 and@1
2 like@1 it@-1 elephant@-2 to@-1 starting@-2
3 be@-1 to@-2 i@-2 t@-2 being@-1
4 a@-1 i@1 my@-1 the@-1 a@-2
5 you@1 i@-2 i@-1 i@1 me@1
6 t@-1 can@-2 don@-2 i@-1 to@-1
7 i@-2 m@-1 m@-2 i@1 be@-1
8 my@-2 my@-1 i@1 the@-2 a@-1
9 an@-1 i@1 with@1 a@-2 for@-2
10 she@-1 it@-1 her@-2 he@-1 door@-2
11 your@-1 my@-1 in@-2 the@-1 i@1
12 up@1 i@-2 i@-1 a@-1 to@-1
13 the@-1 the@-2 in@-2 i@1 of@-2
14 and@-1 is@-1 he@-1 for@-1 the@1
15 a@-2 a@-1 i@1 but@1 and@1
16 i@-2 love@-1 m@-1 i@1 i@-1
17 proud@-2 to@-1 pleased@-2 i@1 rider@1
18 with@1 i@-2 to@1 m@-2 m@-1
19 in@-1 i@1 to@-1 for@-1 and@1
20 me@1 i@-1 you@1 you@-1 to@-1
21 to@-1 the@1 i@1 i@-1 to@1
22 lil@-1 gucci@-1 nia@-2 lady@1 essential@-1
23 i@-1 i@-2 rt@-2 i@1 a@1
24 i@-1 i@-2 a@1 rt@-2 i@1
25 of@1 the@-1 a@-1 i@-2 the@-2
26 san@-1 santa@-1 at@-2 churrascaria@-1 his@-1
27 de@1 de@-1 de@-2 m@-2 at@-1
28 to@-2 i@1 see@-1 the@-1 and@1
29 i@1 rt@-1 i@-2 the@-2 you@1
30 and@1 i@-2 the@-1 the@-2 my@-1
31 my@-1 i@1 in@-2 and@1 on@-2
32 i@-2 to@-2 i@1 the@1 for@-1
33 of@-1 the@-1 i@1 out@-2 i@-2
34 fa@1 oy@-2 santiago@-2 cia@-1 tylko@-1
35 my@1 the@1 i@-2 i@-1 a@1
36 so@-1 m@-2 i@-2 i@1 m@-1
37 gain@-2 gain@-1 rts@-2 followtrick@-1 teamfollowback@-1
38 from@-1 video@-2 national@-1 day@1 i@1
39 on@-1 i@1 on@-2 the@-1 the@-2
40 in@1 i@-2 the@-2 get@-1 to@-2
41 i@1 rt@-1 the@-1 i@-2 a@-2
42 to@1 i@-2 i@-1 m@-1 not@-1
43 rt@-1 rt@-2 i@-1 the@-2 i@-2
44 a@-1 i@1 and@1 the@-1 of@1
45 to@-1 i@-2 i@-1 it@1 my@1
46 you@-1 i@1 love@-2 i@-2 you@-2
47 t@1 i@-1 you@-1 if@-2 it@-1
48 flaherty@1 partnered@-2 rip@-1 pag@-1 affected@-2
49 on@1 i@-2 the@-1 a@-1 a@-2
```

In [440]:

```
# .transform will compute the distance from each term to each cluster.
distances = kmeans.transform(X)
```

```
print distances[0]
```

```
[ 1.13521267  1.1240182   1.10742592  1.08732169  1.03694299  1.04006013
  1.15302655  1.15812103  1.03107212  1.08102953  1.10724202  1.05408489
  1.11956906  1.02070019  1.0508641   1.01893944  1.0311229   1.1370101
  1.05999362  1.010669    1.0954494   1.07168689  1.02534256  1.22808261
  1.05724649  1.10921475  1.08896775  1.0245512   1.03153311  1.12516247
  1.02367389  1.10663023  1.09598667  1.05891253  1.04467328  1.07224092
  1.12788733  1.09460034  1.0410211   1.06544318  1.06445839  1.0165742
  1.14504997  0.99765603  1.12288503  1.13720466  1.04061974  1.37525835
  1.07813631  1.06099406]
```

In [441]:

```
# Finally, we'll print the words that are closest
# to the mean of each cluster.
terms = np.array(contexts.keys())
for i in range(num_clusters):
    print i, ' '.join(terms[np.argsort(distances[:,i])[1:10]])
```

```
0 saw got entered checked realized learned heard had made
1 world truth worst gym bathroom devil beach masters most
2 looks mctc feel titties feels sleeps essays look fly
3 bothered loyal honest expressed surprised attending appreciated able trusted
4 few little joke couple guy job good stoner boyfriend
5 cursing ship meet if invite miss annoying stalk tell
6 wait even stand breathe contain stop handle blame care
7 hungry guessing melting confused gonna lien getting afraid buried
8 yard dream glasses nationalsiblingday teacher liam twin watches brother
9 hour option adult indirect egg assignment argument upgrade acc
10 stays raps bent moaned screams tasted s takes hurts
11 face opinion butt stomach life mouth head heart car
12 wake fucked hurry messed ended give woken waking catching
13 same beach bathroom worst truth most masters way gym
14 hayley jazmyn adjust shake follows adj chase then signing
15 mood favor person photo kid pace tool tan buddy
16 you getting u probably afraid hungry done some a
17 satisfied lasted host boli say work share do play
18 partnered content quote done deeply pleased deal agree concerned
19 class japan town front depth politics ohio bed cleveland
20 excuse tell remind gave make reminds ignore inspired followed
21 play eat draw clarify go brush meet sleep spend
22 sis punk rob freaky mane doe steve wayne czym
23 hate got swear am hope volunteer ll think thought
24 got hate am hope swear have volunteer thought just
25 amount type sort instead none part rest lots pair
26 fransico monica cruz pride rn sons spf ce maria
27 deus de banda el centro las gol e hospital
28 ourselves playlist bathroom dms moon portland pausing him them
29 but ahhh cause yes yep yeah lol and because
30 paradise fun prison pencil legend gps brothers yogurt favour
31 tl phone wrist hair heart dad car soul own
32 u a him not that some getting them afraid
33 justice habit india couples college weirdos those duty nowhere
34 madre vuruyor nun qu oy sms ada chcecie metropolitana
35 stole updated wasting in rearranged change packing washed into
36 much sleepy cute hard glad lazy bad far upset
37 followtrick onedirection ua tcfollowtrain mm teamfollowback ipadgames retwee
t vod
38 dallas minecraft argentina sos congress throwbackthursday battle mw monsters
```

```
39 soundcloud yelp youtube sunday speak tumblr saturday holiday instagram
40 faith performing stuck checked participate interested masturbate landed invo
lved
41 when ahhh cause yes and lol yep yeah because
42 trying listen unable listening used addicted going talk able
43 do one all lol like shit man and for
44 few couple little joke stoner cookie cassette nap chance
45 find watch take be do say make put play
46 were guys shawty are spinnrtaylorswift inspired
happybdayshaymitchellfrombrazil bby join
47 ain wouldn didn haven couldn wasn won shouldn hadn
48 mr cdnpoli parallels makism autism gising tayo society mplaces
49 focus focusing hooked depends working lock focused poop banged
```

Clearly, interpreting these results requires a bit of investigation.

Some patterns do emerge:

- Cluster 47 is all contractions
- Cluster 29,41 seems to be inerjections/disfluencies
- Cluster 39 refers to tech sites.
- Cluster 34 has Polish and what looks like Spanish.

As the number of tweets increases, we expect these clusters to become more coherent.

CS 429: Information Retrieval

Lecture 26: HITS

Dr. Aron Culotta

Illinois Institute of Technology

Spring 2014

The **Hubs and Authorities** algorithm is a simple procedure to assign each page two scores:

- **hub score:** how likely is this page to be a directory?
- **authority score:** how likely is this page to be a trustworthy resource on a topic?

Let F_u be *forward* links (going out from u).

Let B_u be *back* links (going in to u).

$ \begin{align} h(u) = \sum_{v \in F_u} a(v)\\ a(u) = \sum_{v \in B_u} h(v)\\ \end{align}$

In words:

- The hub score for u is the sum of the authority scores for all pages linked from u.
- The authority score for u is the sum of the hub scores for all pages linking to u.

As for PageRank, we can compute these iteratively until convergence:

1. Initialize all hub/authority scores to 1.0
2. Loop until convergence
   A. update authority scores
   B. update hub scores

Let's try this out on some real data:

In [160]:

```
# Get some search results.
from google import search
urls = set([u for u in search('football teams', stop=20)])
print 'top', len(urls), 'results:\n', '\n'.join(urls)
```

```
top 32 results:
http://www.nfl.com/teams/seattleseahawks/profile?team=SEA
http://www.azcardinals.com/
http://www.usatoday.com/story/sports/nfl/2014/04/20/offseason-program-first-pha
se-begins/7946325/
http://www.goducks.com/SportSelect.dbml?SPID=233
http://www.nfl.com/teams/baltimoreravens/profile?team=BAL
```

http://www.cbssports.com/collegefootball/teams
http://www.friendstagger.com/tag-your-friends-as-nfl-football-teams-2-facebook-myspace/
http://www.nfl.com/teams/denverbroncos/profile?team=DEN
http://sports.yahoo.com/ncaa/football/teams
http://highschoolsports.nj.com/football/teams/
http://espn.go.com/nfl/teams
http://www.king5.com/sports/football-Offseason-activities-OTA-255999771.html
http://www.willamette.edu/athletics/football/
http://www.friendstagger.com/tag-your-friends-as-nfl-football-teams-facebook-myspace/
http://www.cbssports.com/nfl/teams
http://www.freevector.com/canadian-football-teams/
http://www.forbes.com/nfl-valuations/
http://www.nfl.com/teams/newenglandpatriots/profile?team=NE
http://www.sodahead.com/entertainment/are-you-a-red-skins-fan-or-other/question-3688699/
http://www.sporcle.com/games/printzj/ncaahelmet1
http://www.sbnation.com/college-football/2014/3/31/5546726/professional-wrestlers-as-college-football-teams
http://en.wikipedia.org/wiki/National_Football_League
http://nfltix.net/nfl-football-prismatic-stickers-complete-set-of-32-teams-decals/
http://www.maxpreps.com/news/lRfk29Nphkue38lH0WvW0A/maxpreps-2013-all-american-football-teams.htm
http://www.nfl.com/teams
http://en.wikipedia.org/wiki/Football_team
http://www.puma.com/football/teams
http://msn.foxsports.com/nfl/teams
http://en.wikipedia.org/wiki/Category:American_football_teams
http://sportsillustrated.cnn.com/football/nfl/teams/
http://blogs.ajc.com/georgia-high-school-sports/2013/12/27/ajc-all-state-football-teams-2013/?cxntfid=blogs_georgia_high_school_sports
http://www.pro-football-reference.com/teams/

In [161]:

```
# Download links for each url. Store inlinks/outlinks for each page in original set.
from collections import defaultdict
import requests
from BeautifulSoup import BeautifulSoup
inlinks = defaultdict(lambda: set())   # url -> set of inlinks
outlinks = defaultdict(lambda: set())  # url -> set of outlinks
for url in urls:
    soup = BeautifulSoup(requests.get(url).text)
    # Exclude self links and restrict to links in original search results.
    links = set([a['href'] for a in soup.findAll('a') if a.get('href')
                 and a['href'] in urls and a['href'] != url])
    outlinks[url] = links
    for link in links:
        inlinks[link].add(url)
```

In [192]:

```
# Print outlinks.
for url in outlinks:
    if len(outlinks[url]) > 0:
        print '\n', url, '->\n', '\n'.join(outlinks[url])
```

```
http://www.nfl.com/teams/seattleseahawks/profile?team=SEA ->
http://www.azcardinals.com/

http://www.nfl.com/teams/baltimoreravens/profile?team=BAL ->
http://www.azcardinals.com/

http://www.cbssports.com/collegefootball/teams ->
http://www.cbssports.com/nfl/teams

http://www.friendstagger.com/tag-your-friends-as-nfl-football-teams-2-facebook-
myspace/ ->
http://www.friendstagger.com/tag-your-friends-as-nfl-football-teams-facebook-my
space/

http://www.nfl.com/teams/denverbroncos/profile?team=DEN ->
http://www.azcardinals.com/

http://www.friendstagger.com/tag-your-friends-as-nfl-football-teams-facebook-my
space/ ->
http://www.friendstagger.com/tag-your-friends-as-nfl-football-teams-2-facebook-
myspace/

http://www.cbssports.com/nfl/teams ->
http://www.cbssports.com/collegefootball/teams

http://www.nfl.com/teams/newenglandpatriots/profile?team=NE ->
http://www.azcardinals.com/

http://www.nfl.com/teams ->
http://www.azcardinals.com/
```

In [193]:

```
# Print inlinks
for url in inlinks:
    if len(inlinks[url]) > 0:
        print '\n', url, '<-\n', '\n'.join(inlinks[url])
```

```
http://www.cbssports.com/collegefootball/teams <-
http://www.cbssports.com/nfl/teams

http://www.friendstagger.com/tag-your-friends-as-nfl-football-teams-2-facebook-
myspace/ <-
http://www.friendstagger.com/tag-your-friends-as-nfl-football-teams-facebook-my
space/

http://www.friendstagger.com/tag-your-friends-as-nfl-football-teams-facebook-my
space/ <-
http://www.friendstagger.com/tag-your-friends-as-nfl-football-teams-2-facebook-
myspace/

http://www.cbssports.com/nfl/teams <-
http://www.cbssports.com/collegefootball/teams

http://www.azcardinals.com/ <-
http://www.nfl.com/teams/seattleseahawks/profile?team=SEA
http://www.nfl.com/teams/baltimoreravens/profile?team=BAL
```

```
http://www.nfl.com/teams
http://www.nfl.com/teams/newenglandpatriots/profile?team=NE
http://www.nfl.com/teams/denverbroncos/profile?team=DEN
```

In [237]:

```
# Initialize hubs and authorities scores and print.
hubs = dict([(u, 1.0) for u in urls])
authorities = dict([(u, 1.0) for u in urls])
def print_top(hubs, authorities):
    print 'Top hubs\n', '\n'.join('%s %.6f' % (u[0], u[1]) for u in sorted(hubs.
items(), key=lambda x: x[1], reverse=True)[:5])
    print 'Top authorities\n', '\n'.join('%s %.6f' % (u[0], u[1]) for u in sorte
d(authorities.items(), key=lambda x: x[1], reverse=True)[:5])

print_top(hubs, authorities)
```

```
Top hubs
http://www.nfl.com/teams/seattleseahawks/profile?team=SEA 1.000000
http://www.azcardinals.com/ 1.000000
http://www.usatoday.com/story/sports/nfl/2014/04/20/offseason-program-first-pha
se-begins/7946325/ 1.000000
http://www.king5.com/sports/football-Offseason-activities-OTA-255999771.html 1.
000000
http://www.nfl.com/teams/baltimoreravens/profile?team=BAL 1.000000
Top authorities
http://www.nfl.com/teams/seattleseahawks/profile?team=SEA 1.000000
http://www.azcardinals.com/ 1.000000
http://www.usatoday.com/story/sports/nfl/2014/04/20/offseason-program-first-pha
se-begins/7946325/ 1.000000
http://www.king5.com/sports/football-Offseason-activities-OTA-255999771.html 1.
000000
http://www.nfl.com/teams/baltimoreravens/profile?team=BAL 1.000000
```

In [238]:

```
# Update hub and authority scores.
import math

def update(hubs, authorities, inlinks, outlinks):
    for u in authorities:
        authorities[u] += sum([hubs[inlink] for inlink in inlinks[u]])
    normalize(authorities)
    for u in hubs:
        hubs[u] += sum([authorities[outlink] for outlink in outlinks[u]])
    normalize(hubs)

def normalize(d):
    norm = math.sqrt(sum([v * v for v in d.values()]))
    for k in d:
        d[k] /= norm
```

In [239]:

```
update(hubs, authorities, inlinks, outlinks)
print_top(hubs, authorities)
```

```
Top hubs
http://www.nfl.com/teams/seattleseahawks/profile?team=SEA 0.255349
```

```
http://www.nfl.com/teams/baltimoreravens/profile?team=BAL 0.255349
http://www.nfl.com/teams/denverbroncos/profile?team=DEN 0.255349
http://www.nfl.com/teams/newenglandpatriots/profile?team=NE 0.255349
http://www.nfl.com/teams 0.255349
Top authorities
http://www.azcardinals.com/ 0.675053
http://www.cbssports.com/collegefootball/teams 0.225018
http://www.friendstagger.com/tag-your-friends-as-nfl-football-teams-2-facebook-
myspace/ 0.225018
http://www.friendstagger.com/tag-your-friends-as-nfl-football-teams-facebook-my
space/ 0.225018
http://www.cbssports.com/nfl/teams 0.225018
```

Expanding the set of urls

- For a given query, begin with the *root* set of the top k matching documents.
- Expand the set to include all forward and backward links from the root.

When would this help?