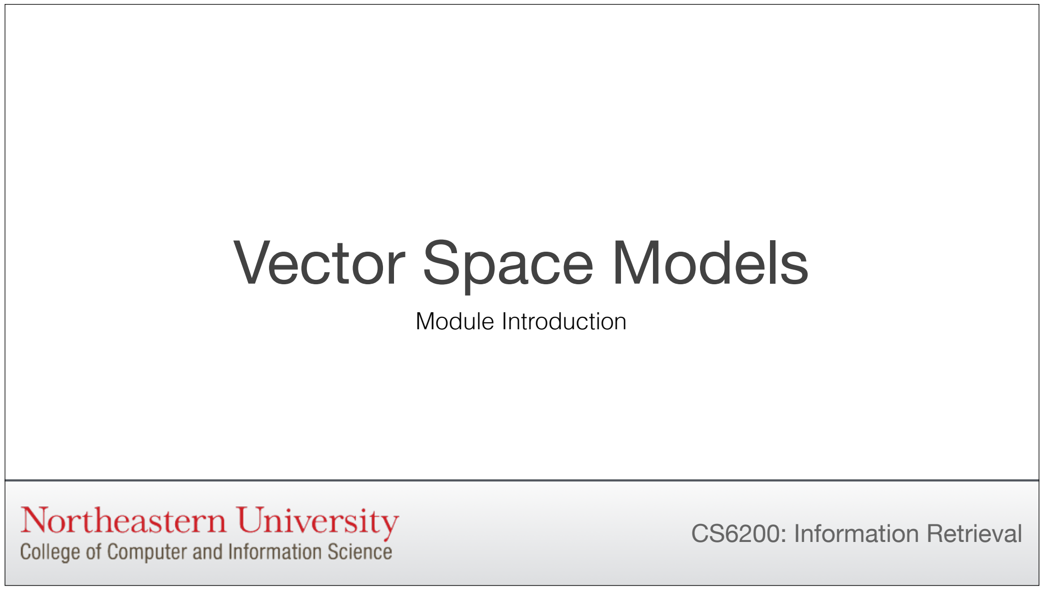


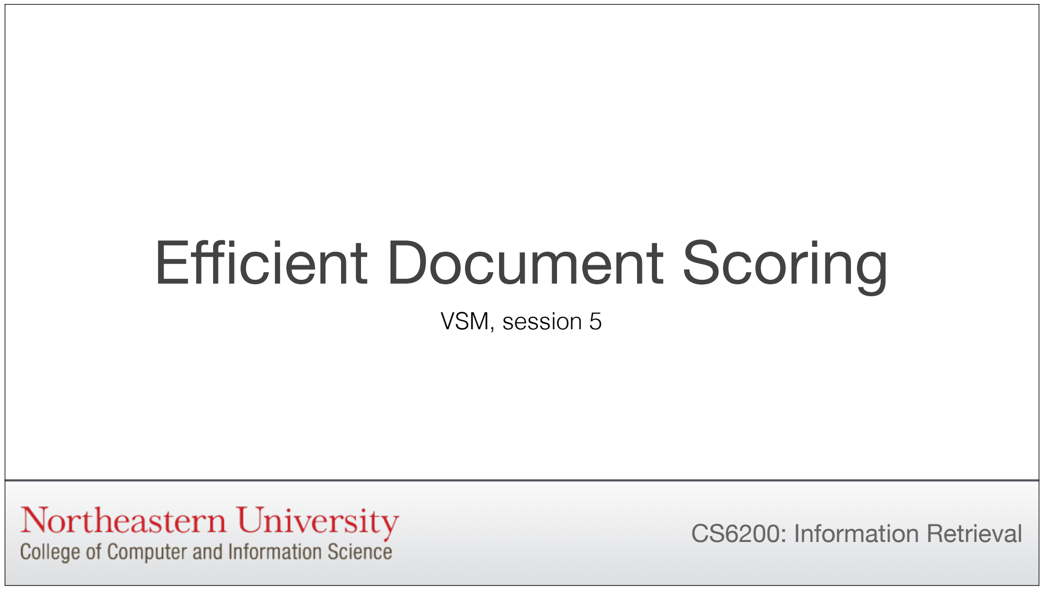
Vector Space Models

Module Introduction

Northeastern University
College of Computer and Information Science

CS6200: Information Retrieval

In the first module, we introduced Vector Space Models as an alternative to Boolean Retrieval. This module discusses VSMs in a lot more detail. By the end of the module, you should be ready to build a fairly capable search engine using VSMs. Let's start by taking a closer look at what's going on in a Vector Space Model.



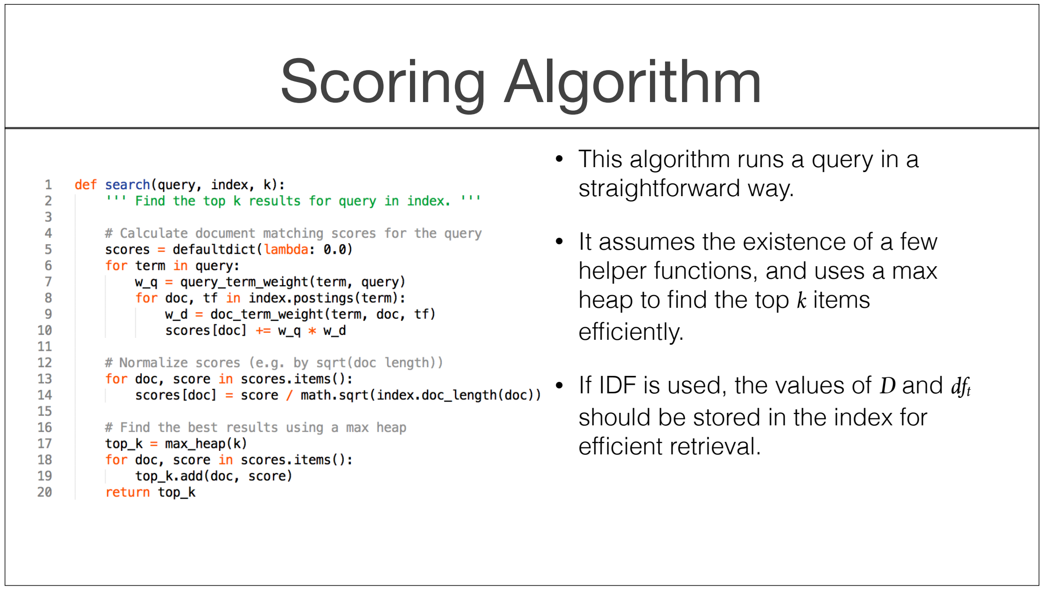
Efficient Document Scoring

VSM, session 5

Northeastern University
College of Computer and Information Science

CS6200: Information Retrieval

This session covers several strategies for speeding up the matching process at query time. Optimizing query run time is critical for a few reasons. It’s essential for providing a good customer experience – who wants to wait ten minutes to get their query results? It’s also important for the business to finish queries as rapidly as possible: faster queries consume fewer resources and satisfy more customers at less expense.

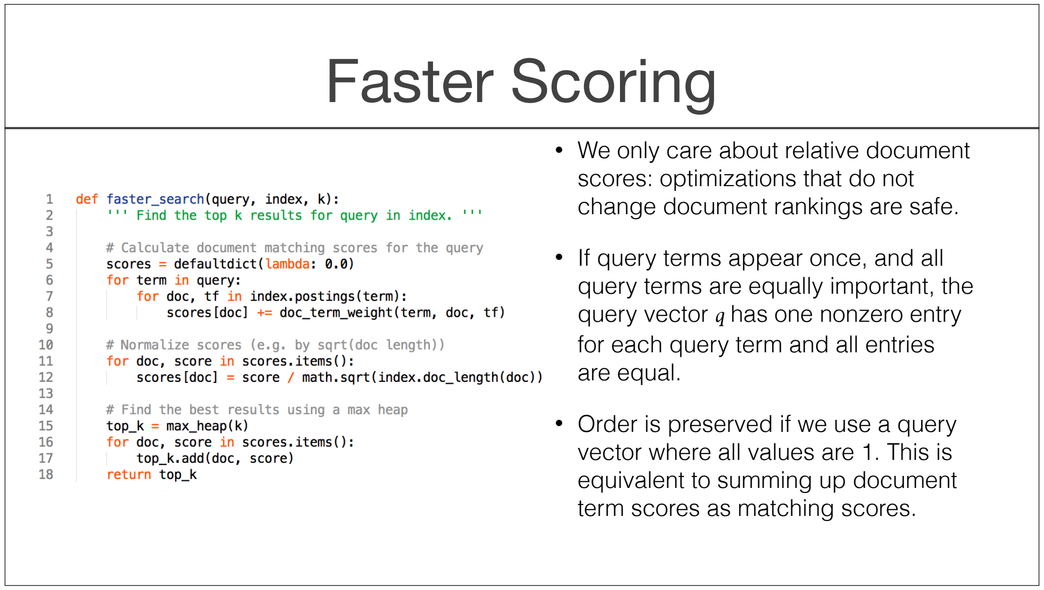


Scoring Algorithm

```python
def search(query, index, k):
    ''' Find the top k results for query in index. '''

    # Calculate document matching scores for the query
    scores = defaultdict(lambda: 0.0)
    for term in query:
        w_q = query_term_weight(term, query)
        for doc, tf in index.postings(term):
            w_d = doc_term_weight(term, doc, tf)
            scores[doc] += w_q * w_d

    # Normalize scores (e.g. by sqrt(doc length))
    for doc, score in scores.items():
        scores[doc] = score / math.sqrt(index.doc_length(doc))

    # Find the best results using a max heap
    top_k = max_heap(k)
    for doc, score in scores.items():
        top_k.add(doc, score)
    return top_k
```

- This algorithm runs a query in a straightforward way.
- It assumes the existence of a few helper functions, and uses a max heap to find the top k items efficiently.
- If IDF is used, the values of D and df_t should be stored in the index for efficient retrieval.

First, let's take a look at a generic query algorithm. We're passed a list of query terms, an index to run the query over, and the maximum number of documents to return.
For each query term, we first calculate the value in the query vector for that term. Then we iterate over the term's posting list. For each document, we calculate the value in the document vector for the term and then add the product of query and document values to the score we're accumulating for the document.
When we're finished processing terms and posting lists, we need to normalize our scores. We iterate over the scores, dividing by our normalization term. In this case, we use the sqrt of the document length. If we were using IDF, we would want to store the number of documents and the df for each term in the index so we could retrieve them in constant time.
Finally, we need to efficiently find the k highest-scoring documents. We construct a max heap of capacity k, and add each of our documents. The max heap is going to throw out all the documents with scores below the highest k scores it's seen so far. When we're done, we return the contents of the max heap as our sorted list.
This algorithm is straightforward and works well, but as it turns out, we can do better.



Faster Scoring

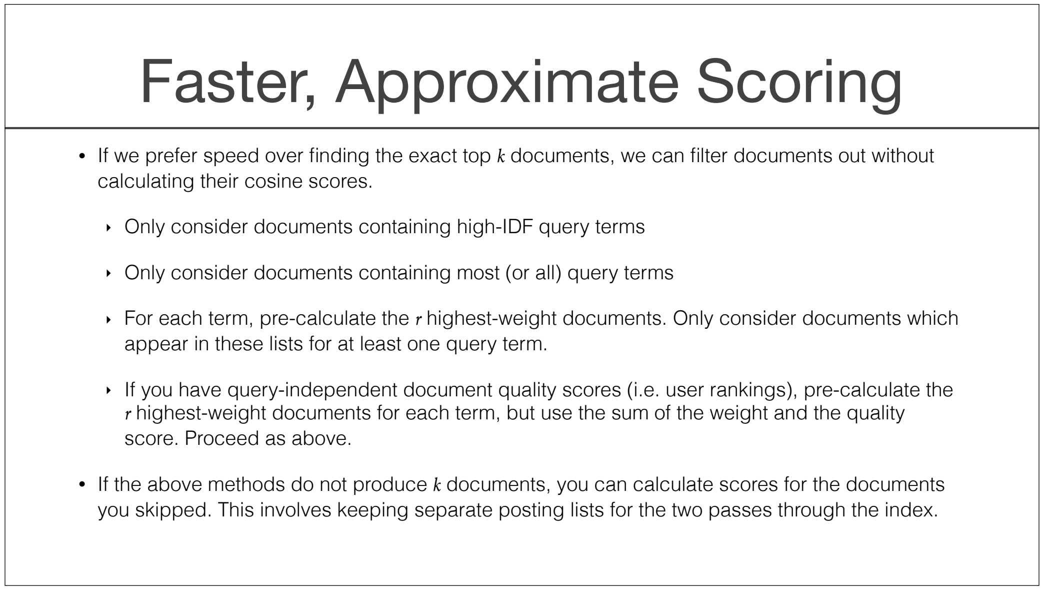
```python
def faster_search(query, index, k):
    ''' Find the top k results for query in index. '''

    # Calculate document matching scores for the query
    scores = defaultdict(lambda: 0.0)
    for term in query:
        for doc, tf in index.postings(term):
            scores[doc] += doc_term_weight(term, doc, tf)

    # Normalize scores (e.g. by sqrt(doc length))
    for doc, score in scores.items():
        scores[doc] = score / math.sqrt(index.doc_length(doc))

    # Find the best results using a max heap
    top_k = max_heap(k)
    for doc, score in scores.items():
        top_k.add(doc, score)
    return top_k
```

- We only care about relative document scores: optimizations that do not change document rankings are safe.
- If query terms appear once, and all query terms are equally important, the query vector q has one nonzero entry for each query term and all entries are equal.
- Order is preserved if we use a query vector where all values are 1. This is equivalent to summing up document term scores as matching scores.

To see how we can speed up matching, let's think a little harder about the matching scores. Our ultimate goal is to rank the documents and pick the top k. We don't really care about the matching scores, except as a means to that end. That means that optimizations which produce different matching scores are fair game, as long as they are guaranteed to produce the same ranking as the correct matching scores.

Let's consider one example. Suppose that the query doesn't repeat any terms, or at least that we're happy to ignore repetition when it occurs. Suppose, too, that we want to treat all query terms as being equally important. If these suppositions are true, then what does the query vector look like? It will have one nonzero value for each query term, and those nonzero values will all be equal.

That leads us to an easy optimization. If we change all the nonzero values in the query vector to 1, we haven't changed the document order. We won't be calculating the correct matching scores anymore, but we don't care about that. If the query vector values are all 1, then we don't have to calculate expensive query term scores and we don't even need to multiply the query term scores by the document term scores. We can just add up document term scores for the terms which appear in the query. That's what this code does. We're skipping the query score calculation and just adding up document scores. If our two suppositions are correct – no query terms are repeated, and we weight all query terms equally – then the ranking produced by this algorithm is exactly the same as the ranking produced by the previous algorithm.



Faster, Approximate Scoring

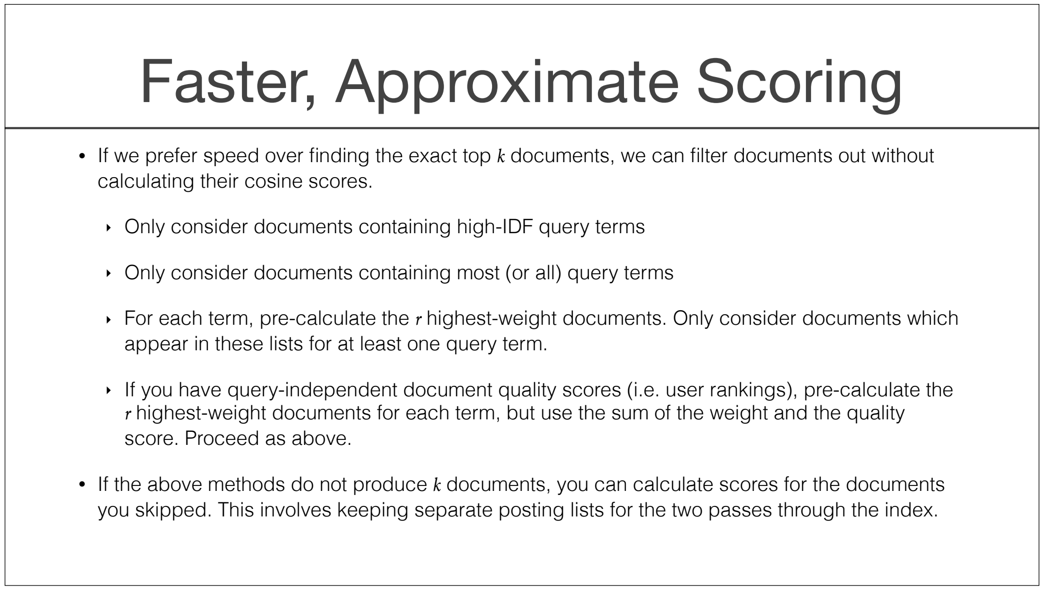
- If we prefer speed over finding the exact top *k* documents, we can filter documents out without calculating their cosine scores.
  - Only consider documents containing high-IDF query terms
  - Only consider documents containing most (or all) query terms
  - For each term, pre-calculate the *r* highest-weight documents. Only consider documents which appear in these lists for at least one query term.
  - If you have query-independent document quality scores (i.e. user rankings), pre-calculate the *r* highest-weight documents for each term, but use the sum of the weight and the quality score. Proceed as above.
- If the above methods do not produce *k* documents, you can calculate scores for the documents you skipped. This involves keeping separate posting lists for the two passes through the index.

If we want to do even better, we can relax a little about returning the top k documents and instead try to return k documents which are approximately as good as the top k. This isn't always a safe strategy. It depends on the size of your collection and the algorithm's ability to find a lot of good matches. If you generally have thousands of relevant documents for your users' queries and you just have to return 10 good ones, then these tips are probably worth it. If query performance is very bad, or you have a small collection but a lot of diverse information needs to match, you may want to stick to exact scoring.

Having said that, let's consider some things we can do.

First, we can ignore documents which don't contain any high IDF query terms. These terms are likely to be the more important terms in the query, so you often get similar performance. Since higher IDF means the terms are in fewer documents, these terms typically have shorter posting lists, which further speeds up search.

You can also filter out documents that don't have most, or all, all of the query terms. For many queries, this can match the information need better than high-scoring documents for just a few query terms. However, be careful of "syntactic glue" words, or redundant words the user may enter. When these words are present, this strategy can be more dangerous.



Faster, Approximate Scoring

- If we prefer speed over finding the exact top *k* documents, we can filter documents out without calculating their cosine scores.
  - Only consider documents containing high-IDF query terms
  - Only consider documents containing most (or all) query terms
  - For each term, pre-calculate the *r* highest-weight documents. Only consider documents which appear in these lists for at least one query term.
  - If you have query-independent document quality scores (i.e. user rankings), pre-calculate the *r* highest-weight documents for each term, but use the sum of the weight and the quality score. Proceed as above.
- If the above methods do not produce *k* documents, you can calculate scores for the documents you skipped. This involves keeping separate posting lists for the two passes through the index.

As a third approach, you can build what are called "champion lists" for each term in your index. These are lists of the r highest-weight documents, and represent the r documents which emphasize that term the most strongly. At query time, you can take the union of the champion lists for all query terms and only consider documents in that set. This tends to do better for simpler information needs that are expressed in just a few terms. When queries contain multiple words that need to match simultaneously, such as multi-word city names like "New York City," champion lists can ignore words that are good for the entire query and instead favor words that are just good for one of its terms.

In many IR systems, we have ways to measure document quality that are independent of any query. For instance, in product search we often have user ratings for the products. These can be used for another optimization. Choose a matching function that includes both the similarity score and the quality score. For instance, your document score could be 0.3 times the quality plus 0.7 times the matching score. Then for each term, pre-calculate the list of the r highest-scoring documents using that formula to build your champion list. From that point, use the champion list as we described above. There are more sophisticated ways to mix information about document quality and query matching, and we'll cover them later, but this simple formula is one approach.

Any of these methods can be used situationally. For instance, if a user runs a query with just a couple of high-IDF terms and a lot of low-IDF terms, filter documents that match the high-IDF terms. If another user submits a query where most terms have similar IDF scores, don't filter the documents.

With a little foresight, you can also arrange to add more documents if any of these filters finds fewer than k results. For instance, if you're using champion lists then you can keep two posting lists for each term: one for the champions, and one for all the other documents. Then if you don't find k documents, you can go back and calculate scores for the others. This should hopefully happen rarely enough that you still get most of the savings from champion lists.

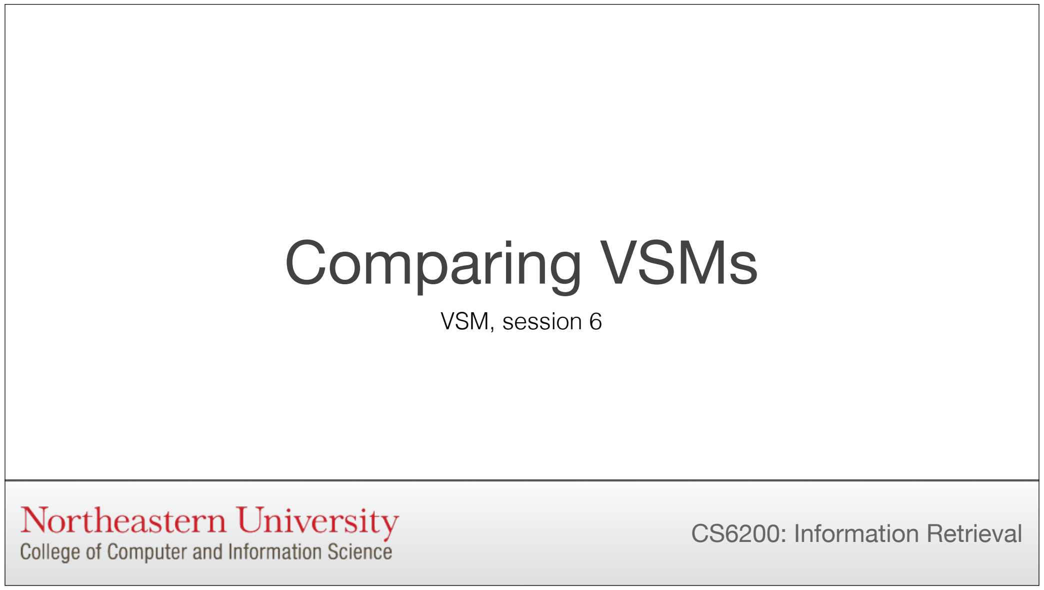
Wrapping Up

- There are many optimizations we can consider, but they focus on a few key ideas:
    - For exact scoring, find ways to mathematically deduce the document ranking without calculating the full cosine similarity.
    - For approximate scoring, choose either query terms or documents which you can safely ignore in order to reduce the necessary calculations without reducing search quality by too much.
- Next, we'll compare the performance of several VSM techniques.

Let's wrap up.

[read bullets]

Thanks for watching.



Comparing VSMs

VSM, session 6

Northeastern University
College of Computer and Information Science

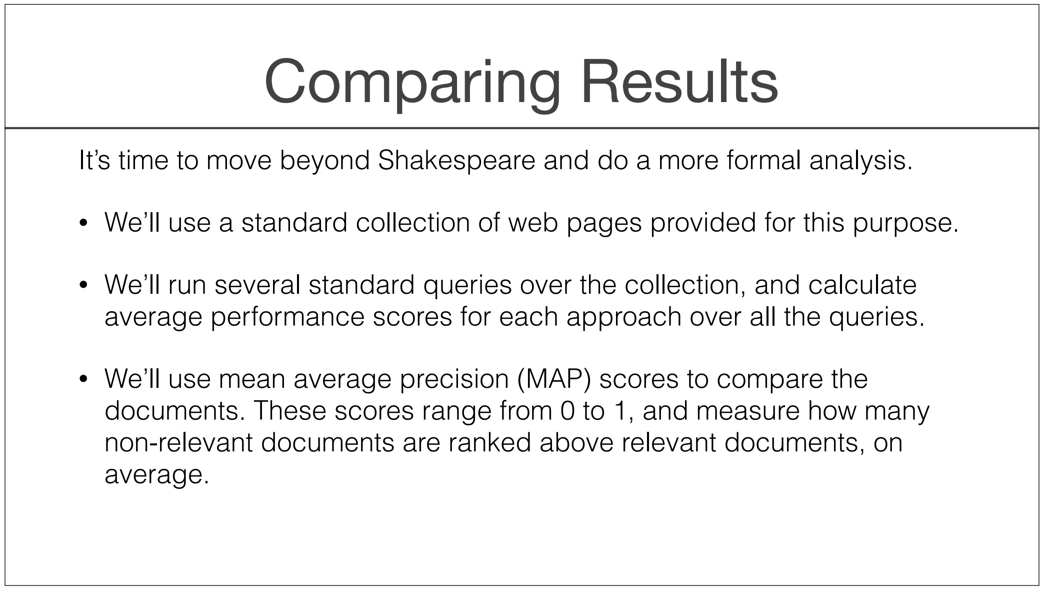
CS6200: Information Retrieval

It’s time for another shoot off! This time, we’re taking a closer look at several vector space models to see how well they really work.

Meet the Contenders

- In this shoot-off, we compare several VSM scoring functions to assess their quality. We will try to see the difference between:
  - Linear vs. logarithmic TF scores
  - IDF vs. no document frequency information
  - Dot product vs. cosine similarity vs. approximate cosine similarity

We've seen a lot of functions we could use for term scores and matching scores. The goal of this video is to find out how they're different from each other. We'll ask:
whether logarithmic term frequency scores really do better than normal term frequency scores,
whether using document frequency information such as IDF helps,
and we'll compare the dot product to cosine similarity and the document length normalization function we saw previously.

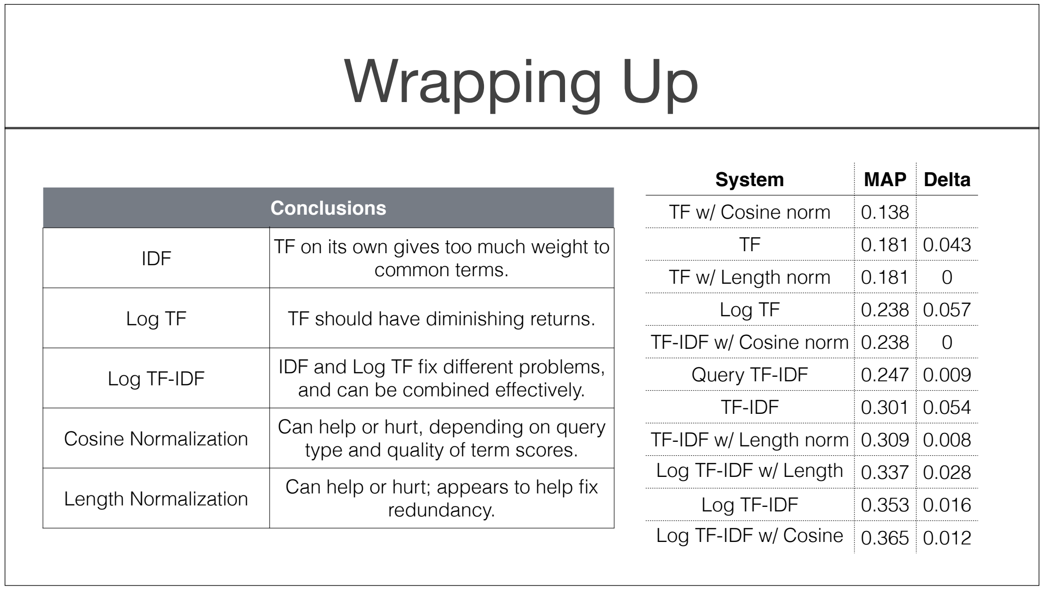


Comparing Results

It’s time to move beyond Shakespeare and do a more formal analysis.

- We’ll use a standard collection of web pages provided for this purpose.
- We’ll run several standard queries over the collection, and calculate average performance scores for each approach over all the queries.
- We’ll use mean average precision (MAP) scores to compare the documents. These scores range from 0 to 1, and measure how many non-relevant documents are ranked above relevant documents, on average.

[read bullets]



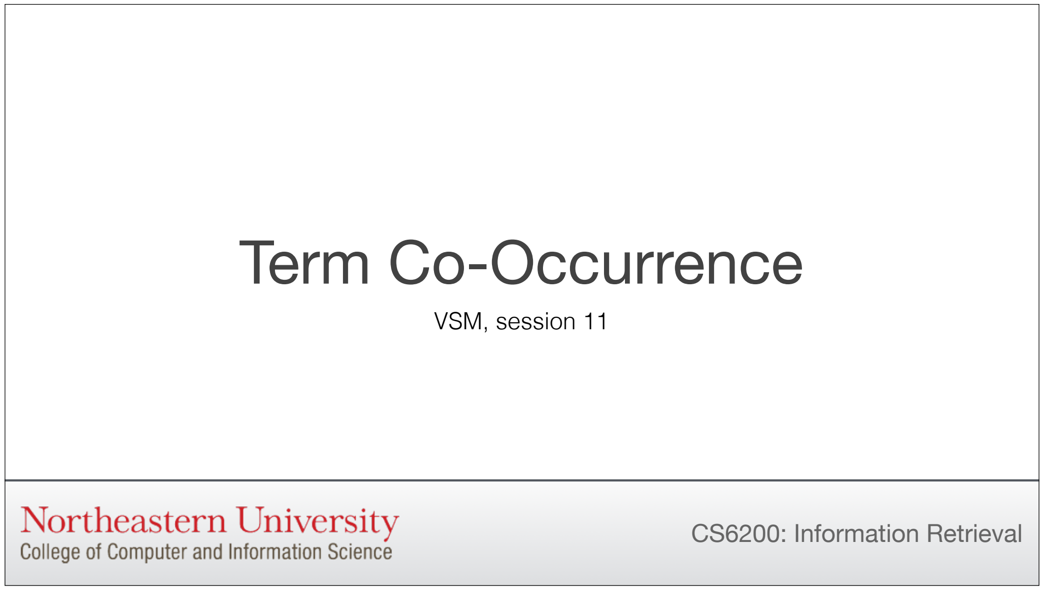
Wrapping Up

Conclusions	
IDF	TF on its own gives too much weight to common terms.
Log TF	TF should have diminishing returns.
Log TF-IDF	IDF and Log TF fix different problems, and can be combined effectively.
Cosine Normalization	Can help or hurt, depending on query type and quality of term scores.
Length Normalization	Can help or hurt; appears to help fix redundancy.

System	MAP	Delta
TF w/ Cosine norm	0.138	
TF	0.181	0.043
TF w/ Length norm	0.181	0
Log TF	0.238	0.057
TF-IDF w/ Cosine norm	0.238	0
Query TF-IDF	0.247	0.009
TF-IDF	0.301	0.054
TF-IDF w/ Length norm	0.309	0.008
Log TF-IDF w/ Length	0.337	0.028
Log TF-IDF	0.353	0.016
Log TF-IDF w/ Cosine	0.365	0.012

Let's wrap up.
The table on the right shows the MAP for each of our runs. It's sorted from the worst to the best performer. Notice that all the TF systems did the worst, the TF-IDF systems did better, and the log TF-IDF systems did the best. This suggests that the term score function matters more than the normalization function. Normalization is a refinement that can't help you if your term scores aren't high quality.

In fact, cosine normalization actually hurt us when we used low quality TF scores. The better the term score function, the better cosine normalization did, until it finally got the top score overall with the best term scores. There's a hidden factor at play here, too: all these results apply to the CACM document collection we used, but these documents and queries all share certain properties that won't necessarily hold up for other document collections. How can we tell which approaches will help or hurt in other contexts?



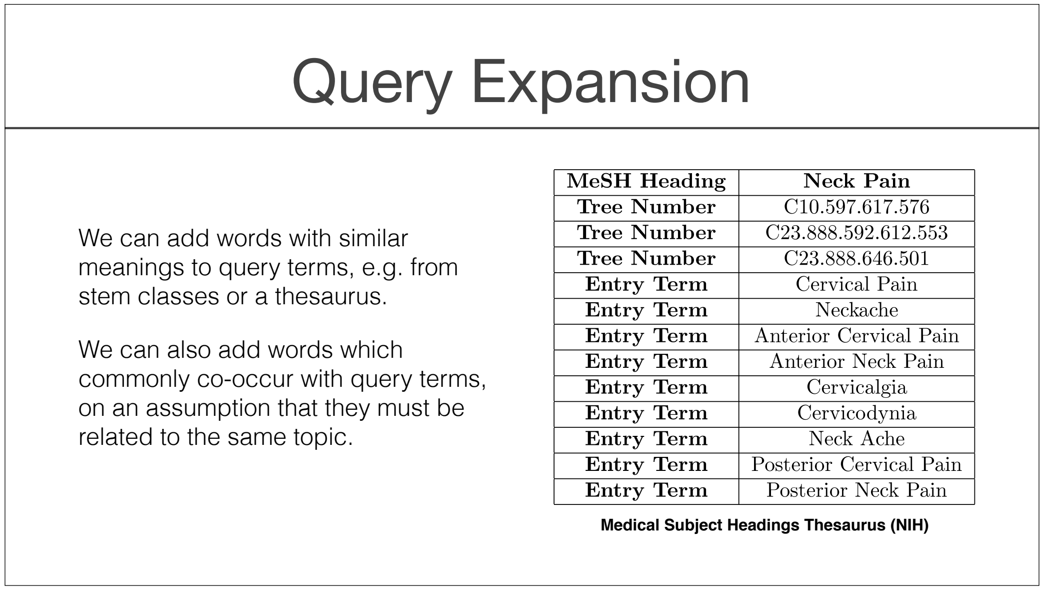
Term Co-Occurrence

VSM, session 11

Northeastern University
College of Computer and Information Science

CS6200: Information Retrieval

This session covers several co-occurrence measures. We’re focusing on term co-occurrence here, but these measures can be used for many other statistical tasks. For instance, you might be interested in which users have reviewed the same product, or which web pages link to the same URL. For now, let’s restrict the conversation to term co-occurrence for query expansion.



Query Expansion

We can add words with similar meanings to query terms, e.g. from stem classes or a thesaurus.

We can also add words which commonly co-occur with query terms, on an assumption that they must be related to the same topic.

MeSH Heading	**Neck Pain**
Tree Number	C10.597.617.576
Tree Number	C23.888.592.612.553
Tree Number	C23.888.646.501
Entry Term	Cervical Pain
Entry Term	Neckache
Entry Term	Anterior Cervical Pain
Entry Term	Anterior Neck Pain
Entry Term	Cervicalgia
Entry Term	Cervicodynia
Entry Term	Neck Ache
Entry Term	Posterior Cervical Pain
Entry Term	Posterior Neck Pain

Medical Subject Headings Thesaurus (NIH)

In query expansion, we look for words on the query's theme to improve matching with relevant documents. We've discussed building stem classes to find terms semantically related to the query terms. Another approach is to apply a thesaurus. A general thesaurus is often too broad to use for query expansion, but there are many focused thesauri on more specific topics. You can see the Medical Subject Headings Thesaurus here. This is build by NIH, and relates specific medical conditions, their symptoms, and their treatments. These can be useful tools.

Another approach is to search your document collection for terms that commonly co-occur with the query terms. The hope is that words which frequenty occur with query terms are expressing the same topic. You could build a list of these related words for every word in your collection, and have them available for fast access at query time. But how can we measure how strongly two terms are associated in the collection?

Term Association Measures

There are many measures of term co-occurrence.

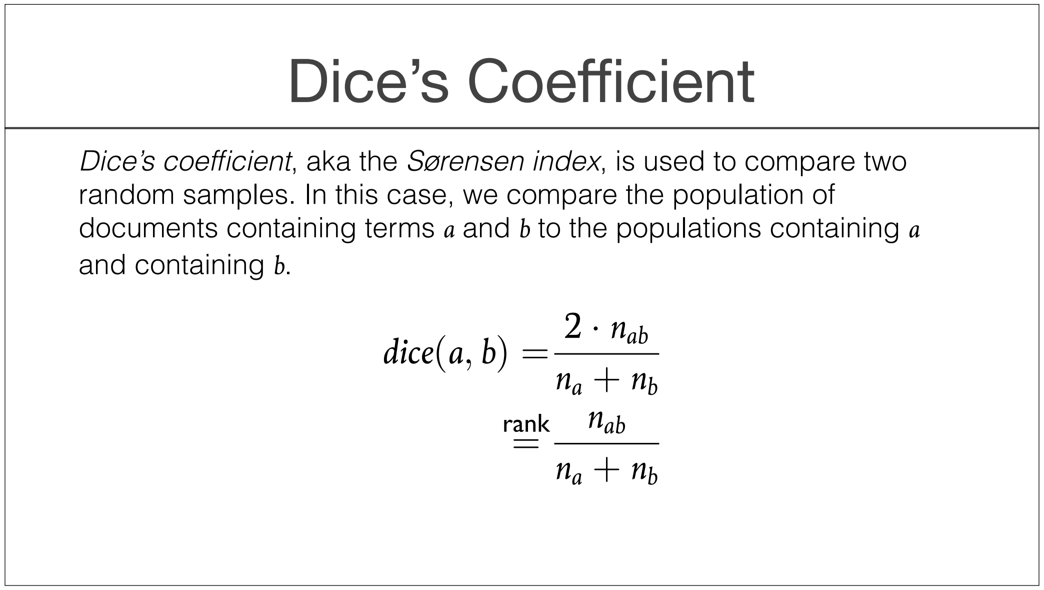
We'll summarize them here, and then examine what each means and how they differ.

Measure	Formula
Mutual Information (MIM)	$\frac{n_{ab}}{n_a \cdot n_b}$
Expected Mutual Inf. (EMIM)	$n_{ab} \cdot \log\left(N \cdot \frac{n_{ab}}{n_a \cdot n_b}\right)$
Chi-square (X^2)	$\frac{(n_{ab} - \frac{1}{N} \cdot n_a \cdot n_b)^2}{n_a \cdot n_b}$
Dice's coefficient (Dice)	$\frac{n_{ab}}{n_a + n_b}$

Measures of co-occurrence.

* These formulas are partial, but rank-equivalent to the full formulas.

Today, we'll cover simplified versions of four different association measures. Mutual Information and Expected Mutual Information come from information theory, and Chi-squared tests and Dice's Coefficient are from statistics. The formulas are given here, for reference, but we'll cover them in more detail in just a moment.
It's worth mentioning that these formulas are not the full versions of these measures. However, they are rank-equivalent to the full formulas, so if you're just using them to sort terms it's faster and simpler to use these forms.
Now let's take a look at each formula, and then look at some data to see how they do.



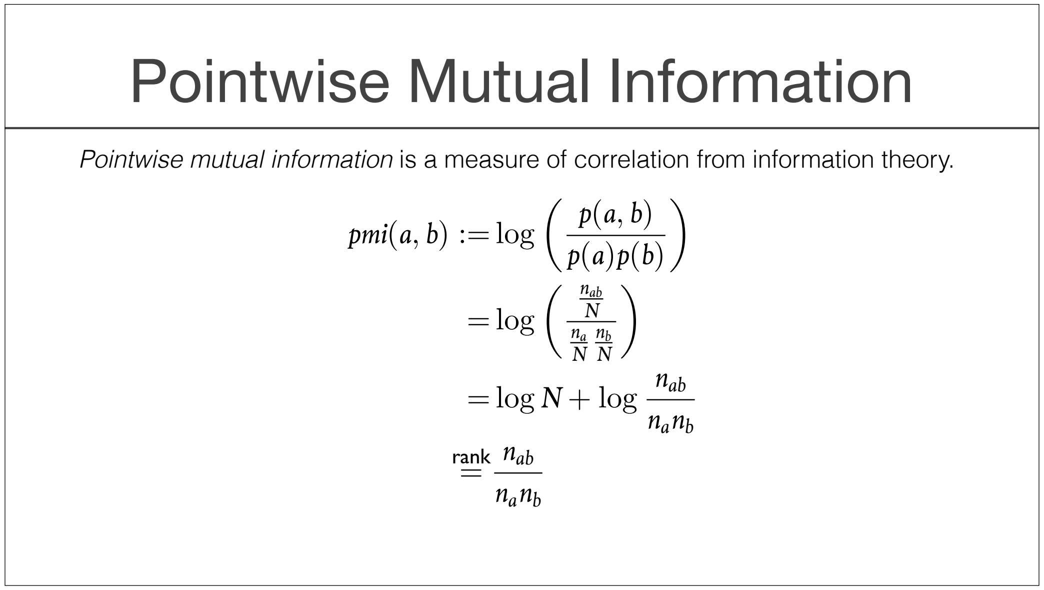
Dice's Coefficient

Dice's coefficient, aka the *Sørensen index*, is used to compare two random samples. In this case, we compare the population of documents containing terms a and b to the populations containing a and containing b.

$$dice(a, b) = \frac{2 \cdot n_{ab}}{n_a + n_b}$$
$$\overset{\text{rank}}{=} \frac{n_{ab}}{n_a + n_b}$$

Dice's coefficient imagines that we have random samples of two events: the event that term a occurs in a document, and the event that term b occurs. It compares how often these events occur together to the total number of times either event occurs.

Let's take a look at the simplified formula n_ab / n_a + n_b. n_ab is the number of documents in the index which contain both term a and term b. n_a is the number of documents that contain term a, and n_b is the number of documents containing term b. If a and b always occur together, then n_ab is going to equal n_a and n_b, and this formula will equal 1/2. The more they occur without each other, the bigger n_a + n_b will become relative to n_ab, so the smaller the number will get.



Pointwise Mutual Information

Pointwise mutual information is a measure of correlation from information theory.

$$
\begin{aligned}
pmi(a,b) &:= \log\left(\frac{p(a,b)}{p(a)p(b)}\right) \\
&= \log\left(\frac{\frac{n_{ab}}{N}}{\frac{n_a}{N}\frac{n_b}{N}}\right) \\
&= \log N + \log\frac{n_{ab}}{n_a n_b} \\
&\overset{\text{rank}}{=} \frac{n_{ab}}{n_a n_b}
\end{aligned}
$$

Pointwise mutual information measures how correlated two random events are to each other. That is, if you know that term a appears in a document, how much information does that give you about whether term b will appear?

If we use base 2 for the logarithm, the amount of information is measured in bits. It's going to have the largest magnitude when knowing whether term a appears in a document lets you predict with perfect accuracy whether term b appears, and be closest to zero when they're totally independent.

The rank-equivalent formula on the bottom of the slide is similar to Dice's coefficient, except that the denominator grows faster as the terms appear without each other more often. This produces a different ordering that punishes terms much more harshly for not co-occurring as often. You could argue that it punishes them too harshly.

Expected Mutual Information

Expected mutual information corrects a bias of pointwise mutual information toward low frequency terms.

$$
\begin{aligned}
emim(a,b) &\propto P(a,b) \cdot \log \frac{P(a,b)}{P(a)P(b)} \\
&= \frac{n_{ab}}{N} \log \left(N \cdot \frac{n_{ab}}{n_a \cdot n_b} \right) \\
&\overset{\text{rank}}{=} n_{ab} \cdot \log \left(N \cdot \frac{n_{ab}}{n_a \cdot n_b} \right)
\end{aligned}
$$

Expected mutual information tries to correct this bias. It’s still measuring the level of dependence between the two random events, but this function is smoother and gives larger values for the lower-frequency terms.
You can still see our approximation of mutual information inside the log function. We’re essentially just scaling this up by multiplying by n_ab outside the logarithm, so that we tend to pay more attention to co-occurrences even when one of the two terms is relatively infrequent compared to the other.

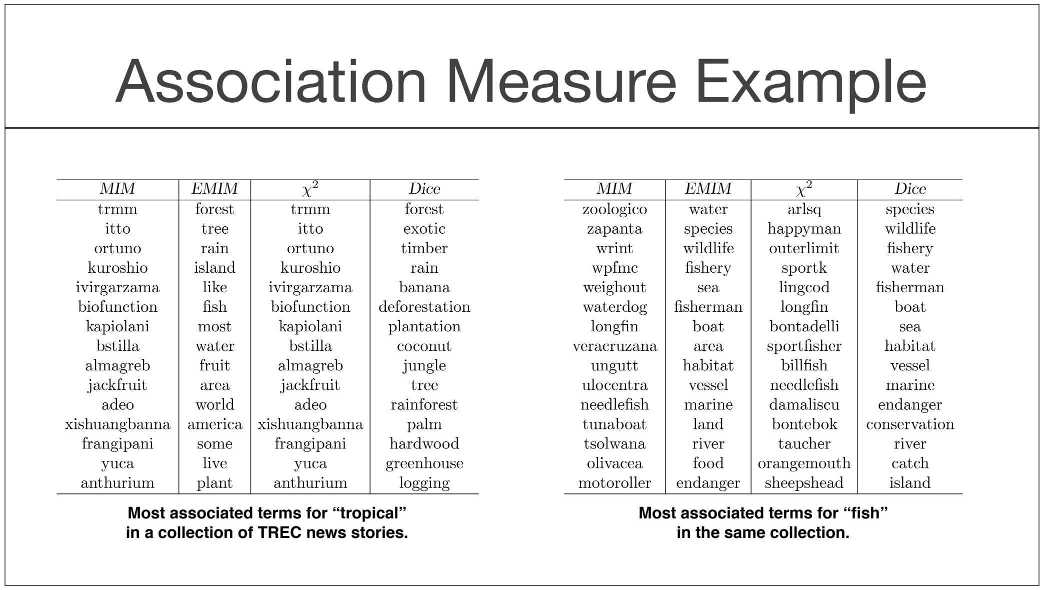
Pearson's Chi-squared Measure

Pearson's Chi-squared test is a test of statistical significance which compares the number of term co-occurrences to the number we'd expect if the terms were independent. (This is also not the full form of this measure.)

$$chi2(a,b) = \frac{\left(n_{ab} - N \cdot \frac{n_a}{N} \cdot \frac{n_b}{N}\right)^2}{N \cdot \frac{n_a}{N} \cdot \frac{n_b}{N}}$$

$$\overset{\text{rank}}{=} \frac{\left(n_{ab} - \frac{1}{N} \cdot n_a \cdot n_b\right)^2}{n_a \cdot n_b}$$

Chi^2 is the last of our term association measures. This is a statistical significance test that's used to measure whether the co-occurrences happen by chance, or whether they happen because the words are really related to each other. We'll talk more about significance testing in the module on evaluation. For now, it's enough to point out that the denominator is the same as for pointwise mutual information, but the numerator is looking at the squared difference between the number of co-occurrences and something related to how often each of the terms occurs on its own. If the two terms are very common, we expect the co-occurrences are more likely to happen by chance. If they're very rare, but always happen with each other, they are more likely to be related. That's the key insight behind this measure.
Now let's take a look at how these four measures perform on real data to get a sense of what they do differently.



Association Measure Example

MIM	*EMIM*	χ^2	*Dice*
trmm	forest	trmm	forest
itto	tree	itto	exotic
ortuno	rain	ortuno	timber
kuroshio	island	kuroshio	rain
ivirgarzama	like	ivirgarzama	banana
biofunction	fish	biofunction	deforestation
kapiolani	most	kapiolani	plantation
bstilla	water	bstilla	coconut
almagreb	fruit	almagreb	jungle
jackfruit	area	jackfruit	tree
adeo	world	adeo	rainforest
xishuangbanna	america	xishuangbanna	palm
frangipani	some	frangipani	hardwood
yuca	live	yuca	greenhouse
anthurium	plant	anthurium	logging

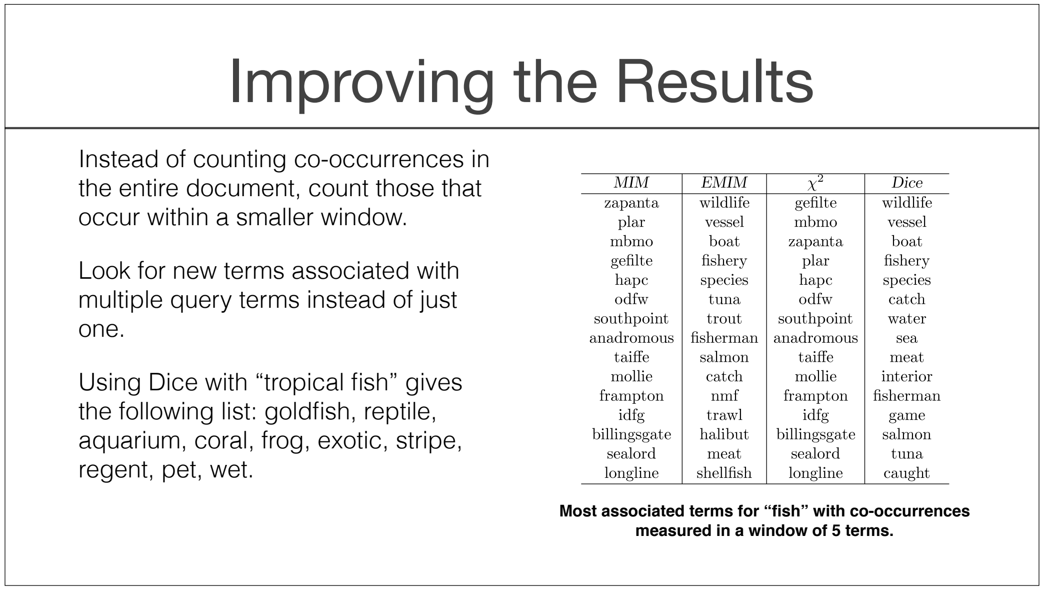
Most associated terms for "tropical" in a collection of TREC news stories.

MIM	*EMIM*	χ^2	*Dice*
zoologico	water	arlsq	species
zapanta	species	happyman	wildlife
wrint	wildlife	outerlimit	fishery
wpfmc	fishery	sportk	water
weighout	sea	lingcod	fisherman
waterdog	fisherman	longfin	boat
longfin	boat	bontadelli	sea
veracruzana	area	sportfisher	habitat
ungutt	habitat	billfish	vessel
ulocentra	vessel	needlefish	marine
needlefish	marine	damaliscu	endanger
tunaboat	land	bontebok	conservation
tsolwana	river	taucher	river
olivacea	food	orangemouth	catch
motoroller	endanger	sheepshead	island

Most associated terms for "fish" in the same collection.

Suppose the user has run the query "tropical fish," and we want to expand the query by adding terms related to each of the query terms. These tables show the 15 highest-ranking terms with each of the four association measures, for each of the two query terms.

The distributions for point wise mutual information, labeled MIM here, and for chi^2, are fairly similar. The distributions for expected mutual information and dice's coefficient resemble each other more than the other two. What's going on here? Well, MIM and chi^2 are focusing on the lowest-frequency terms that tend to appear with the query terms. The words in this list tend to be very rare, and to almost always appear with the query term when they do appear. That isn't necessarily what we want: these terms might be too infrequent to point out relevant documents. They are also rare enough that the terms for the query term "tropical" are unlikely to show up in documents about the query term "fish," and vice versa.
The lists for EMIM and Dice's coefficient are better: they don't focus as much on extremely rare terms. They still have the problem, though, that they aren't related to the overall query. They are focused on one particular query term at the expense of the other.
How can we fix this?



Improving the Results

Instead of counting co-occurrences in the entire document, count those that occur within a smaller window.

Look for new terms associated with multiple query terms instead of just one.

Using Dice with “tropical fish” gives the following list: goldfish, reptile, aquarium, coral, frog, exotic, stripe, regent, pet, wet.

MIM	*EMIM*	χ^2	*Dice*
zapanta	wildlife	gefilte	wildlife
plar	vessel	mbmo	vessel
mbmo	boat	zapanta	boat
gefilte	fishery	plar	fishery
hapc	species	hapc	species
odfw	tuna	odfw	catch
southpoint	trout	southpoint	water
anadromous	fisherman	anadromous	sea
taiffe	salmon	taiffe	meat
mollie	catch	mollie	interior
frampton	nmf	frampton	fisherman
idfg	trawl	idfg	game
billingsgate	halibut	billingsgate	salmon
sealord	meat	sealord	tuna
longline	shellfish	longline	caught

Most associated terms for “fish” with co-occurrences measured in a window of 5 terms.

A few tricks can help us focus on terms more related to the overall query. They mainly involve filtering out some of the co-occurrences in order to focus on the ones we think really matter.

First, we want to filter terms out of the list that occur in the same documents by accident. We can accomplish this by only counting co-occurrences which happen within some fixed distance of the query term. So when we’re looking for terms related to “fish,” we’ll only consider co-occurrences within 10 words of each occurrence of “fish” in each document from our collection. The intuition here is that the document might be talking about a lot of things, but the words closest to the query term are most likely to be closely related to its topic. The table on the right shows an improved list for “fish” that only counts co-occurrences which occur within a window of 5 terms from the word “fish.”

As a second improvement, we can just count co-occurrences when they occur simultaneously for all query terms. If you use Dice’s coefficient on both terms, “tropical” and “fish,” you get a much better list of terms for query expansion.

Wrapping Up

Using term association measures to select words for query expansion can help improve retrieval performance.

However, it can also worsen performance if care is not taken to provide meaningful context. This approach can suffer from “topic drift.”

In our next session we’ll look at relevance feedback, which finds terms for expansion based on information about which documents are relevant to the query.

Let’s wrap up.
Our goal here is to select good words to add to the query vector. Our strategy is to find words which tend to co-occur with the query terms in the document collection. We’ve seen a lot of ways to do that, and found that it’s important to use measures that don’t focus too much on extremely rare terms. It’s also important to filter the list of co-occurrences intelligently so we’re only counting the ones that are likely to find good expansion terms.
Next, we’ll look at a way to modify the query based on knowledge about the relevance of the best-matching documents for the query.
Thanks for watching.