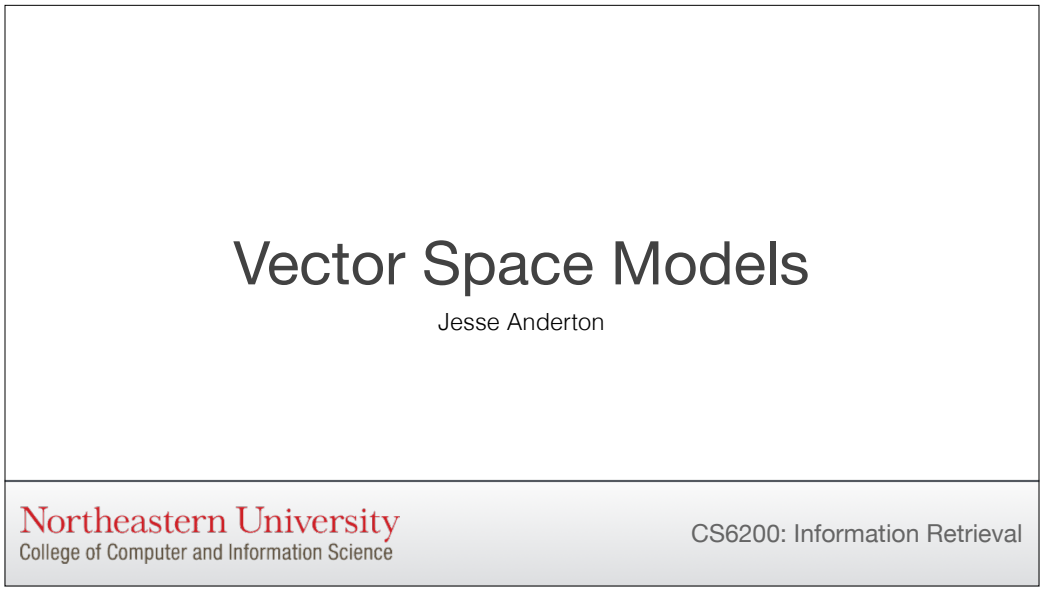
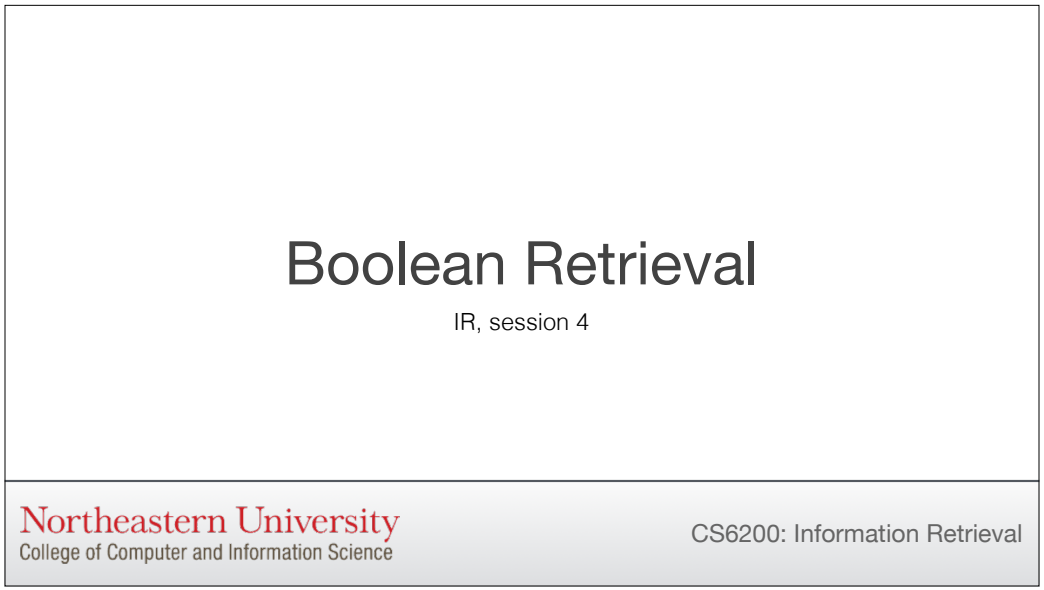


In the first module, we introduced Vector Space Models as an alternative to Boolean Retrieval. This module discusses VSMs in a lot more detail. By the end of the module, you should be ready to build a fairly capable search engine using VSMs. Let's start by taking a closer look at what's going on in a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



This video introduces Boolean Search, one of the first and most successful IR systems. Over the next few videos in this module, we'll show how to build a simple but complete Boolean retrieval system. This will give us a chance to introduce all the main components of an IR system. Then, in future modules, we'll examine each of these components in more detail.

Ad-hoc Search

- In **ad-hoc search**, a user's information need is expressed as a list of keywords. We find relevant documents from the collection by performing *semantic matching* between the query and documents.
- **Semantic Matching** is measuring the similarity of meaning between two texts, e.g. a query and a document. There are many approaches with varying degrees of sophistication.
- For our first attempt at semantic matching, we will use a Boolean retrieval model. In this model, queries are Boolean expressions. Words match only themselves, and complex information needs are expressed by building complex queries.

Boolean Retrieval is a system for performing ad-hoc search.

[1] As we saw in the last video, in ad-hoc search, a user's information need is expressed as a list of keywords. We find relevant documents from the collection by performing what's called "semantic matching" between the query and the documents in our collection.

[2] Semantic Matching is measuring the similarity of meaning — semantics — between two texts. In this case, we're comparing a query and a document. There are many approaches, and we'll talk about quite a few in the module on Ranking, but for now we'll keep it simple.

[3] For our first attempt, we'll Boolean Retrieval. In Boolean Retrieval, a query is a Boolean expression. A keyword matches itself, and only itself. That means we know nothing about synonyms or other nuances of language. In order to express a complex information need, the user will have to build a complex query that combines a lot of keywords using Boolean operators like AND and OR. Let's see what this means.

Boolean Matching

In Boolean Retrieval, queries are keywords combined using Boolean operators. The result is the set of documents satisfying the expression.

- “caesar” returns all documents which contain the term “caesar.”
- “caesar AND brutus” returns documents which contain both terms.
- “caesar OR brutus” returns documents which contain either term.

Real world systems have many such operators, built out of Boolean logic and set theory.

Boolean Matching is very literal. We combine keywords using standard Boolean operators: AND, OR, and so on. Documents are filtered using the set operations specified by the query. A document is considered relevant if and only if it satisfies the Boolean expression. We leave it to the user to make sure the query precisely expresses their information need.

For instance, the query “caesar” will return every document which contains “caesar,” whether it’s talking about the emperor or the salad.

“caesar AND Brutus” will return every document which contains caesar and also contains brutus. “caesar OR brutus” returns every document which contains either keyword.

In order to actually implement this, we’ll introduce a data structure to keep track of which keywords apply to which documents.

Term Incidence Matrix

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	…
Antony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	1	0	0	0	0	
Cleopatra	1	0	0	0	0	0	
mercy	1	0	1	1	1	1	
worser	1	0	1	1	1	0	
…							

► **Figure 1.1** A term-document incidence matrix. Matrix element (t, d) is 1 if the play in column d contains the word in row t, and is 0 otherwise.

- A **term incidence matrix** records the terms used in each document in our collection.
- It has one row for each term found in any document, and one column for each document in our collection.
- We will use the example in the textbook by Manning et al, and use the plays of William Shakespeare as our collection.

[1] A Term Inciastedence Matrix is a data structure we can use to run Boolean Retrieval. It records the terms used in each of the documents in our collection.

[2] Each row represents one of the words in our vocabulary. There's one row for each word found in any document. Columns represent our documents. The entry at row i and column j will be one if word i appears in document j, and zero otherwise. We can run queries using bitwise operators on the rows of this matrix.

If you look at the matrix on the left, you can see some of the plays of William Shakespeare as columns, and terms found in those plays as rows. For instance, the word "Antony" appears in Antony and Cleopatra, Julius Caesar, and Macbeth, but not in any of the others.

[3] Let's use this matrix to work through an example, introduced in the IR textbook by Chris Manning.

Example

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	…
Antony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	1	0	0	0	0	
Cleopatra	1	0	0	0	0	0	
mercy	1	0	1	1	1	1	
worser	1	0	1	1	1	0	
…							

► **Figure 1.1** A term-document incidence matrix. Matrix element (t, d) is 1 if the play in column d contains the word in row t, and is 0 otherwise.

For this example, we'll look for words in the plays of William Shakespeare. Each play gets a column, and each word gets a row. If we wanted to run the query "Caesar," we'd just look for ones in the row for "Caesar." Let's run a more interesting query, and look for "Caesar AND Brutus AND NOT Calpurnia."

We'll start by searching for Caesar and Brutus. We'll list the rows for each, and combine them using bitwise AND. The result is one whenever both of the inputs were one.

Next, we need to figure out what NOT Calpurnia is. To get that, we just flip the zeros to ones and the ones to zeros in the Calpurnia row.

Finally, we combine the two output rows using bitwise AND to calculate our final answer. And the winners are: Antony and Cleopatra, and Hamlet.

Commercial Systems

- Boolean Retrieval has been commercially successful, with products surviving for decades.
- It has mainly been used by trained searchers in vertical search and enterprise search engines, e.g. in the leading legal and journalistic search engines from the 1970s on.
- This happened somewhat to the frustration of IR researchers, whose more advanced matching models took years to become widely adopted.

Information need: Information on the legal theories involved in preventing the disclosure of trade secrets by employees formerly employed by a competing company.
Query: "trade secret" /s disclos! /s prevent /s employe!

Information need: Requirements for disabled people to be able to access a workplace.
Query: disab! /p access! /s work-site work-place (employment /3 place)

Information need: Cases about a host's responsibility for drunk guests.
Query: host! /p (responsib! liab!) /p (intoxicat! drunk!) /p guest

Example Boolean Queries

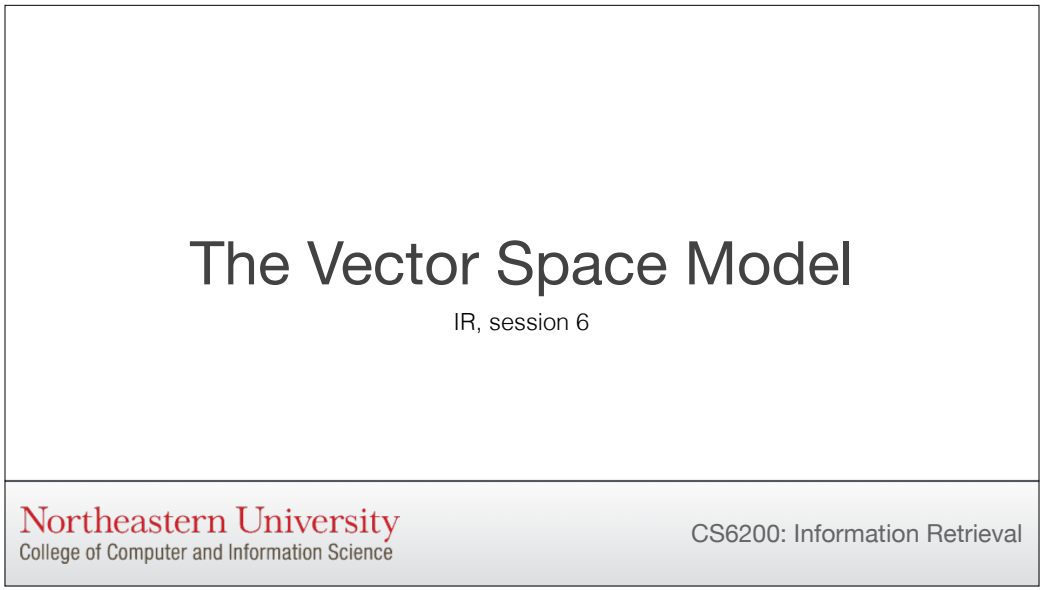
Now that we've seen a really simple Boolean Retrieval system, let's talk about what people use commercially. Boolean Retrieval was the dominant IR system in commercial software for decades, from the 60s and 70s until it was finally replaced by more sophisticated semantic matching systems in the 90s as the Internet gained in popularity.
This was a little bit of an "I told you so" moment for the IR research community, whose improved models had been around for quite some time by then. We'll look at some of those systems soon.
First, let's look at the query language for a real commercial Boolean Retrieval system. These examples come from Manning's textbook.
These systems were mainly used by expert searchers who were trained in the custom query language used by the search software. There are a few interesting query language features on display. Let's look at the first query. First, you can surround a phrase with quotation marks. Also, if you look at the second and the last keywords, you can see the prefix of a word followed by an exclamation point. This is used to refer to any word that starts with that prefix. It's a simple approach to what's called stemming, which we'll talk about quite a bit in the module on query understanding.
Finally, these queries contain a lot of proximity operators. The slash-s means that the terms on either side should be found in the same sentence, and slash-p means they should be found in the same paragraph. There's a slash-3 in the second query which means that the terms should be found within three words of each other.
So these languages can get fairly complicated. There are quite a few other operators people have built into query languages. The textbook by Croft has a section on a query language for a search engine called Galago if you're interested in seeing what a modern query language looks like.

Wrapping Up

- Boolean Retrieval is a simple model, but is precisely defined and easy to implement.
- Its users feel they have control over the system, and can build up complex queries iteratively to refine the results.
- Next, we'll see how to refine our example to handle large scale collections.

Let's wrap up. Although Boolean Retrieval is simple, it's easy to define, implement, and extend. Its users can be very loyal, partly because it gives them a lot of precise control over what's considered relevant for their queries. They typically run a few exploratory queries, building up a complex expression of their information need over the course of a search session. These properties have made Boolean Retrieval a very successful model.

In our next video, we'll flesh out our simplistic approach to Boolean Retrieval and introduce the key data structure used to make IR possible at scale. Thanks for watching.



When I introduced Boolean Retrieval, I mentioned that the research community was somewhat relieved when the commercial world started to adopt some of its better retrieval models in the 90s. We'll look at one of those improved models today. The vector space model uses a nice mathematical generalization of query/document matching which, unlike the set-based Boolean Retrieval model, has a lot of flexibility for tuning retrieval performance. But first, let's look at why Boolean Retrieval isn't good enough.

Flaws of Boolean Retrieval

- Boolean Retrieval has a few big shortcomings that limit its usefulness.
- Since it's based on set membership, a document either matches the query or doesn't. There's no way for some documents to match more than others.
- It also has little flexibility to allow us to try to match the *information need*. We often want to refine the submitted query before we run it, but the complex operators in Boolean Retrieval queries make that difficult.
- It is helpful to use *simpler* queries in order to support *more complex* processing behind the scenes. The computer should generally do more work than the user.

[read 1] Several of these shortcomings are fundamental to the system, so in order to fix them we'll need to change our basic approach.

First, [read 2]. We could get away with this when IR systems used comparatively tiny collections of documents and searchers were willing to spend hours carefully tuning complex queries. Once these users were satisfied with their queries, they often looked at every document the system retrieved. On a collection as large as the Internet, though, you'll often find thousands of relevant documents for your query. The problem becomes choosing the BEST relevant documents.

Another problem is that it has little… [read 3]

We'll talk at length about the query refinement techniques available to us, but for now let's just say that the query a retrieval system actually runs is seldom the exact query the user types in. The system changes it in order to improve search results.

[read 4]

Another Look at the Matrix

- Let's take another look at the term incidence matrix to find a way forward.
- This (rotated) matrix has a column for every word in the vocabulary, so any document can be a vector in this matrix.
- We can also represent keyword queries as vectors. From this point of view, queries are just "short documents."
- Our goal is to find documents which are "on the same topic" as the query.

	Antony	Brutus	mercy	worser	...
Julius Caesar	1	1	0	0	
The Tempest	0	0	1	1	
Hamlet	0	1	1	1	
Othello	0	0	1	1	
⋮					
antony and brutus	1	1	0	0	

[read 1]

[read 2]

[read 3] This notion of representing queries and documents in the same way turns out to be very powerful. Remember that the core task of IR is to match documents and queries. It's much more natural to compare objects with the same representation.

[read 4] We'll look at a lot of ways to approach this problem, but let's start simple.

More Nuanced Term Scores

- So far, our matrix uses binary scores to indicate the presence of a term.
- If we allow more values, we can also indicate how well the document matches that particular term. Terms more central to the document's topic should have larger values.
- Let's use *term frequency (TF)*: the number of occurrences of a term in a document. This reflects an intuition that a term which appears more often is more central to the document.

Binary Scores

	Antony	Brutus	mercy	worser	...
Julius Caesar					
The Tempest					
Hamlet					
Othello					
⋮					

TF Scores

	Antony	Brutus	mercy	worser	...
Julius Caesar					
The Tempest					
Hamlet					
Othello					
⋮					

[read 1]
So the term vector for Julius Caesar would be 1 1 0 0,
The Tempest is 0 0 1 1,
Hamlet is 0 1 1 1, and
Othello is 0 0 1 1.
[read 2] Remember that our goal is to move beyond the simple binary set membership world, which is perfectly captured by ones and zeros, and into a more nuanced world where documents can be just a little relevant or highly relevant. When we match a given term from a query, we'd like the documents' scores for that query term to be higher if those documents are more likely to be good matches. If the query has multiple terms, we'll combine the scores from all its terms so that documents which match all terms will end up with higher scores than documents that just match one.
One simple term score function we can use is term frequency: the number…[read rest of 3]

More Nuanced Term Scores

- So far, our matrix uses binary scores to indicate the presence of a term.
- If we allow more values, we can also indicate how well the document matches that particular term. Terms more central to the document's topic should have larger values.
- Let's use *term frequency (TF)*: the number of occurrences of a term in a document. This reflects an intuition that a term which appears more often is more central to the document.

Binary Scores

	Antony	Brutus	mercy	worser	...
Julius Caesar					
The Tempest					
Hamlet					
Othello					
⋮					

TF Scores

	Antony	Brutus	mercy	worser	...
Julius Caesar					
The Tempest					
Hamlet					
Othello					
⋮					

The TF scores for Julius Caesar turn out to be 128, 379, 0, and 0

The scores for The Tempest are 0, 0, 8, and 1

Hamlet gets 0, 1, 6, and 1, and

Othello gets 0, 0, 5, and 2.

We can already see that Julius Caesar is a much better result for the term "Brutus" than Hamlet is. Brutus is a central character of the first play, and the second only mentions him once, as a reference to his betrayal of Caesar. This distinction is obvious to anyone who knows the two plays, but is invisible to Boolean Retrieval's binary term scores.

Matching Scores

- In order to sort documents by their relevance to a particular query, we will generate a *query matching score* for each document and sort by that.
- The matching score should be a function of two vectors which outputs a number. The number should be *larger* when the vectors express *more similar* topics.
- If they use the same words, they probably express similar topics. A reasonable starting place is to use the dot product of the vectors.

$$s := \langle Q, D \rangle$$
$$= Q \cdot D$$

$$\begin{matrix} Q & & D & & s \end{matrix}$$
$$\begin{bmatrix} \\ \\ \\ \\ \\ \\ \end{bmatrix} \cdot \begin{bmatrix} \\ \\ \\ \\ \\ \\ \end{bmatrix} = $$

[show 1] We compare a document and a query by calculating a matching score for the two term vectors. Those matching scores will impose an ordering on the collection of documents, and we'll use that ordering as our final ranked list.

That's the big idea: each document gets a matching score, and the matching scores allow us to sort the documents. So, what properties should the matching score function have? [pause]

I mentioned earlier that using the same mathematical representation for queries and documents makes it easier to compare them. In the Vector Space Model, we're representing documents and queries as term vectors, so [read 2].

That is, we want to get larger numbers when the query and document are more related, and smaller numbers when they're less related. But how can we look at two vectors and decide how related they are?

Matching Scores

- In order to sort documents by their relevance to a particular query, we will generate a *query matching score* for each document and sort by that.
- The matching score should be a function of two vectors which outputs a number. The number should be *larger* when the vectors express *more similar* topics.
- If they use the same words, they probably express similar topics. A reasonable starting place is to use the dot product of the vectors.

$$s := \langle Q, D \rangle$$
$$= Q \cdot D$$

$$\begin{matrix} Q & & D & & s \\ \begin{bmatrix} \\ \\ \\ \\ \\ \end{bmatrix} & \begin{bmatrix} \\ \\ \\ \\ \\ \end{bmatrix} & \cdot & \begin{bmatrix} \\ \\ \\ \\ \\ \end{bmatrix} & = \end{matrix}$$

The intuition of the Vector Space Model is that [show 3] if they use the same words, they probably express similar topics. One function we could use to compare term vectors is the dot product. This function isn't the best choice we could make, but it serves our purposes for now.

Let's see how this works. Suppose our query is for "caesar and brutus," and our vocabulary includes caesar, brutus, mercy, and worser. The term vector for the query is going to be [1 1 0 0]. Using TF term scores, the play Julius Caesar has the term vector [128 379 0 0]. Since the query didn't repeat any terms, the dot product of these vectors is just the sum of the scores for the two query terms, or 507.

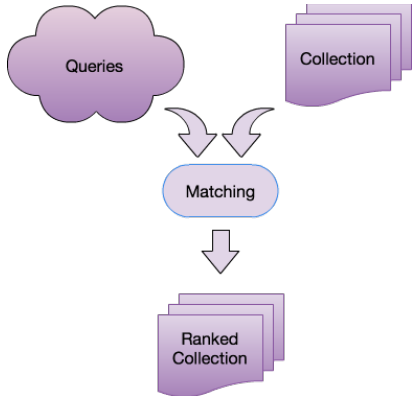
As another example, consider the play Hamlet with the term vector [0 1 6 1]. This play doesn't mention caesar and only mentions brutus once, so its matching score is 1. That means Julius Caesar comes before Hamlet in the ranking, as it should.

But something else happened that doesn't happen in Boolean Retrieval: even though Caesar doesn't appear in Hamlet at all, we still got a matching score for the document. In Boolean Retrieval, the document would have been excluded entirely. This kind of matching is a little fuzzier than in the strict world of Boolean Retrieval. It allows us to find the best matching documents in the collection, even if they don't match all of the query terms.

Query/Document Matching

In order to run a search engine using the vector space model:

1. Create an inverted index containing the TF of each term in each document.
2. When the user enters a query, create a term vector using the TF of each query term.
3. Iterate over each document, calculating the matching score of the document's term vector with the query's term vector.
4. Sort all documents by their matching scores in order to get a ranked list, and show the user the best 1000 documents.



Let's put it all together. Here are the steps for performing retrieval using the Vector Space Model.

[read 0]

[read 1] This is done in advance, as soon as we get the documents in our collection.

[read 2]

[read 3]

Finally, [read 4]

The number 1000 is arbitrary, but happens to be a standard choice. In Google, for instance, you initially see the top 10 results but you can go through the top 1000. In research, 1000 is a convenient cut off because it gives us enough results to perform meaningful analysis on their quality, without trying to impose an order on the whole collection.

Wrapping Up

- In the module on vector space models, we'll discuss this model in much more depth. Here are a few things that can change:
  - Which terms do we include?
  - Which *term score* function do we use?
  - Which *matching score* function do we use?
  - How do we avoid iterating over all indexed documents for every query?
- In our next session, we'll compare Boolean Retrieval to the Vector Space Model to see how they differ.

Let's wrap up. We've seen two retrieval models now: Boolean Retrieval and Vector Space Models. In the next module, we're going to take a much deeper look at VSMs. Vector-based matching scores are pretty flexible, and you can do a lot of things to improve their performance beyond the simple example we just saw. In particular, consider these changes:
[read 2] In particular, what terms could we add to the query to improve performance?
[read 3] TF has some drawbacks, which we'll look at later.
[read 4] and finally,
[read 5] We generally don't have time to scan the whole index, and the user is only likely to read the first few documents we return anyway. How do we read just a small part of the index, but make sure that the best documents we find are the best documents in the collection?
[read 6]
Thanks for watching!

Vector Spaces

- In mathematical terms, Vector Space Models treat documents and queries as vectors in $\mathbb{R}^n$, where n is the vocabulary size.
- We define an inner product $\langle q, d \rangle$ which we use as a similarity function between query and document. Higher values should correspond to greater query relevance.
- These models uses a "bag of words" assumption: word order makes no difference, and there is no concept of meaningful phrases.

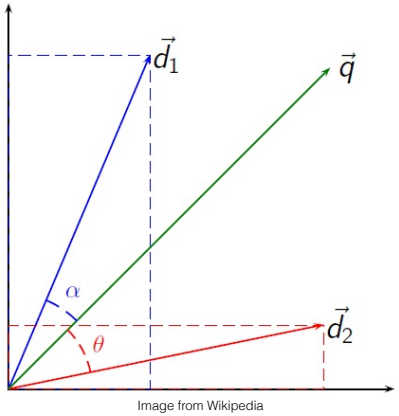


Image from Wikipedia

[read 1]

We have a number for each term in the vocabulary which indicates how related that term is to the document's content. Generally, only the values for terms which actually appear in the document are nonzero. However, as we'll see, it's very useful to add some carefully-chosen nonzero terms to the vectors for queries: they're so short that they don't really express their topics adequately, and it makes a big difference if we can flesh them out a bit.

Once we have our two vectors, we use an inner product as a similarity funciton… [read 2]. I say "an inner product" because there are many functions we could potentially use for this purpose. The simplest is the dot product, but there are many more. We'll talk about several them later in the module. The inner product function we choose defines what's called an "inner product space," such that vectors within that space use the inner product for their notion of distance. Intuitively, the closer two vectors are to each other, based on their inner product, the more semantic content they should have in common.

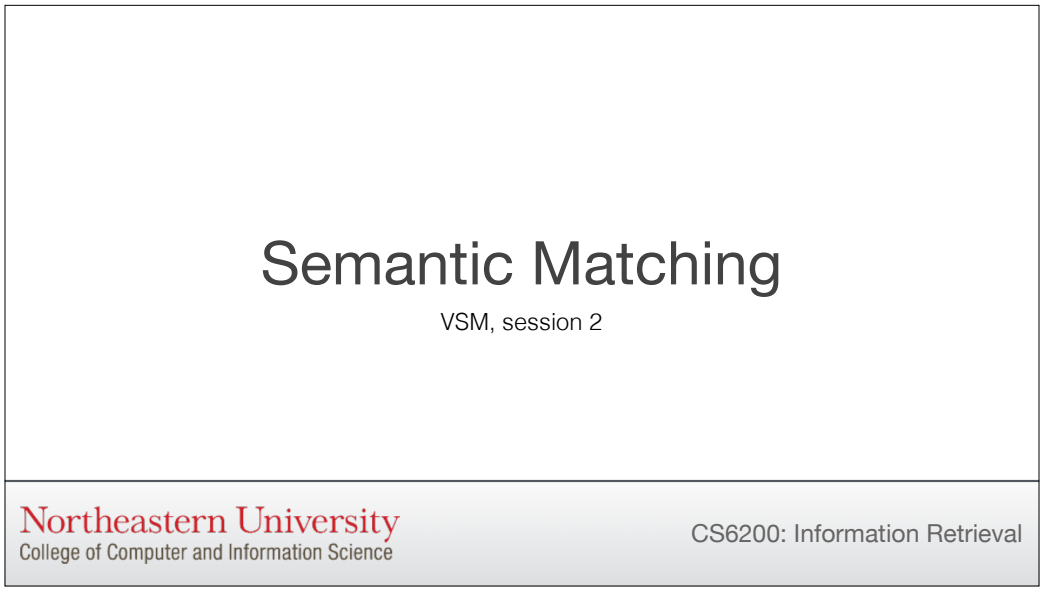
Note that [read 3]. These limitations will be relaxed in future modules, but many very useful models use the bag of words assumption. It turns out that you can get a long way using basic word use and word co-occurrence statistics, without having to figure out the much harder question of understanding what each document is really saying.

Big Questions for VSMs

Our goal is to produce the best ranking we can get, using VSMs. We need to answer these questions:

- What's the best term score function we can use?
- What's the best way to calculate matching scores?
- How can we do these calculations efficiently, to minimize query processing time?
- How can we extend the query to better represent its topic?

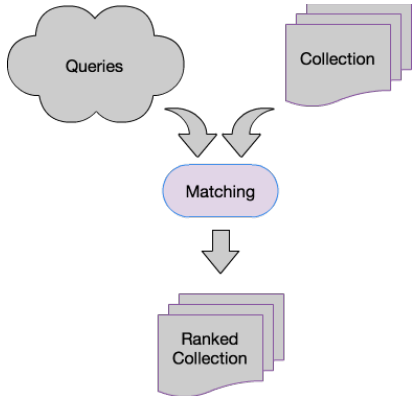
[read bullets]



Before we dive into the technical details of VSMs let's talk about the bigger picture problem: semantic matching. This is a fundamental task of IR: matching documents to queries based on whether they mean the same thing.

Semantic Matching

- The concept of semantic matching between queries and documents is key to search, but useful for many other tasks.
  - In machine translation, we want to match passages with identical semantics but in different languages.
  - In advertising, we want to match ads for products related to what the user is searching for.
  - In genetics, we want to match amino acid sequences that produce proteins with similar effects.



As it turns out, semantic matching is involved in a lot of tasks beyond information retrieval.
[read two]
[read three]
[read four] in this case, the semantics of the amino acid sequence involve how they will be interpreted by the protein building mechanisms of the cell rather than how a human would understand them.

Semantic Matching with VSMs

- In VSMs, we model documents and queries as vectors within some space, and try to find vectors that are “near” each other.
- There are several pitfalls to watch out for with this model:
  - We use the bag of words assumption, and lose phrase information.
  - Our matching score function has to account for document lengths.
  - Queries omit many keywords which the user would include if they were being thorough. The query “President Nixon” is missing the keyword “Richard.”

Bag of Words Assumption

“Paul loves ice cream.”
≅
“Ice cream loves Paul.”
≅
“Ice loves cream Paul.”

Document Length Bias

“Paul loves ice cream.”
<
“Paul loves ice cream.
Paul loves ice cream.
Paul loves ice cream.”

Of course here we’re concerned mostly with Vector space models for IR.
[read 1] This is done by carefully choosing the values we store in the vectors, and the way that we define distance between them.
[read 2]
[read 3] for instance consider these three examples. With the bag of words assumption, these three sentences are semantically identical. To a VSM, “Paul loves ice cream” and ”ice cream loves Paul” are exactly the same sentence.
[read 4] this might not be obvious at first, but the lengths of two documents has a big effect on how distant they are from each other, and doesn't necessarily affect whether they're on the same topic. The vector for “Paul loves ice cream” has a smaller magnitude than the same sentence repeated 3 times, even though the second document doesn't say anything extra.
Also, [read 5]. Presumably the president's full name is relevant to the query.
Let's talk about these three issues in more depth.

Bag of Words Assumption

- There are a few problems with this assumption:
  - We can suffer from query document mismatch when alternative terms are used to specify some semantic concept.
  - Many alternate queries express the same information need, and users often choose different ways to phrase the same concept.
  - We can't make use of phrasal features or language syntax.

Query	Document	Semantic Match?
ny times	new york times	yes
seattle best hotel	seattle best hotels	yes
pool schedule	swimming pool schedule	yes
china kong	china hong kong	no
why are windows so expensive	why are macs so expensive	no

Example due to: Li, H. and Xu, J. 2014. Semantic Matching in Search. Foundations and Trends® in Information Retrieval 7, 5 (Jun. 2014)

Here are some drawbacks of the bag of words assumption.

First, [read 1]. We're representing the meaning of a document as simply the collection of words it contains, so a naive implementation won't be able to match documents that express the same topic in different ways.

Relatedly, [read 2].

Another big drawback is that [read 3].

Document Length Normalization

- With a naive matching score, such as the dot product, longer documents have an unfair advantage due to term repetition.
- Very short documents may be less relevant (but what about a collection of FAQ answers?), but very long documents may also be less relevant.
- We will discuss several ways to normalize term vectors to address this:
  - Instead of TF, consider the fraction of a document the term occupies.
  - Take into account the number of *distinct* terms in the document to account for repetitive documents.
  - Various fast approximations of these approaches.

Document length normalization is another issue we'll cover.

[read 1]

[show 2] Very short documents are often less relevant, but very long documents often have the same problem.

[read rest]

Query Expansion

- The user's query may specify a concept using terms that happen to be missing from a given relevant document.
- We address this by adding terms to the query before performing matching.
- This must be done carefully in order to avoid adding terms not related to the user's information need.

Queries for "Obama family tree"
barack obama family
the obama family
obamas
barack obama geneology
obama's ancestry
obama family tree
barack obama family history

Example due to: Li, H. and Xu, J. 2014. Semantic Matching in Search. Foundations and Trends® in Information Retrieval 7, 5 (Jun. 2014)

Query Expansion is our last topic of the module.

[read 1] Consider this list of queries, all for the same information need. They're expressing the same concept in slightly different ways, and in a naive implementation of VSMs they'll all match slightly different collections of documents.

[read 2] Our goal is for all queries with the same information need to match the most relevant documents to that need.

However, [read 3]. This is called "topic drift," and can cause you to match non-relevant documents. It's challenging to pick the right terms when you have so little information about the information need.

Wrapping Up

- We'll consider solutions to these problems and others in upcoming sessions.
- Spoiler alert: the bag of words assumption is not relaxed in this module, but we do move away from it with Language Models.
- Next, we'll take a closer look at term scores and learn how to emphasize more important terms.

Let's wrap up.
[read bullets]
Thanks for watching.

Term Scores

VSM, Session 3

Northeastern University
College of Computer and Information Science

This module explains what's wrong with using simple term frequency scores for our document vectors, and then explains several useful alternatives.

Flaws of TF

- In the first module, we introduced term frequency $tf_{t,d}$ as a way to measure how much term t contributes to document d's meaning.
- One obvious flaw with this scheme is that some of the most common words, such as "and" or "the," contribute no meaning to a document.
- We'd like a term score which assigns high weight to terms that are both *distinctive* to the document and *frequent* within the document.

Term	Min TF	Max TF	# of Plays
and	426	1,001	37
the	403	1,143	37
die	3	40	37
love	12	171	37
betray	1	6	24
rome	1	110	16
fairy	1	28	10
brutus	1	379	8
verona	5	13	3
romeo	312	312	1

TF of selected terms in Shakespeare's plays

[read 1] This is a good start, but it has some problems.

[read 2] Take a look at this table, giving the TF scores for a selection of words in Shakespeare's plays. The terms with the highest term scores are "syntactic glue" words, such as and and the, which show up in hundreds of times in every play. We clearly can't trust a high TF score by itself to tell us which terms are most important for a document. However, it is giving us useful information for some words. The word "love" also shows up in every play, but the number of occurrences varies dramatically. A romantic play like Romeo and Juliet will include the word much more often than a political tale like Julius Caesar. As an extreme example, consider the term "romeo." This word only shows up in one play, and it shows up 312 times there. These two facts indicate that it's a strong signal of topicality, if "plays about romeo" is your topic.

To summarize, what we're looking for is a term score [read 3]

Let's take a deeper look at the relative frequency of different terms in a corpus. It turns out that, regardless of the language used, term frequencies in any sufficiently large corpus will approximately follow something informally called "Zipf's Law."

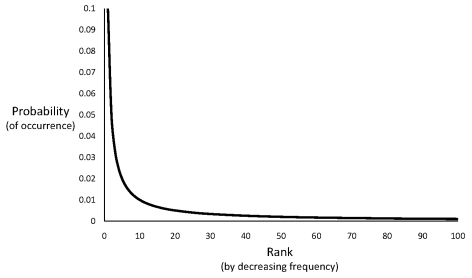
Zipf's Law

- If you sort words by their frequency, the frequency f times the rank r is roughly constant:

$$freq(t) \cdot rank(t) \approx k, \text{ or}$$
$$Pr(t) \cdot rank(t) \approx c$$
for language-dependant c, k

  - Word 2 appears 1/2 as often as word 1
  - Word 10 appears 1/10 as often as word 1, and 1/2 as often as word 5

- Very common words have much higher TF values, but most are "syntactic glue"
- In English, $c \approx 0.1$



Zipf's Law

Zipf's Law states that if you sort all the terms in your corpus by their frequency, the frequency times the rank in that sorted list is roughly constant.
That is, frequency times rank is always pretty close to some constant k. Equivalently, the probability of seeing a term times its rank when all the terms are sorted by frequency is roughly constant.
In practical terms, that means that the second most popular word will show up roughly half as often as the most popular word. The tenth most popular word will appear 1/10 as often as the most popular word, and half as often as the fifth most popular. This is called a "power law" distribution. A handful of words appear extremely frequently, and a majority of the vocabulary is pretty uncommon.
Anybody who's learned multiple languages is already familiar with this concept: you can learn just a few hundred words of a new language and usually be able to get your point across, and as you learn more and more words you use each of them less and less often.
Let's look at a real world data set and see how Zipf's Law holds up there.

How do we fix it?

- We want term scores to be proportional to TF, but we want to discount scores for too-common terms.
- Two common ways to discount:
  - A term's cumulative frequency cf_t is total number of occurrences of term t in the collection.
  - The term's document frequency df_t is the number of documents in the collection which contain term t.
- The most common way to discount is to multiply by $log(D/df_t)$., where D is the number of documents in the collection. This is called IDF, for inverse document frequency, and leads to TF-IDF scores.

$$tf\text{-}idf_{t,d} := tf_{t,d} \cdot \log\left(D/df_t\right)$$

Term	Doc	tf	df	cf	tf df	tf cf	tf-idf
and	King Lear	737	37	25,932	19.92	0.028	0
love	Romeo and Juliet	150	37	2,019	4.05	0.074	0
rome	Hamlet	2	16	332	0.125	0.006	1.68
rome	Julius Caesar	42	16	332	2.625	0.127	35.21
romeo	Romeo and Juliet	312	1	312	312	1	1126.61

Various term score functions.
Why is TF-IDF 0 for "love" in Romeo and Juliet?

So, we know that the highest TF scores are probably useless for semantic matching. How do we fix that?
[read 1]
Many approaches to discounting these terms have been suggested, but in general they reduce the scores when some other measure, like the total number of occurrences of the term in the whole collection, is high.
Let's look at two different ways we could do this. The cumulative frequency, or cf, is the number of times the term appears in the whole collection. The document frequency, or df, is the number of documents in which the term appears.
We commonly discount by multiplying the TF score by the logarithm of the number of documents in the collection divided by the df. This is called the inverse document frequency, or IDF.

How do we fix it?

- We want term scores to be proportional to TF, but we want to discount scores for too-common terms.
- Two common ways to discount:
  - A term's cumulative frequency cf_t is total number of occurrences of term t in the collection.
  - The term's document frequency df_t is the number of documents in the collection which contain term t.
- The most common way to discount is to multiply by $log(D/df_t)$., where D is the number of documents in the collection. This is called IDF, for inverse document frequency, and leads to TF-IDF scores.

$$tf\text{-}idf_{t,d} := tf_{t,d} \cdot \log\left(D/df_t\right)$$

Term	Doc	tf	df	cf	tf df	tf cf	tf-idf
and	King Lear	737	37	25,932	19.92	0.028	0
love	Romeo and Juliet	150	37	2,019	4.05	0.074	0
rome	Hamlet	2	16	332	0.125	0.006	1.68
rome	Julius Caesar	42	16	332	2.625	0.127	35.21
romeo	Romeo and Juliet	312	1	312	312	1	1126.61

Various term score functions.
Why is TF-IDF 0 for "love" in Romeo and Juliet?

Let's compare these different weighting schemes in the table on the right.

The term and shows up in every document. Its TF in King Lear is 737. If you divide that by the DF you get roughly 20, and by CF you get almost zero. Its TF-IDF score is zero, because it appears in every document.

The term rome shows up much more in Julius Caesar than in Hamlet, though it's much less common than "and." If we just discount by df or cf, it still has a lower score than "and." However, its TF-IDF score is higher for Hamlet, and much higher for Julius Caesar. That's exactly what we want.

As an extreme example, the term Romeo, which only shows up in a single play, has a very high TF-IDF score. That's perfect, because this term is a perfect feature in this corpus for finding plays about Romeo.

Nonlinear TF Scaling

- Another problem with TF is that repeated use of a term doesn't necessarily imply more information about it.
- A document with TF=20 seems unlikely to have exactly ten times the information as a document with TF=2.
- We want term repetition to give a document diminishing returns. A common solution is to use the logarithm instead of a linear function.

$$wf_{t,d} := \begin{cases} 1 + log(tf_{t,d}) & \text{if } tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$wf\text{-}idf_{t,d} := wf_{t,d} \cdot idf_t$$

*wf stands for "weight function" and is not an official name.

Another issue with term frequency is [read 1].

[read 2] Maybe it was just more verbose or repetitive, not more informative.

We commonly address this by using the logarithm of the term frequency rather than the TF itself. This has the diminishing returns we want to see for TF scores.

But perhaps you're not happy with using the logarithm: it's just a heuristic, chosen because it happens to have the diminishing returns property. What else could you do?

Normalized TF Scores

- As an alternative to nonlinear scaling, we could consider normalizing the TF scores by the TF of the most popular term in the document.
- The role of a in this formula is to produce scores that change more gradually as $tf_{t,d}$ changes.
- a is commonly set to 0.4.

$$ntf_{t,d} := a + (1 - a)\frac{tf_{t,d}}{max\text{-}tf_d}, 0 < a < 1$$

$$max\text{-}tf_d := \max_{\tau \in d} tf_{\tau,d}$$

Another approach is to normalize the TF scores by the TF of the most popular term in the document. This approach is basically asking, "what fraction of the document is comprised of this term?" If the document is just repetitive, it will probably have a limited vocabulary. These terms won't benefit from having a higher absolute TF, because they still comprise similar fractions of the document. On the other hand, if a document has a wide vocabulary but emphasizes a handful of terms, then they are probably central to the them of a diverse, informative document.

In this particular formulation, we use a parameter, "a," so we get similar scores for terms with very similar TF scores. This is called "smoothing," and will be covered in more detail in the next module.

Wrapping Up

- The term score functions used for VSMs are often heuristics based on some insight into a problem with previously-used functions.
- When we discuss Language Models, we will implement these insights in a more principled way.
- We showed solutions for several problems with raw TF scores, but the resulting scores are far from perfect.
  - We don't separate word senses: "love" can be used to mean many things
  - A popular word can nevertheless have pages devoted to it, and be the topic of some query. Can we effectively search for Shakespearean plays that tell love stories using TF-IDF scores?

Let's wrap up.

[read bullets]

Thanks for watching.

Matching Scores

TVM, Session 4

Northeastern University
College of Computer and Information Science

Now that we've looked at some different term score functions we can use, let's focus on how to compare two term vectors to each other.

Finding Similar Vectors

- Imagine that we have perfect term scores: our vectors exactly capture the document's (or query's) meaning.
- How can we compare two of these vectors so we can rank the documents?
- Let's try a similarity function based on the Euclidean distance between the vectors.
- What's wrong?
  - In the query's term vector, TF=1.
  - Documents with TF > 1 are further from the query, so have lower similarity.

$$dist(q, d) := \sqrt{\sum_t (q_t - d_t)^2}$$

$$sim(q, d) := \frac{1}{1 + dist(q, d)}$$

Play	TF	Distance	Similarity
Henry VI, part 2	1	0	1.0
Hamlet	1	0	1.0
Antony and Cleopatra	4	4.59	0.179
Coriolanus	109	165.40	0.006
Julius Caesar	379	578.9	0.002

Plays for query "brutus" using TF-IDF term scores

Imagine that we are incredibly brilliant, and have found the perfect term score function. Our term vectors perfectly capture the document's meaning. We still have a big challenge: how can we compare two of these term vectors so we can rank the documents?

For starters, let's try a similarity function based on the Euclidean distance between vectors. Euclidean distance is the standard distance function you're probably used to. It considers the two vectors as points in a Euclidean space, and measures how far those points are from each other. It's the square root of the sum of the squared distances along each axis. This is exactly like Pythagorus' theorem, scaled up to any number of dimensions.

Finding Similar Vectors

- Imagine that we have perfect term scores: our vectors exactly capture the document's (or query's) meaning.
- How can we compare two of these vectors so we can rank the documents?
- Let's try a similarity function based on the Euclidean distance between the vectors.
- What's wrong?
  - In the query's term vector, TF=1.
  - Documents with TF > 1 are further from the query, so have lower similarity.

$$dist(q, d) := \sqrt{\sum_t (q_t - d_t)^2}$$

$$sim(q, d) := \frac{1}{1 + dist(q, d)}$$

Play	TF	Distance	Similarity
Henry VI, part 2	1	0	1.0
Hamlet	1	0	1.0
Antony and Cleopatra	4	4.59	0.179
Coriolanus	109	165.40	0.006
Julius Caesar	379	578.9	0.002

Plays for query "brutus" using TF-IDF term scores

For our similarity function, we'll use 1 over 1 plus the Euclidean distance. This gets bigger when the distance gets smaller, so closer documents will be more similar.

How does it work? Well, let's keep things simple and delete every word from Shakespeare's plays except for "brutus." We'll use TF term scores, and compare just the six plays where the term appears.

By this measure, Henry VI and Hamlet are perfect matches for the query "brutus," with TF 1. Coriolanus and Julius Caesar, both of which have characters named Brutus, are considered terrible matches, with very low matching scores.

In fact, the more often the term Brutus appears in a play, the lower the similarity gets. What's going wrong here?

Let's think about exactly what we're doing. In the query's term vector, brutus has TF=1. Since we're using Euclidean distance, documents with TF=1 will match perfectly; TF=0 and TF=2 will be identical to each other and just a little worse than TF=1, and as TF gets higher than 2 the document's vector moves further and further from the query vector. This is the opposite of what we want! This similarity function is a failed experiment.

Dot Product Similarity

- We used the dot product in module 1. How does that work?
- For many documents, it gives the results we want.
- However, imagine building a document by repeating the contents of some other document.
- Should two copies of Julius Caesar really match better than a single copy?
- Should "The Complete Plays of Shakespeare" match better than individual plays it contains?

$$sim(q, d) := q \cdot d$$

Play	TF	Similarity
Henry VI, part 2	1	2.34
Hamlet	1	2.34
Antony and Cleopatra	4	9.38
Coriolanus	109	255.65
Julius Caesar	379	888.91
Julius Caesar x 2	758	1777.83
Julius Caesar x 3	1137	2666.74

Plays for query "brutus" using TF-IDF term scores

What if we just use the dot product of the two vectors? We tried this in the last module, and it appeared to do pretty well.

One reason this doesn't work well is that it unfairly favors repetitive documents. If you made a new play by copying Julius Caesar twice, that new play would match everything the original play matched, but twice as much.

This similarity function is fast and works fairly well, so it does get used sometimes. However, there's a more principled way to compare these vectors.

Cosine Similarity

- Cosine Similarity solves the problems of both Euclidean-based similarity and the dot product.
  - Instead of using distance between the vectors, we should use the *angle* between them.
  - Instead of using the dot product, we should use a length-normalized dot product. That is, convert to unit vectors and take their dot product.

$$sim(q, d) := \frac{q \cdot d}{\|q\| \cdot \|d\|} = \frac{q \cdot d}{\sqrt{\sum_i q_i^2} \cdot \sqrt{\sum_i d_i^2}} = \frac{q}{\sqrt{\sum_i q_i^2}} \cdot \frac{d}{\sqrt{\sum_i d_i^2}}$$

Play	TF	Similarity
Henry VI, part 2	1	0.002
Antony and Cleopatra	4	0.004
Coriolanus	109	0.122
Julius Caesar	379	0.550
Julius Caesar x 2	758	0.550

Plays for query “brutus” using TF-IDF term scores

Cosine Similarity solves the problems with our prior two similarity functions. It’s based on the intuition that the magnitude of a term vector isn’t very important. What matters is which terms show up in the vector, and what their relative sizes are. In other words, what’s the term vector’s angle?

This function uses the angle between two vectors as the distance between them, and totally ignores their relative lengths.

Another way to look at it is that it converts the two term vectors to unit vectors, with magnitude 1, and then uses the dot product between them.

If we look at the same plays with Cosine Similarity, we get exactly the results we want. Henry VI is the worst match, and Julius Caesar the best. Double Julius Caesar doesn’t change its matching score at all, because it just changes the vector’s magnitude and we’re ignoring that.

However, this function isn’t perfect either.

Approximating Cosine Similarity

- The normalization term for cosine similarity can't be calculated in advance, if it depends on df_t or cf_t.
- For faster querying, we sometimes approximate it using the number of terms in the document.
- This preserves some information about relative document length, which can sometimes be helpful.

$$sim(q, d) \approx \frac{q}{\sqrt{len(q)}} \cdot \frac{d}{\sqrt{len(d)}}$$

Play	TF	Similarity
Henry VI, part 2	1	0.014
Antony and Cleopatra	4	0.056
Coriolanus	109	1.478
Julius Caesar	379	6.109
Julius Caesar x 2	758	8.639

Plays for query "brutus" using TF-IDF term scores

One problem is that [read 1]. If we're using TF-IDF, for instance, we'll need to calculate the term for the denominator at query time by iterating over all the terms in each document we're considering. That gets expensive fast.

[read 2] This isn't quite converting the vectors to unit vectors, but it's close, and much faster to calculate at query time.

It also preserves some [read 3]. In fact, there's another technique that's sometimes used to preserve this information in a more precise way.

Pivoted Normalized Document Length

- Some long documents have many short sections, each relevant to a different query.
- These are hurt by Cosine Similarity because they contain many more distinct terms than average.
- If we normalize by a number less than the length for short documents, and more than the length for long documents, we can give a slight boost to longer documents.
- This comes in both exact and approximate forms.

$$sim(q, d) := \frac{q}{\|q\|} \cdot \frac{d}{a\|d\| + (1-a)piv},$$

$0 < a < 1$; piv determined empirically*

$$\approx \frac{q}{\|q\|} \cdot \frac{d}{au_d + (1-a)piv},$$

u_d is # unique terms in d

* See: http://nlp.stanford.edu/IR-book/html/htmledition/pivoted-normalized-document-length-1.html

Some long documents, such as FAQs, have many [read 1]

[read 2]

There's a simple way to fix this: just normalize by a [read 3]

The way you implement the details depend on exactly what you want to do. [read 4]

If you're interested in learning the details of this approach, follow the URL at the bottom of the slide.

SMART Notation

- VSM weights can be denoted as *ddd.qqq*, where *ddd* indicates the scheme for document weights and *qqq* the scheme for queries. The triples are: term frequency, doc frequency, normalization.
- A common choice is lnc.ltc: document vectors use log term frequency and cosine normalization, and query vectors use log term frequency, IDF, and cosine normalization.

Term frequency		Document frequency		Normalization	
n (natural)	$\mathrm{tf}_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1+\log(\mathrm{tf}_{t,d})$	t (idf)	$\log \frac{N}{\mathrm{df}_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2+w_2^2+\ldots+w_M^2}}$
a (augmented)	$0.5+\frac{0.5\times \mathrm{tf}_{t,d}}{\max_t(\mathrm{tf}_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N-\mathrm{df}_t}{\mathrm{df}_t}\}$	u (pivoted unique)	$1/u$ (Section 6.4.4)
b (boolean)	$\begin{cases}1 & \text{if } \mathrm{tf}_{t,d}>0\\ 0 & \text{otherwise}\end{cases}$			b (byte size)	$1/\mathit{CharLength}^{\alpha}$, $\alpha<1$
L (log ave)	$\frac{1+\log(\mathrm{tf}_{t,d})}{1+\log(\mathrm{ave}_{t\in d}(\mathrm{tf}_{t,d}))}$				

► **Figure 6.7** SMART notation for tf-idf variants. Here *CharLength* is the number of characters in the document.

Image from: http://nlp.stanford.edu/IR-book/html/htmledition/document-and-query-weighting-schemes-1.html

Let's bring this all together. There's a standard notation for the term scores and matching scores for VSMs. It's called SMART notation, named after one of the first VSMs published in the literature.

In SMART notation, [read 1]. So if you wanted to use log term frequency weights with IDF and cosine similarity, you would call that l (for log TF) t (for IDF) c (for cosine normalization).

A common choice in production systems is [read 2].

There are a few options in here that we didn't cover. Check out the textbook by Manning et al for the details.

Wrapping Up

- Ultimately, the choice of a scoring system to use depends on a balance between accuracy and performance.
- Ignoring document length entirely with cosine similarity is a big improvement over the simple dot product, but it turns out that there are subtle cases when document length information is helpful.
- Next, we'll look at ways to efficiently calculate these scores at query time.