Semántica, Datos Conectados y Minería de Datos Textual

Minería de textos y minería Web

Cristina Tîrnăucă

Dept. Matesco, Universidad de Cantabria

Facultad de Ciencias – Máster en Data Science

Web Mining

Definition

Web mining is the process of finding patterns from the web.
It involves analyzing the data or extracting information about a piece of data from the web.

Types

- Web usage mining
  The process of extracting user information from server logs
- Web structure mining
  The process of using graph theory to analyse the structure of a website
  - Extracting patterns
  - Mining the structure of the document
- Web content mining
  The process of mining, extraction and integration of useful data, information and knowledge from web page content.

Text Mining

Definition

Text mining is the process of deriving high-quality information from text.

Text analysis involves

- information retrieval (to find relevant documents),
- lexical analysis (to study word frequency distributions),
- syntactic parsing
- pattern recognition,
- tagging/annotation,
- information extraction,
- data mining techniques (including link and association analysis, visualization, and predictive analytics).

Text Mining Tasks

- text categorization (spam detection, authorship identification, age/gender identification, assigning subject categories, topics, or genres),
- sentiment analysis (detection of attitudes, holder, target, type)
- text clustering (news),
- concept/entity extraction (also called named entity extraction),
- document summarization,
- entity relation modeling (i.e., learning relations between named entities) - useful in question answering.

Information Retrieval

Information Retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).
These days we frequently think first of web search, but there are many other cases:

- E-mail search
- Searching your laptop
- Corporate knowledge bases
- Legal information retrieval

Basic Assumptions of Information Retrieval

Collection: A set of documents
Assume it is a static collection for the moment
Goal: Retrieve documents with information that is relevant to the user's information need and helps the user complete a task
Evaluation: How good are the retrieved docs?

- Precision: Fraction of retrieved docs that are relevant to the user's information need
- Recall : Fraction of relevant docs in collection that are retrieved

Information Retrieval, an Example

- Which plays of Shakespeare contain the words Brutus AND Caesar but NOT Calpurnia?
- One could grep all of Shakespeare's plays for Brutus and Caesar, then strip out lines containing Calpurnia?
- Why is that not the answer?
  - Slow (for large corpora)
  - NOT Calpurnia is non-trivial
  - Other operations (e.g., find the word Romans near countrymen) not feasible
  - Ranked retrieval (best documents to return)

Term-document Incidence Matrices

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othelo	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

So we have a 0/1 vector for each term.
To answer query: take the vectors for Brutus, Caesar and Calpurnia (complemented) $\rightarrow$ bitwise AND.
110100 AND
110111 AND
101111 =
100100

Term-document Incidence Matrices for Bigger Collections

- Consider N = 1 million documents, each with about 1000 words.
- Avg 6 bytes/word including spaces/punctuation
  6GB of data in the documents.
- Say there are M = 500K distinct terms among these.
- Can't build the matrix!
  - 500K x 1M matrix has half-a-trillion 0's and 1's.
  - But it has no more than one billion 1's.
    matrix is extremely sparse.
  - What's a better representation?
    We only record the 1 positions.

Inverted index

For each term t, we must store a list of all documents that contain t. Identify each doc by a docID, a document serial number.
Can we used fixed-size arrays for this?

Brutus	⇒	1	2	4	11	31	45	173	174
Caesar	⇒	1	2	4	5	6	16	57	132
Calpurnia	⇒	2	31	54	101				

What happens if the word Caesar is added to document 14?

Inverted Index

We need variable-size postings lists

- On disk, a continuous run of postings is normal and best
- In memory, can use linked lists or variable length arrays
  Some trade-offs in size/ease of insertion

Data is organized into:

- Dictionary of terms
- Postings (sorted by DocID)

Initial stages of text processing

- Tokenization
  Cut character sequence into word tokens
- Normalization
  Map text and query term to same form
  You want U.S.A. and USA to match
- Stemming
  We may wish different forms of a root to match
  authorize, authorization
- Stop words
  We may omit very common words (or not)
  the, a, to, of

Inverted Index

Query processing: AND

- Locate both words in the Dictionary
- Retrieve their postings
- "Merge" the two postings (intersect the document sets):
  - If postings are sorted by docID, it can be done in time $\mathcal{O}(len1 + len2)$

Inverted Index

Query processing: phrase queries

- It no longer suffices to store only term:docs entries
- A first solution: biword indexes
  - Longer phrases can be processed by breaking them down (can lead to false positives)
  - Index blowup due to bigger dictionary
  - Not the standard solution, but can be used as part of a compound strategy
- A second solution: positional indexes
  - In the postings, store, for each term the position(s) in which tokens of it appear
  - For phrase queries, we use a merge algorithm recursively at the document level
  - But we now need to deal with more than just equality

Ranked Retrieval

Why do we need more than boolean queries?

- Thus far, our queries have all been Boolean.
  Documents either match or don't.
- Good for expert users with precise understanding of their needs and the collection.
  Also good for applications: Applications can easily consume 1000s of results.
- Not good for the majority of users.
  - Most users incapable of writing Boolean queries (or they are, but they think it's too much work).
  - Most users don't want to wade through 1000s of results.
  - This is particularly true of web search.
- Boolean queries often result in either too few (0) or too many (1000s) results.
- 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

Ranked Retrieval Models

- Rather than a set of documents satisfying a query expression, in ranked retrieval models, the system returns an ordering over the (top) documents in the collection with respect to 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
- When a system produces a ranked result set, large result sets are not an issue

Scoring as the basis of ranked retrieval

- We wish to return in order the documents most likely to be useful to the searcher
- We need a way of assigning a score to a query/document pair
  Simplified scenario: if the query has only one term
  - If the query term does not occur in the document: score should be 0
  - The more frequent the query term in the document, the higher the score should be

Jaccard coefficient

A commonly used measure of overlap of two sets A and B is the Jaccard coefficient

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

Issues with Jaccard for scoring:

- It doesn't consider term frequency (how many times a term occurs in a document)
  Rare terms in a collection are more informative than frequent terms
- We need a more sophisticated way of normalizing for length

An Example

Given the following four documents:

d1: I bought one computer and one laptop for a great price at one new retail sellers.

d2: For the price of this laptop, you could have bought one desktop computer for home and another one for your job.

d3: Computer repair budget: 200 euros (a 55 euros discount is offered to registered customers).

d4: I need the budget of the laptop and the budget of the printer, in dollars and in euros.

Let us consider a subset of the vocabulary: computer, euros, budget, laptop, price, one, in.

Our search: computer price

Binary Term-document Incidence Matrix

	d1	d2	d3	d4
computer	1	1	1	0
euros	0	0	1	1
budget	0	0	1	1
laptop	1	1	0	1
price	1	1	0	0
one	1	1	0	0
in	0	0	0	1

Term-document Count Matrices

Table: Term frequency $tf_{t,d}$

	d1	d2	d3	d4
computer	1	1	1	0
euros	0	0	2	1
budget	0	0	1	2
laptop	1	1	0	1
price	1	1	0	0
one	3	1	0	0
in	0	0	0	2

$tf_{t,d}$ = how many times the term t appears in document d
Each document is a vector in $\mathbb{N}^{|V|}$

Logarithmic Term Frequency

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

Table: Log-frequency weight, $w_{t,d}$

	d1	d2	d3	d4
computer	1	1	1	0
euros	0	0	1.301	1
budget	0	0	1	1.301
laptop	1	1	0	1
price	1	1	0	0
one	1.477	1	0	0
in	0	0	0	1.301

Score for a document-query pair: sum over terms t in q and d:

$$\sum_{t \in q \cap d} 1 + \log_{10} tf_{t,d}$$

Document Frequency

df_t = in how many documents the word t appears

Table: Document frequency, df_t

Vocabulary		Posting list			df_t	idf_t
computer	$\rightarrow$	d1,	d2,	d3	3	0.125
euros	$\rightarrow$	d3,	d4		2	0.301
budget	$\rightarrow$	d3,	d4		2	0.301
laptop	$\rightarrow$	d1,	d2,	d4	3	0.125
price	$\rightarrow$	d1,	d2		2	0.301
one	$\rightarrow$	d1,	d2		2	0.301
in	$\rightarrow$	d4			1	0.602

- the base of the log is immaterial
- idf has no effect on ranking one term queries.

Inverse document frequency:
$idf_t = \log_{10}(N/df_t)$

Term Frequency, Inverse Document Frequency Weighting

$\text{tf-idf}_{t,d} = (1 + \log_{10} tf_{t,d}) * \log_{10}(N/df_t)$

Table: Term Frequency, Inverse Document Frequency Weighting: $\text{tf-idf}_{t,d}$

	d1	d2	d3	d4
computer	1 * 0.125	1*0.125	1*0.125	0
euros	0	0	1.301*0.301	1*0.301
budget	0	0	1*0.301	1.301*0.301
laptop	1*0.125	1*0.125	0	1*0.125
price	1*0.301	1*0.301	0	0
one	1.477*0.301	1*0.301	0	0
in	0	0	0	1.301*0.602

Final score:

$$\sum_{t \in q \cap d} (1 + \log_{10} tf_{t,d}) * \log_{10}(N/df_t)$$

Each document is now represented by a real-valued vector of tf-idf weights.

The Vector Space Model

Documents as vectors

- Now 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 space
- Key idea 2: Rank documents according to their proximity to the query in this space
- proximity = similarity of vectors
- 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

- Using Euclidean distance is a bad idea, because Euclidean distance is large for vectors of different lengths
- Using angle instead of distance is much better:
  - 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.

The Cosine Similarity

From angles to cosine

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

Length normalization

- A vector can be (length-) normalized by dividing each of its components by its length - for this we use the L_2 norm: $| \vec{x} |_2 = \sqrt{\sum_i x_i^2}$
- Dividing a vector by its L2 norm makes it a unit (length) vector (on surface of unit hypersphere)
- Effect on the two documents d and d' (d appended to itself): they have identical vectors after length-normalization.
  Long and short documents now have comparable weights.

Cosine Similarity

$$cos(q,d) = \frac{\sum_{t \in q \cap d} q_t d_t}{\sqrt{\sum_{t \in V} q_t^2} * \sqrt{\sum_{t \in V} d_t^2}}$$

q_t is the tf-idf weight of term t in the query
d_t is the tf-idf weight of term t in the document

Variants of tf-idf

Term frequency	Document frequency	Normalization
n (natural): $tf_{t,d}$	n (no): 1	n (none): 1
l (logarithm): $1 + \log_{10}(tf_{t,d})$	t (idf): $\log(\frac{N}{df_t})$	c (cosine): $\frac{1}{\sqrt{\sum_i x_i^2}}$
a (augmented): $0.5 + \frac{0.5 * tf_{t,d}}{\max_t(tf_{t,d})}$	p (prob idf): $\max\{0, \log\frac{N - df_t}{df_t}\}$	u (pivoted unique): $1/u$
b (boolean) $\begin{cases} 1 & \text{if } tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$		b (byte size): $1/CharLength^{\alpha}$
L (log ave): $\frac{1+\log_{10}(tf_{t,d})}{1+\log_{10}(ave_{t \in d}(tf_{t,d}))}$		$(\alpha < 1)$

Weighting Schemes

- Weighting may differ in queries vs documents
- Many search engines allow for different weightings for queries vs. documents
- SMART Notation: denotes the combination in use in an engine, with the notation ddd.qqq, using the acronyms from the previous table
- A very standard weighting scheme is: lnc.ltc

$$cos(q,d) = \frac{\sum_{t \in q \cap d} q_t d_t}{\sqrt{\sum_{t \in V} q_t^2} * \sqrt{\sum_{t \in V} d_t^2}}$$

$$q_t = (1 + \log_{10} tf_{t,q}) * \log_{10}(N/df_t)$$

$$d_t = (1 + \log_{10} tf_{t,d}) * \log_{10}(N/df_t)$$

Evaluating an IR System

- An information need is translated into a query
- Relevance is assessed relative to the information need not the query
- E.g., Information need: *I'm looking for information on whether drinking red wine is more effective at reducing your risk of heart attacks than white wine.*
- Query: *wine red white heart attack effective*
- You evaluate whether the doc addresses the information need, not whether it has these words

Two Current Evaluation Measure

Precision-recall curve

(Assume 10 relevant documents in collection)

		Recall	Precision
1	R	0.1	1.0
2	N	0.1	0.5
3	N	0.1	0.33
4	R	0.2	0.5
5	R	0.3	0.6
6	N	0.3	0.5
7	R	0.4	0.57
8	N	0.4	0.5

Mean average precision (MAP)

Average of the precision value obtained for the top k documents, each time a relevant document is retrieved.

$MAP = (1/1 + 2/4 + 3/5 + 4/7)/4 = 0.67$