CSE4334/5334 Data Mining

TF-IDF and Similarity

(Put together from many sources)

Sharma Chakravarthy
Department of Computer Science and Engineering
University of Texas at Arlington

Query-Based Search of Documents, Similarity

- We are familiar with queries on Databases
  - Queries are evaluated against the database contents and you get exact answers! (Boolean queries in Information Retrieval (or IR) vocabulary)
- Can we query documents?
  - Can we get exact answers/matches (or binary answers) for the above?
  - What is an exact match when we search? And how useful is it?
- What is a query on a document?
- Further, can we say whether two documents are similar or dissimilar?
  - Based on what they contain, NOT semantics!
- Text or document retrieval pre-dates database query processing!
- It also pre-dates what we know as Google search!

Query-Based Search of Documents, Similarity

- We routinely search for documents (web content) on web
- We can also do it on any document repository!
  - Archived answers from Q/A network
  - Review collection for movies
  - Checking for plagiarism (paper, analysis, articles, programs, …)
  - Emails (Enron emails have been used extensively for research)
  - Reuter articles (we have used both of these in our research)
- What do we input as query to web search (or Google)?
- What is the reason for Google's search success?
  1. PageRank algorithm!
     - cse 6331 covers more on this
  2. Matching relevant documents and score/rank them with respect to a query
     - We will focus on this in this course!

Document Search/Classification

- Consider 3 documents:

  Doc 1: The game of life is an everlasting learning experience
  Doc 2: The unexamined life is not worth living
  Doc 3: never stop learning through experience

- Query: life learning experience
  1. If you look for exact matches, only doc 1 matches it
  2. But if you want relevant answer, doc 3 could be a potential answer
- However, doc 2 is certainly not a match or answer or relevant
- The question is: how can we accomplish 2 above in a principled manner?

Queries and large number of answers

- (Boolean) queries are good for expert users with precise understanding of their needs and the collection.
  - Also ok for applications (using APIs): can easily consume 1000s of results.
- Not good for majority of users.
  - Most users are incapable of writing Boolean queries (or if they can, they think it's too much work).
  - Most users certainly don't want to wade through 1000s of results.
    - This is particularly true of any large collection (web search is one of them)
- We modify queries, to some extent, during web search when we do not get answers we are looking for! (refining the search)
  - Query relaxation is a technique that does this for database queries!
- Trade off between simplicity and expressiveness!

Problem with Boolean search

- Feast or Famine!
- Boolean queries often result in either too few (= 0) or too many (1000s or even Millions of) results.
- Query 1: "*standard user dlink 650*" → 200,000 hits
- Query 2: "*standard user dlink 650 no card found*": 0 hits
  - Thought experiment: come up with queries for google that give 0 or a specific number of total answers
- It takes a lot of skill to come up with a query that produces a manageable number of hits.
  - AND gives too few; OR gives too many

How to specify a query?

- Natural Language
  - Ideal mechanism
  - But inherently hard given ambiguities of language
    - e.g. school – educational institution; group of fish
    - e.g., java – island or coffee flavor
  - Mechanisms such as Question-Answering frameworks focus on sophisticated language models built for specific domains independently.
  - Effectiveness of a language depends upon the type of users, context, domain, … Are programming languages context-free or context sensitive?
- Aside: regular, context-free, and context-sensitive grammars

Back of the Envelope Summary

Specification	Learning Curve	Precision	Utility	Schema/ Source Knowledge
SQL	High	High	Medium-High	High
QBE	Low	High	Medium-High	Medium
Templates	Low	High	Medium	Low
Natural Language	Low-Medium	Medium	Medium-High	Low-Medium
Search	Low	Low	Low	Low
[illegible]	[illegible]	[illegible]	Hig	[illegible]

	Structured	Unstructured
Search	✓?	✓
Query	✓	??

Our Focus

Ranked retrieval models

- Of a large set of documents satisfying a query expression, in ranked retrieval, the system returns an ordering over the (top-k) documents in the collection for a query
- Free text queries: Rather than a query language of operators and expressions, the user's query is just one or more words in a human language (1st Q/A network was in Korean!)
- In principle, these are two separate choices, but in practice, ranked retrieval has normally been associated with free text queries and vice versa
  - DBMSs also now provide ranked retrieval in a limited manner!

Feast or Famine: not a problem in ranked retrieval

- When a system produces a ranked result set, large result sets are not an issue
  - The size of the result set is not an issue as it can be configured!
  - We just show the top *k* (≈ 10) results
  - We don't overwhelm the user
  - What is critical for this to succeed?
  - Premise: The ranking algorithm works!
- Perhaps, this is exactly what differentiates Google from other search engines

Scoring as the basis of ranked retrieval

- We wish to return a small number of documents most likely to be useful to the searcher
- How can we rank-order the documents in the collection with respect to a query?
- Assign a score – say in [0, 1] – to each document
- This score measures how well document and query "match".

Query-document matching scores

- We need a way of assigning a score to a query/document pair
  - Why?
- Let's start with a one-term query
- If the query term does not occur in the document
  - score should be 0
- The more frequently the query term occurs in the document, the higher the score (should be)
- We will look at a number of alternatives for this.

Take 1: Jaccard coefficient

- Recall a commonly used measure of overlap of two sets *A* and *B*
- jaccard(*A,B*) = |*A* ∩ *B*| / |*A* ∪ *B*|
- jaccard(*A, A*) = 1
- jaccard(*A, B*) = 0 if *A* ∩ *B* = 0
- *A* and *B* don't have to be the same size. (Why is this important in search?)
  - The elements of corpus you are searching are not the same size!
- Always assigns a number between 0 and 1.

Issues with Jaccard for scoring

- It doesn't consider *term frequency* (how many times a term occurs in a document) (why?)
- Rare terms in a collection are more informative than frequent terms. Jaccard doesn't consider this information
- We need a more sophisticated way of normalizing for length
- Later in this lecture, we'll use magnitude
- . . . instead of |A ∩ B|/|A ∪ B| (Jaccard) for length normalization.

Take 2: Binary term-document 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

Each document is represented by a binary vector $\in \{0,1\}^{|V|}$

Occurrence/non-occurrence is captured! Matrix is very large! Matrix is also sparse!

Is binary representation good?

Does this discriminate based on the number of occurrences?

Take 3: Term-document count matrix

- Consider the number of occurrences of a term in a document:
  - Each document is a $\boxed{\text{count vector}}$ in $\mathbb{N}^{|V|}$: a column below

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	157	73	0	0	0	0
Brutus	4	157	0	1	0	0
Caesar	232	227	0	2	1	1
Calpurnia	0	10	0	0	0	0
Cleopatra	57	0	0	0	0	0
mercy	2	0	3	5	5	1
worser	2	0	1	1	1	0

  - This seems to be a better representation than the others!

Term frequency tf

- Should we use every word in the document as a term?
- If not, what should we do?
- How should we handle words in different tenses, adjectives, plural etc.?
- Should the positions of words be considered?

Definitions

- Word – A delimited string of characters as it appears in the document
- Term – A "normalized" word (case, morphology, spelling etc.) an equivalence of words
- Token – an instance of a word or token occurring in a document
- Type – The same as a term in most cases: an equivalence class of tokens

Common words

- Stop Words = extremely common words which would appear to be of little value in helping select (or differentiate) documents matching a user need
  - Examples: a, an, and, are, as, at, be, by, for, from, has, he, in, is, it, its, of, on, that, the, to, was, were, will, with
- Stop word elimination used to be standard in older IR systems
- But you may need stop words for phrase queries, e.g. "King of Denmark"
- What is the problem with stop words?

Lemmatization

- Reduce inflectional/variant forms to base form
  - Example: am, are, is → be
  - Example: car, cars, car's, cars' → car
  - Example: the boy's cars are different colors → the boy car be different color
- Lemmatization implies doing "proper" reduction to dictionary headword form (the lemma).
- Inflectional morphology (cutting → cut) vs. derivational morphology (destruction → destroy)

Stemming

- Definition of stemming: Crude heuristic process that chops off the ends of words in the hope of achieving what "principled" lemmatization attempts to do with a lot of linguistic knowledge.
- Language dependent
- Often inflectional and derivational
- Example for derivational: automate, automatic, automation all reduce to automat
- Why don't we do lemmatization?
- Why should we do stemming?

Porter Stemmer: a few rules

Rule	Example
SSES → SS	caresses → caress
IES → I	ponies → poni
SS → SS	caress → caress
S → S	cats → cat

- Need to be careful
  - Reducing care → car is incorrect!
- There are several implementations of stemming and eliminating stop words

Bag of words model (why is it a bag and not a set?)

- Vector representation doesn't consider the ordering of words in a document
- *John is quicker than Mary* and *Mary is quicker than John* have the same vectors
- This is called the bag of words model.
- In a sense, this is a simplification: The positional index is able to distinguish these two documents.
- For now: bag of words model + stemming + stop word elimination
  - Remember, bag is a multi-set with no order!

Term frequency tf

- The term frequency $tf_{t,d}$ of term *t* in document *d* is defined as the number of times *t* occurs in *d*.
- We want to use tf when computing query-document match scores. But how?
- Raw term frequency is not what we want:
  - A document with 10 occurrences of the term is more relevant than a document with 1 occurrence of the term.
  - But not 10 times more relevant.
- Relevance does not increase proportionally with term frequency. Frequency = count in IR

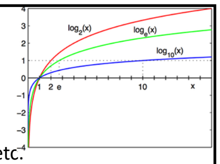
Normalizing counts (or weights)

- Instead of raw counts, normalized values/weights are easier to understand

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	0.35	0.16	0	0	0	0
Brutus	0.009	0.34	0	0.11	0	0
Caesar	0.51	0.49	0	0.22	0.14	0.5
Calpurnia	0	0.02	0	0	0	0
Cleopatra	0.13	0	0	0	0	0
mercy	0.004	0	0.75	0.56	0.71	0.5
worser	0.004	0	0.25	0.11	0.14	0

- Divide each element by the sum of the column in which it appears

Log-frequency weighting



- The log frequency weight of term t in d is

$$w_{t,d} = \begin{cases} 1 + \log_{10} \mathrm{tf}_{t,d}, & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- 0 → 0, 1 → 1, 2 → 1.3, 10 → 2, 1000 → 4, etc.
- Score for a document-query pair: sum over terms *t* in both *q* and *d*:
- score $= \sum_{t \in q \cap d} (1 + \log \mathrm{tf}_{t,d})$

Why is 1 added?

- The score is 0 if none of the query terms is present in the document. Else non-zero. Increases slowly from 1

Term frequency score

- Note
  - the columns (vectors) are very large
    - Tens and hundreds of thousands (based on the size of the documents)
    - Further they are sparse!
- Can we just use this as the score of a document-query pair?
  - Why will this not work well?
- If the document has some rare words, they will never come up as the score does not reflect them in any way!
- How do we include the contribution of rare words to the matching process?

Document Frequency

- Rare terms are more informative than frequent terms
  - Recall stop words
- Consider a term in the query that is rare in the collection (e.g., *lackadaisical*)
- A document containing this term is very likely to be relevant to the query *lackadaisical* *(in the presence of multiple terms)*
- We want a high weight for rare terms like *lackadaisical*
  - *So those documents will come up even with multiple words!*

Document Frequency

- Frequent terms are less informative than rare terms
- Consider a query term that is frequent in the collection (e.g., *high, increase, line*)
- A document containing such a term is more likely to be relevant than a document that doesn't
- But it's not a sure indicator of relevance.
- → For frequent terms, we want high positive weights for words like *high, increase, and line*
- But lower weights than for rare terms.
- We will use document frequency (df) to capture this.

Document Frequency (contd.)

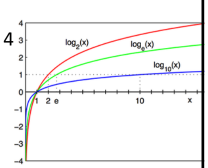
- The document frequency is the number of documents in the collection that the term occurs in
  - Note that this refers to the collection and a term t, not individual documents!
- df_t is the document frequency of *t*: the number of documents that contain *t*
  - df_t is an inverse measure of the informativeness of *t*
  - $\mathrm{df}_t \leq N$ *(total number of documents)*
- This can be a large number for frequent terms!
- For 100,000 documents, if a rare word t occurs in 10 of them, $\mathrm{df}_t = 10$
- Df_t usually has a low value for rare terms!

Idf weight

- We define the idf (inverse document frequency) of *t* by

$$\mathrm{idf}_t = \log_{10}(N/\mathrm{df}_t)$$

= log 10 (100,000/10) = 4



- We use log (N/df_t) instead of N/df_t to "dampen" the effect of idf.
- Without log, it would have been 10,000
- Term frequency for this is: $1+\log_{10} \mathrm{tf}_{t,d} = 1+\log_{10} 10 = 1$
- As t occurrence increases $\mathrm{tf}_{t,d}$ increases and idf_t decreases!
- It turns out that the base of the log is immaterial!

Idf example with N as 1 Million

Term	Df_t	N/df_t	idf_t
Calpurnia	1	1,000,000	6
Animal	100	10000	5
Sunday	1000	1000	4
Fly	10,000	100	3
Under	100,000	10	2
The	1,000,000	1	1

$$\mathrm{idf}_t = \log_{10}(N/\mathrm{df}_t)$$

HW: suppose term t1 occurs 10 Times and a rare term t2 occurs 10 times. For 1 Million documents Check their tf and idf weights.

Does it satisfy the hypothesis we Indicated earlier?

There is one idf value for each term *t* in a collection.

Effect of idf on ranking

- Does idf have an effect on ranking for one-term queries?
  - only if it is a rare term!
- idf effect on ranking one term queries
  - idf affects the ranking of documents for queries with at least two terms
  - For the query capricious person, idf weighting makes occurrences of capricious count for much more in the final document ranking than occurrences of person.

Collection vs. Document frequency

- The collection frequency of *t* is the number of occurrences of *t* in the collection, counting multiple occurrences.
- Example:

Word	Collection frequency	Document frequency
insurance	10440	3997
try	10422	8760

- Which word is a better search term (and should get a higher weight)?
  - It depends on collection size! But in general lower document frequency should get a higher weight!

Tf-idf weighting

- The tf-idf weight of a term is the product of its tf weight and its idf weight.

$$W_{t,d} = (1 + \log_{10} tf_{t,d}) * \log_{10}\left(\frac{N}{df_t}\right)$$

Both terms Start from 0. No fractions

- **N: total number of documents**
- Best known weighting scheme in information retrieval
  - Note: the "-" in tf-idf is a hyphen, not a minus sign!
  - Alternative names: tf.idf, tf x idf
- Increases with the number of occurrences within a document
- Increases with the rarity of the term in the collection

Score of a document for a given query

$$\text{Score}(q,d) = \sum_{t \in q \cap d} \text{tf.idf}_{t,d}$$

Which is $W_{t,d}$

- Score is for a pair – query-document here!
- There are many variants
  - how tf is computed (with/without logs)
  - whether the terms in the query are also weighted!
- Now we have score for each term in a q-d pair, what do we do?
  - We need to map it to similarity of q and d! how do we that?

Binary → count → weight matrix

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	5.25	3.18	0	0	0	0.35
Brutus	1.21	6.1	0	1	0	0
Caesar	8.59	2.54	0	1.51	0.25	0
Calpurnia	0	1.54	0	0	0	0
Cleopatra	2.85	0	0	0	0	0
mercy	1.51	0	1.9	0.12	5.25	0.88
worser	1.37	0	0.11	4.15	0.25	1.95

Each document is now represented by a real-valued vector of tf-idf weights $\in \mathbb{R}^{|V|}$

Documents as vectors

- So we have a |V|-dimensional vector space
- Terms are axes of the space
- Documents are points or vectors in this space
- Very high-dimensional: tens of millions of dimensions when you apply this to a web search engine
- These are very sparse vectors - most entries are zero.

Queries as vectors

- Key idea 1: Do the same for queries: represent them as vectors in the document search space
- Key idea 2: Rank documents according to their proximity to the query in this space
- proximity = similarity of vectors
- proximity ≈ inverse of distance
- Recall: We do this because we want to get away from the you're-either-in-or-out Boolean model.
- Instead: rank more relevant documents higher than less relevant documents

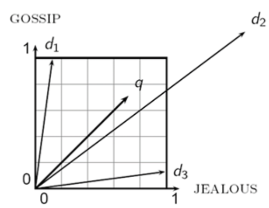
Formalizing vector space proximity

- First cut: distance between two points
  - (= distance between the end points of the two vectors)
- Euclidean distance? Is it a good idea?

- Euclidean distance is a not a good idea
  - because Euclidean distance is large for vectors of different lengths.

Why distance is a bad idea

- The Euclidean distance between *q* and d_2 is large even though the distribution of terms in the query *q* and the distribution of terms in the document d_2 are very similar!



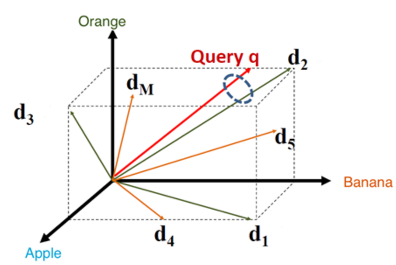
Use angle instead of distance

- Thought experiment: take a document *d* and append it to itself. Call this document *d′*.
- "Semantically" d and d' have the same content
- The Euclidean distance between the two documents can be quite large
- The angle between the two documents is 0, corresponding to maximal similarity.
- Key idea: Rank documents according to angle with query (not magnitude)
  - What angle would you use?

From angles to cosines

- The following two notions are equivalent.
  - Rank documents in <u>decreasing</u> order of the angle between query and document
  - Rank documents in <u>increasing</u> order of cosine(query, document)
- Cosine is a monotonically decreasing function for the interval [0^o, 180^o] as [1, -1]

3 dimensional document vectors and a query



Cosine(query, document)

$$\cos(\vec{q},\vec{d}) = \frac{\vec{q}\bullet\vec{d}}{|\vec{q}||\vec{d}|} = \frac{\vec{q}}{|\vec{q}|}\bullet\frac{\vec{d}}{|\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

q_i is the tf-idf weight of term *i* in the query
d_i is the tf-idf weight of term *i* in the document

cos(*q*,*d*) is the cosine similarity of *q* and *d* … or, equivalently, the cosine of the angle between *q* and *d*.

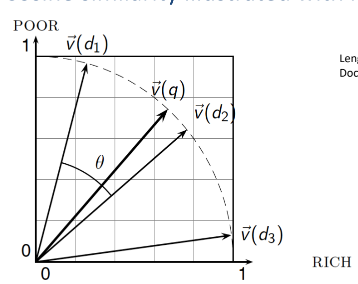
Cosine for length normalized vectors

- For length-normalized vectors, cosine similarity is simply the dot product (or scalar product)

$$\cos(\vec{q},\vec{d}) = \vec{q}\bullet\vec{d} = \sum_{i=1}^{|V|} q_i d_i$$

- For q, d length normalized
- How why do you length-normalize a vector?
- A vector can be (length-) normalized by dividing each of its components by its length – for this we use the L_2 norm
- When you length-normalize, the vector length becomes unit or 1. $\|\vec{x}\|_2 = \sqrt{\sum_i x_i^2}$

Cosine similarity illustrated with length-normalized



Length-normalized query and Documents!

Cosine similarity for 3 documents

- How similar are the novels
  - SaS: *Sense and Sensibility*
  - PaP: *Pride and Prejudice*, and
  - WH: *Wuthering Heights*?

term	SaS	PaP	WH
affection	115	58	20
jealous	10	7	11
gossip	2	0	6
wuthering	0	0	38

Term frequencies (counts)

– For simplicity, we do not use idf weighting!

Cosine similarity for 3 documents (contd.)

Log frequency weighting

term	SaS	PaP	WH
affection	3.06	2.76	2.30
jealous	2.00	1.85	2.04
gossip	1.30	0	1.78
wuthering	0	0	2.58

After length normalization

term	SaS	PaP	WH
affection	0.789	0.832	0.524
jealous	0.515	0.555	0.465
gossip	0.335	0	0.405
wuthering	0	0	0.588

cos(SaS,PaP) ≈ 0.789 × 0.832 + 0.515 × 0.555 + 0.335 × 0.0 + 0.0 × 0.0 ≈ 0.94
cos(SaS,WH) ≈ 0.789x0.524 + 0.515 x 0.465 + 0.335 x 0.405 = 0.79
cos(PaP,WH) ≈ 0.69

Why do we have cos(SaS,PaP) > cos(SaS,WH)?

Summary – vector space ranking

- Represent the query as a weighted tf-idf vector
- Represent each document as a weighted tf-idf vector
- Compute the cosine similarity score for the query vector and each document vector
- Rank documents with respect to the query by score
- Return the top K (e.g., K = 10) to the user
- Efficient computation is a different ball game!
- Note similarity ranking involves computing millions of values for each query! How to speed it up?
  – Pre-computation is a key component!

Resources

http://www.miislita.com/information-retrieval-tutorial/cosine-similarity-tutorial.html

Thank You !!!

For more information visit:
http://itlab.uta.edu

13 December 2018 52 2018 (Warangal)