Lecture 11

Artificial Intelligence: Natural Language Processing (NLP)

NLP Applications, Vector Space Models

COMP 6721, Winter 2021

Outline

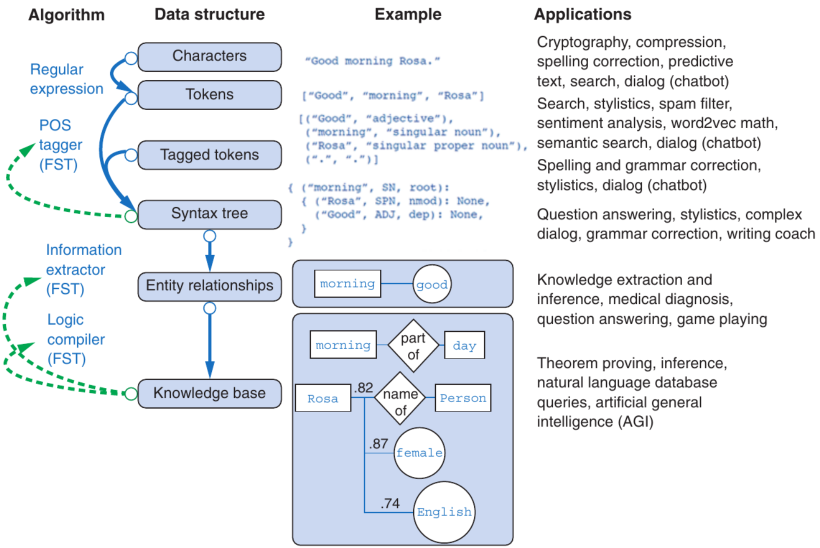
Slides Credit

Includes slides by Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze [MRS08]

NLP Applications

Search	Web	Documents	Autocomplete
Editing	Spelling	Grammar	Style
Dialog	Chatbot	Assistant	Scheduling
Writing	Index	Concordance	Table of contents
Email	Spam filter	Classification	Prioritization
Text mining	Summarization	Knowledge extraction	Medical diagnoses
Law	Legal inference	Precedent search	Subpoena classification
News	Event detection	Fact checking	Headline composition
Attribution	Plagiarism detection	Literary forensics	Style coaching
Sentiment analysis	Community morale monitoring	Product review triage	Customer care
Behavior prediction	Finance	Election forecasting	Marketing
Creative writing	Movie scripts	Poetry	Song lyrics

Example NLP Pipeline



So you want to build an NLP application. . .

Requirements

An NLP system requires a large amount of infrastructure work:

- Document handling, in various formats (plain text, HTML, XML, PDF, . . .), from various sources (files, DBs, email, . . .)
- Annotation handling (stand-off markup)
- Component implementations for standard tasks, like Tokenizers, Sentence Splitters, Part-of-Speech (POS) Taggers, Finite-State Transducers, Full Parsers, Classifiers, Noun Phrase Chunkers, Lemmatizers, Entity Taggers, Coreference Resolution Engines, Summarizers, . . .

As well as *resources* for concrete tasks and languages:

- Lexicons, WordNets
- Grammar files and Language models
- Machine Learning Algorithms & Evaluation Metrics, etc.

Existing Resources

Fortunately, you don't have to start from scratch

Many (open source) tools and resources are available:

NLP Tools: programs performing a single task, like classifiers, parsers, or NP chunkers

Frameworks: integration architectures for combining and controlling all components and resources of an NLP system

Resources: for various languages, like lexicons, wordnets, or grammars

NLP Development

Major Frameworks

Two important frameworks are:

- GATE *(General Architecture for Text Engineering)*, under development since 1995 at University of Sheffield, UK
- UIMA *(Unstructured Information Management Architecture)*, developed by IBM; open-sourced in 2007 (Apache project)

Both frameworks are open source (GATE: LGPL, UIMA: Apache)

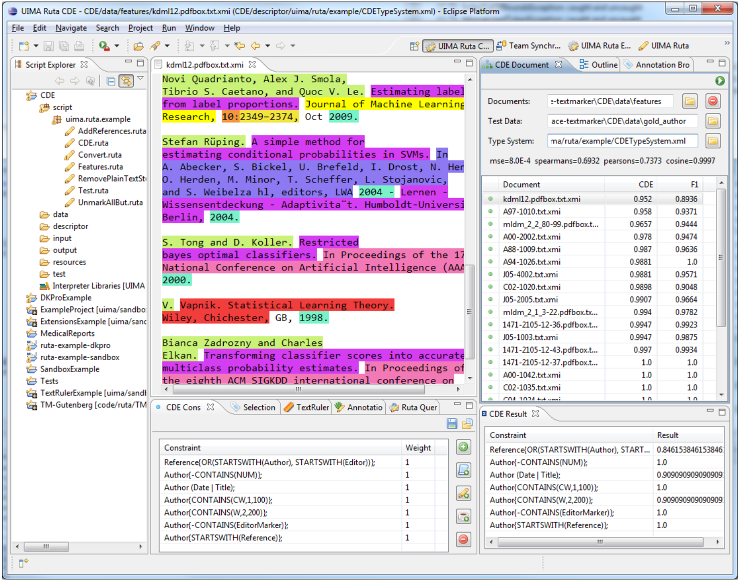
Libraries

- Numerous NLP libraries: NLTK (Python), Stanford CoreNLP, ...
- Various integrations (e.g, CoreNLP has GATE wrapper, Python bindings)

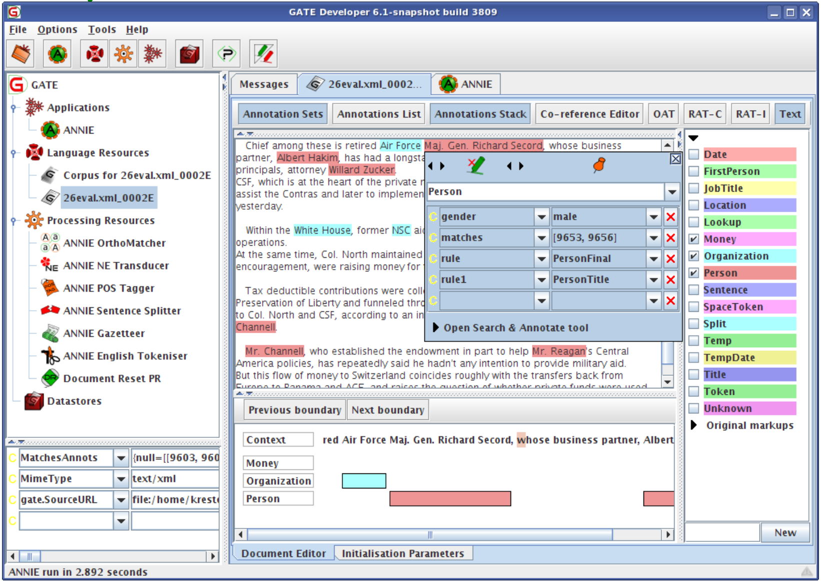
Current Trends

- Increasing use of Deep Learning tools/frameworks for NLP
- Keras, TensorFlow, PyTorch etc.

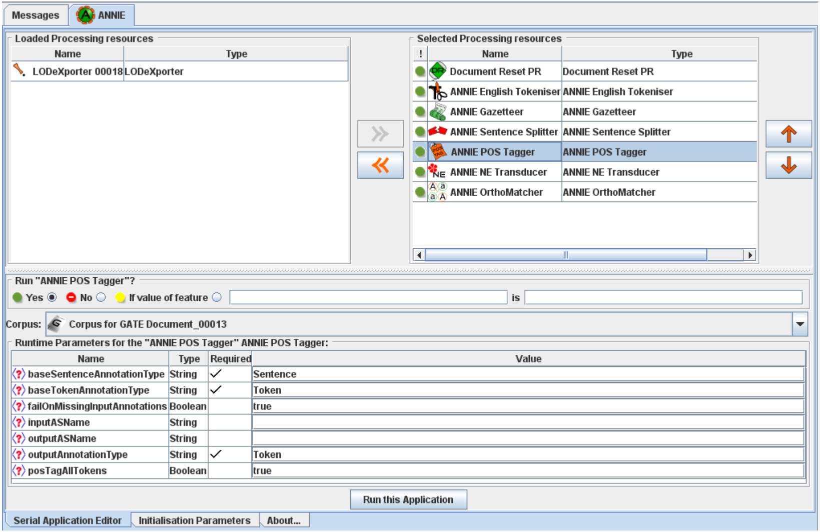
Unstructured Information Management Architecture (UIMA)



General Architecture for Text Engineering (GATE)



NLP Pipeline in GATE



Pipeline Step: Tokenization

Example Tokenisation Rules

```
#numbers#
// a number is any combination of digits
"DECIMAL_DIGIT_NUMBER"+ >Token;kind=number;

#whitespace#
(SPACE_SEPARATOR) >SpaceToken;kind=space;
(CONTROL) >SpaceToken;kind=control;
```

Example Output

Type	Set	Start	End	Features
Token		158	163	{kind=word, length=5, orth=lowercase, string=years}
SpaceToken		163	164	{kind=space, length=1, string= }
Token		164	167	{kind=word, length=3, orth=lowercase, string=ago}
Token		167	168	{kind=punctuation, length=1, string=,}
SpaceToken		168	169	{kind=space, length=1, string= }
Token		169	180	{kind=word, length=11, orth=lowercase, string=researchers}
SpaceToken		180	181	{kind=space, length=1, string= }

1417 Annotations (0 selected)

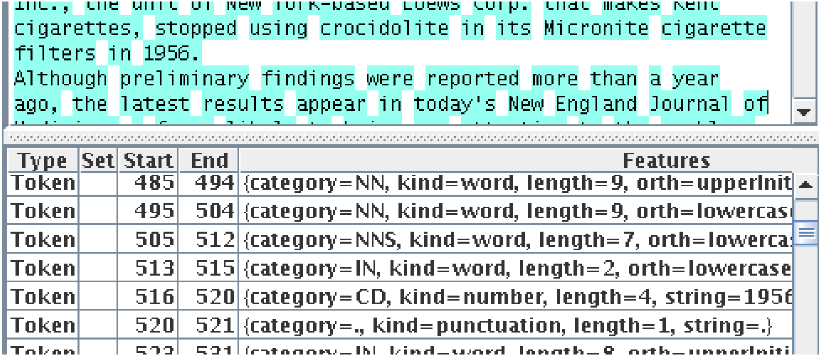
Pipeline Step: POS Tagging

Producing POS Annotations

POS-Tagging assigns a part-of-speech-tag (POS tag) to each `Token`.

- GATE comes with the Hepple tagger for English, which is a modified version of the Brill tagger

Example output


cigarettes, stopped using crocidolite in its Micronite cigarette
filters in 1956.
Although preliminary findings were reported more than a year
ago, the latest results appear in today's New England Journal of

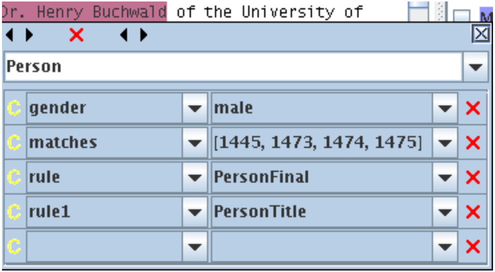
Type	Set	Start	End	Features
Token		485	494	{category=NN, kind=word, length=9, orth=upperInit
Token		495	504	{category=NN, kind=word, length=9, orth=lowercas
Token		505	512	{category=NNS, kind=word, length=7, orth=lowerca
Token		513	515	{category=IN, kind=word, length=2, orth=lowercase
Token		516	520	{category=CD, kind=number, length=4, string=1956
Token		520	521	{category=., kind=punctuation, length=1, string=.}

Pipeline Step: Named Entity (NE) Detection

Transducer-based NE Detection

Using all the information obtained in the previous steps (Tokens, Gazetteer lookups, POS tags), ANNIE now runs a sequence of JAPE-Transducers to detect Named Entities (NE)s.

Example for a detected *Person*



We can now look at the grammar rules that found this person.

Entity Detection: Finding Persons

Strategy

A JAPE grammar rule combines information obtained from POS-tags with Gazetteer lookup information

- although the last name in the example is not in any list, it can be found based on its POS tag and an additional first name/last name rule (not shown)
- many additional rules for other Person patterns, as well as Organizations, Dates, Addresses, ...

Persons with Titles

```
Rule:   PersonTitle
Priority: 35
(
 {Token.category == DT}|
 {Token.category == PRP}|
 {Token.category == RB}
)?
(
 (TITLE)+
 ((FIRSTNAME | FIRSTNAMEAMBIG
   | INITIALS2)
 )?
  (PREFIX)*
  (UPPER)
 (PERSONENDING)?
)
:person --> ...
```

→ **Worksheet #10: "Information Extraction"**

1. NLP Applications

2. Processing & Vectorization
   - Preprocessing and Tokenisation
   - Morphology
   - Bag-of-Words (BOW) Model
   - One-Hot Vectors
   - Computing with Words

3. Document Vector Space Model

4. Notes and Further Reading

Tokenization

Preprocessing

Input files usually need some cleanup before processing can start:

- Remove "fluff" from web pages (ads, navigation bars, ...)
- Normalize text converted from PDF, Doc, or other binary formats
- Deal with errors in OCR'd documents
- Deal with tables, figures, captions, formulas, ...

Tokenization

Text is split into basic units called *Tokens*:

- word tokens
- number tokens
- space tokens
- ...

Consistent tokenization is important for all later processing steps

Tokenization (II)

What is a word?

Unfortunately, even tokenization can be difficult:

- Is "John's" in *John's sick* one token or two?
  If one → problems in parsing (where's the verb?)
  If two → what do we do with *John's house?*
- What to do with hyphens?
  E.g., *database* vs. *data-base* vs. *data base*
- what to do with "C++", "A/C", "`:-)`", "`...`"?

Even worse...

- Some languages don't use whitespace (e.g., Chinese)
  → need to run a *word segmentation* first
- Heavy compounding e.g. in German, decomposition necessary
  "Rinderbraten" → *Rinder|braten?* (roast beef)
  Rind|erb|raten? (cattle inheritance rate)
  Rind|erbraten? (generate cattle through BBQ'ing)

Tokenization (III)

The good, the bad, and the ...

Tokenization can become even more difficult in specific domains.

Software Documents

Documents include lots of source code snippets:

- package `java.util.*`
- *The range-view operation, subList(int fromIndex, int toIndex), returns a List view of the portion of this list whose indices range from fromIndex, inclusive, to toIndex , exclusive.*

Need to deal with URLs, methods, class names, etc.

Tokenization (IV)

Biological/Chemical Documents

Highly complex expressions, chemical formulas, etc.:

- *1,4-β-xylanase II from Trichoderma reesei*
- *When N-formyl-L-methionyl-L-leucyl-L-phenylalanine (fMLP) was injected...*
- *Technetium-99m-CDO-MeB [Bis[1,2-cyclohexanedione-dioximato(1-)-O]-[1,2-cyclohexanedione dioximato(2-)-O] methyl-borato(2-)-N,N′,N″,N‴,N⁗,N′′′′′)-chlorotechnetium) belongs to a family of compounds...*

Morphological Analysis

Morphological Variants

Words are changed through a morphological process called *inflection*:

- typically indicates changes in case, gender, number, tense, etc.
- example *car* → cars, *give* → *gives, gave, given*

Goal: "normalize" words

Stemming and Lemmatization

Two main approaches to normalization:

Stemming
: reduce words to a *base form*

Lemmatization
: reduce words to their *lemma*

Main difference: stemming just finds **any** base form, which doesn't even need to be a word in the language! Lemmatization finds the actual *root* of a word, but requires morphological analysis.

Stemming vs. Lemmatization

Stemming

Commonly used in Information Retrieval:

- Can be achieved with rule-based algorithms, usually based on suffix-stripping
- Standard algorithm for English: the *Porter* stemmer
- Advantages: simple & fast
- Disadvantages:
    - Rules are language-dependent
    - Can create words (stems) that do not exist in the language, e.g.,
      computers → *comput*
    - Often reduces different words to the same stem, e.g.,
      army, arm → *arm*
      stocks, stockings → *stock*

Stemming vs. Lemmatization, Part II

Lemmatization

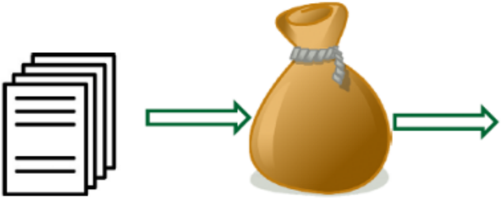
Lemmatization is the process of deriving the base form, or *lemma*, of a word from one of its inflected forms. This requires a morphological analysis, which in turn typically requires a *lexicon*.

- Advantages:
    - identifies the *lemma* (root form), which is an actual word
    - less errors than in stemming
- Disadvantages:
    - more complex than stemming, slower
    - requires additional language-dependent resources

Bag-of-Words (BOW) Model

Task

Turn words into numbers.



Word	Freq.
Mary	2
apples	1
did	2
eat	1
John	1
kill	1
like	1
not	1
to	1

Problems with the Bag-of-Words Model

Mary did kill John.
Mary did not like to eat apples.

John did not kill Mary.
Mary did like to eat apples.

Mary did not like to kill John.
Mary did eat apples.

...

Word	Freq.
Mary	2
apples	1
did	2
eat	1
John	1
kill	1
like	1
not	1
to	1

Word order is ignored

Meaning of the text is lost.

One-Hot Vectors

Vector dimensionality = Vocabulary size

With n-dimensional vectors of $\{0, 1\}$, we can represent each word in our vocabulary that has 1 (one) for the word, else 0 (zero).

Example

We can encode the sentence The big dog as a series of three-dimensional vectors:

the	big	dog
1	0	0
0	1	0
0	0	1

(a "1" means on, or hot; a "0" means off, or absent.)

Note

- Unlike in the BOW model, we do not lose information
- Not practical for long documents

Sentence Vectors

Simplification

Make “sentence vectors”, ignoring the order within a sentence:

```
sentences = """the big dog
the big cat
the big cat and the dog"""
corpus = {}

for i, sent in enumerate(sentences.split('\n')):
    corpus['sent{}'.format(i)] = dict((tok, 1) for tok in
         sent.split())
df = pd.DataFrame.from_records(corpus).fillna(0).astype(int).T
```

	and	big	cat	dog	the
sent0	0	1	0	1	1
sent1	0	1	1	0	1
sent2	1	1	1	1	1

Dot product

Dot product of two n-dimensional vectors

$$\vec{v} \cdot \vec{w} = \sum_{i=1}^{n} v_i \cdot w_i$$

also known as the scalar product or inner product

Note

Do not confuse with the cross product ("xyzzy"), written $\vec{v} \times \vec{w}$

In Python

```
>>> v1 = pd.np.array([1, 2, 3])
>>> v2 = pd.np.array([2, 3, 4])
>>> v1.dot(v2)
20
```

→ Worksheet #10: "Vector dot product"

Sentence similarity

Compute vector overlap

Computing the dot product of two sentence vectors in this encoding tells us how many words they have in common.

Example

We could use this to:

- answer questions by looking at sentence overlap
- summarize documents by removing redundant sentences

This is a first example of a vector space model (VSM)

Binary incidence matrix

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Anthony	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 as a binary vector $\in \{0,1\}^{|V|}$.

[from *Introduction to Information Retrieval*]

Count matrix

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Anthony	157	73	0	0	0	1	
Brutus	4	157	0	2	0	0	
Caesar	232	227	0	2	1	0	
Calpurnia	0	10	0	0	0	0	
Cleopatra	57	0	0	0	0	0	
mercy	2	0	3	8	5	8	
worser	2	0	1	1	1	5	
...							

Each document is now represented as a count vector $\in \mathbb{N}^{|V|}$.

Bag of words model

- We do not consider the order of words in a document.
- *John is quicker than Mary* and *Mary is quicker than John* are represented the same way.
- This is called a bag of words model.

Term frequency tf

The term frequency $\mathrm{tf}_{t,d}$ of term t in document d is defined as the number of times that t occurs in d.

$\rightarrow$ **Worksheet #10: "Term Frequency"**

Frequency in document vs. frequency in collection

- In addition, to term frequency (the frequency of the term in the document) ...
- ...we also want to use the frequency of the term in the collection for weighting and ranking.

Desired weight for rare terms

- Rare terms are more informative than frequent terms.
- Consider a term in the query that is rare in the collection (e.g., arachnocentric).
- A document containing this term is very likely to be relevant.
- $\rightarrow$ We want high weights for rare terms like arachnocentric.

Desired weight for frequent terms

- Frequent terms are less informative than rare terms.
- Consider a term in the query that is frequent in the collection (e.g., good, increase, line).
- A document containing this term is more likely to be relevant than a document that doesn't ...
- ...but words like good, increase and line are not sure indicators of relevance.
- $\rightarrow$ For frequent terms like good, increase, and line, we want positive weights ...
- ...but lower weights than for rare terms.

Document frequency

- We want high weights for rare terms like arachnocentric.
- We want low (positive) weights for frequent words like good, increase, and line.
- We will use document frequency to factor this into computing the matching score.
- The document frequency is the number of documents in the collection that the term occurs in.

idf weight

- df_t is the document frequency, the number of documents that t occurs in.
- df_t is an inverse measure of the informativeness of term t.
- We define the idf weight of term t as follows:

$$\mathrm{idf}_t = \log_{10} \frac{N}{\mathrm{df}_t}$$

  (N is the number of documents in the collection.)
- idf_t is a measure of the informativeness of the term.
- $[\log N/\mathrm{df}_t]$ instead of $[N/\mathrm{df}_t]$ to "dampen" the effect of idf
- Note that we use the log transformation for both term frequency and document frequency.

→ Worksheet #10: "Inverse Document Frequency"

Examples for idf

Compute idf_t using the formula: $\mathrm{idf}_t = \log_{10} \frac{1{,}000{,}000}{\mathrm{df}_t}$

term	df_t	idf_t
calpurnia	1	6
animal	100	4
sunday	1000	3
fly	10,000	2
under	100,000	1
the	1,000,000	0

Effect of idf on ranking

- idf affects the ranking of documents for queries with at least two terms.
- For example, in the query "arachnocentric line", idf weighting increases the relative weight of arachnocentric and decreases the relative weight of line.
- idf has little effect on ranking for one-term queries.

Collection frequency vs. Document frequency

word	collection frequency	document frequency
insurance	10440	3997
try	10422	8760

- Collection frequency of t: number of tokens of t in the collection
- Document frequency of t: number of documents t occurs in
- Why these numbers?
- Which word is a better search term (and should get a higher weight)?
- This example suggests that df (and idf) is better for weighting than cf (and "icf").

tf-idf weighting

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

$$w_{t,d} = (1 + \log \mathrm{tf}_{t,d}) \cdot \log \frac{N}{\mathrm{df}_t}$$

- Set to 0 if $\mathrm{tf}_{t,d} = 0$
- Best known weighting scheme in information retrieval
- Note: the "-" in tf-idf is a hyphen, not a minus sign!
- Alternative names: tf.idf, tf$\times$idf
- Note: there are lots of variations/alternative weighting schemes

→ Worksheet #10: "tf-idf Weights"

Summary: tf-idf

- Assign a tf-idf weight for each term t in each document d:

$$w_{t,d} = \begin{cases} (1 + \log \mathrm{tf}_{t,d}) \cdot \log \frac{N}{\mathrm{df}_t}, & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- The tf-idf weight ...
  - ...increases with the number of occurrences within a document. (term frequency)
  - ...increases with the rarity of the term in the collection. (inverse document frequency)

Binary incidence matrix

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Anthony	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 as a binary vector $\in \{0,1\}^{|V|}$.

[from *Introduction to Information Retrieval*]

Count matrix

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Anthony	157	73	0	0	0	1	
Brutus	4	157	0	2	0	0	
Caesar	232	227	0	2	1	0	
Calpurnia	0	10	0	0	0	0	
Cleopatra	57	0	0	0	0	0	
mercy	2	0	3	8	5	8	
worser	2	0	1	1	1	5	
...							

Each document is now represented as a count vector $\in \mathbb{N}^{|V|}$.

Binary $\rightarrow$ count $\rightarrow$ weight matrix

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Anthony	5.25	3.18	0.0	0.0	0.0	0.35	
Brutus	1.21	6.10	0.0	1.0	0.0	0.0	
Caesar	8.59	2.54	0.0	1.51	0.25	0.0	
Calpurnia	0.0	1.54	0.0	0.0	0.0	0.0	
Cleopatra	2.85	0.0	0.0	0.0	0.0	0.0	
mercy	1.51	0.0	1.90	0.12	5.25	0.88	
worser	1.37	0.0	0.11	4.15	0.25	1.95	
...							

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

Documents as vectors

- Each document is now represented as a real-valued vector of tf-idf weights $\in \mathbb{R}^{|V|}$.
- So we have a $|V|$-dimensional real-valued 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 web search engines
- Each vector is very sparse - most entries are zero.

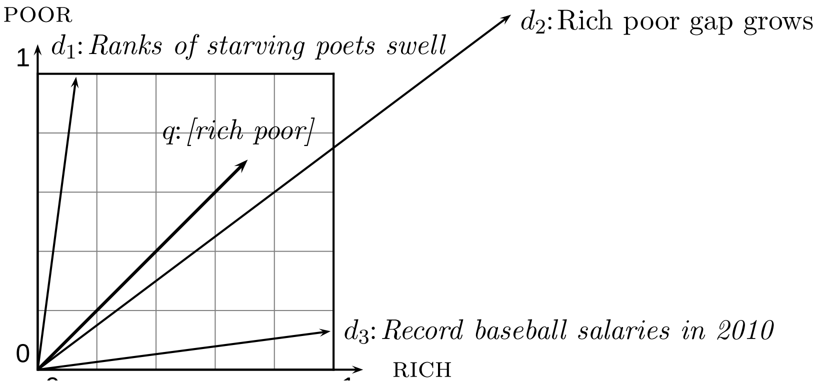
Queries as vectors

- Key idea 1: do the same for queries: represent them as vectors in the high-dimensional space
- Key idea 2: Rank documents according to their proximity to the query
- proximity = similarity
- proximity $\approx$ negative distance
- Recall: We're doing this because we want to get away from the you're-either-in-or-out, feast-or-famine Boolean model.
- Instead: rank relevant documents higher than nonrelevant documents

How do we formalize vector space similarity?

- First cut: (negative) distance between two points
- (= distance between the end points of the two vectors)
- Euclidean distance?
- Euclidean distance is a bad idea . . .
- . . . because Euclidean distance is large for vectors of different lengths.

Why distance is a bad idea



The Euclidean distance of $\vec{q}$ and $\vec{d}_2$ is large although the distribution of terms in the query q and the distribution of terms in the document d_2 are very similar.

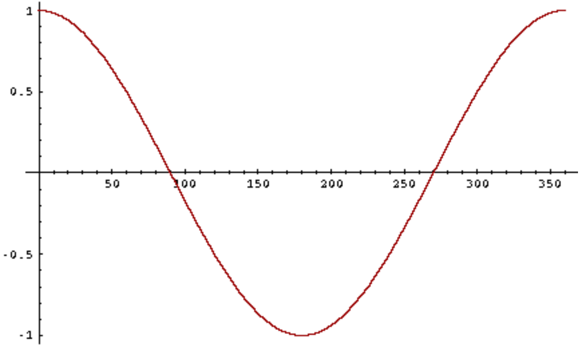
Use angle instead of distance

- Rank documents according to angle with query
- Thought experiment: take a document d and append it to itself. Call this document d'. d' is twice as long as d.
- “Semantically” d and d' have the same content.
- The angle between the two documents is 0, corresponding to maximal similarity …
- …even though the Euclidean distance between the two documents can be quite large.

From angles to cosines

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

Cosine



Length normalization

- How do we compute the cosine?
- A vector can be (length-) normalized by dividing each of its components by its length – here we use the L_2 norm: $||x||_2 = \sqrt{\sum_i x_i^2}$
- This maps vectors onto the unit sphere ...
- ...since after normalization: $||x||_2 = \sqrt{\sum_i x_i^2} = 1.0$
- As a result, longer documents and shorter documents have weights of the same order of magnitude.
- Effect on the two documents d and d' (d appended to itself) from earlier slide: they have identical vectors after length-normalization.

Cosine similarity between query and document

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

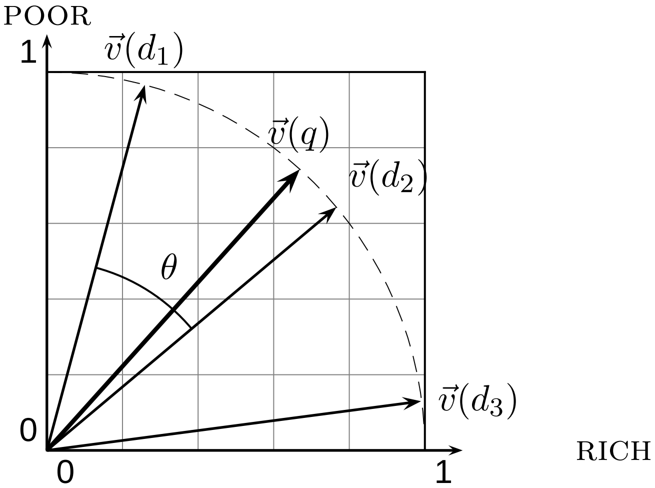
- q_i is the tf-idf weight of term i in the query.
- d_i is the tf-idf weight of term i in the document.
- $|\vec{q}|$ and $|\vec{d}|$ are the lengths of $\vec{q}$ and $\vec{d}$.
- This is the cosine similarity of $\vec{q}$ and $\vec{d}$ ...... or, equivalently, the cosine of the angle between $\vec{q}$ and $\vec{d}$.

Cosine for normalized vectors

- For normalized vectors, the cosine is equivalent to the dot product or scalar product.
- $\cos(\vec{q}, \vec{d}) = \vec{q} \cdot \vec{d} = \sum_i q_i \cdot d_i$
  - (if $\vec{q}$ and $\vec{d}$ are length-normalized).

$\rightarrow$ **Worksheet #10: "Cosine Similarity"**

Cosine similarity illustrated



Cosine: Example

How similar are these novels?

SaS: Sense and Sensibility

PaP: Pride and Prejudice

WH: Wuthering Heights

term frequencies (counts)

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

Cosine: Example

term frequencies (counts)

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

log frequency weighting

term	SaS	PaP	WH
affection	3.06	2.76	2.30
jealous	2.0	1.85	2.04
gossip	1.30	0	1.78
wuthering	0	0	2.58

(To simplify this example, we don't do idf weighting.)

Cosine: Example

log frequency weighting

term	SaS	PaP	WH
affection	3.06	2.76	2.30
jealous	2.0	1.85	2.04
gossip	1.30	0	1.78
wuthering	0	0	2.58

log frequency weighting
& cosine normalization

term	SaS	PaP	WH
affection	0.789	0.832	0.524
jealous	0.515	0.555	0.465
gossip	0.335	0.0	0.405
wuthering	0.0	0.0	0.588

- $\cos(\text{SaS},\text{PaP}) \approx$
  $0.789 * 0.832 + 0.515 * 0.555 + 0.335 * 0.0 + 0.0 * 0.0 \approx 0.94.$
- $\cos(\text{SaS},\text{WH}) \approx 0.79$
- $\cos(\text{PaP},\text{WH}) \approx 0.69$
- Why do we have $\cos(\text{SaS},\text{PaP}) > \cos(\text{SAS},\text{WH})$?

Basic Search Engine using Vector Space Model

- Represent the query as a weighted tf-idf vector
- Represent each document as a weighted tf-idf vector
- Compute the cosine similarity between the query vector and each document vector
- Rank documents with respect to the query
- Return the top K (e.g., $K = 10$) to the user

Applications

What can we do now?

- Search documents based on a query (Information Retrieval) – basis for search engines
- Build a question-answering system – input is a natural language question and we find *sentences* that are similar to the question
- Summarize longer texts, but removing sentences that have a high similarity (thus deemed redundant)
- Compute document similarity – e.g., for detecting *plagiarism* in submissions or finding similar contracts in *case law*
- Make recommendations (movies, photos, music, products, . . .) using *user-to-item* and *item-to-item* similarities (e.g., using tag vectors)

We will later see more sophisticated encodings and models.

Outline

1. NLP Applications
2. Processing & Vectorization
3. Document Vector Space Model
4. **Notes and Further Reading**

Reading Material

Required

- [MRS08, Chapter 6] (Vector Space Model, tf-idf)

Supplemental

- [MRS08, Chapter 8] (Evaluation)

References

[LHH19] Hobson Lane, Cole Howard, and Hannes Max Hapke.
Natural Language Processing in Action.
Manning Publications Co., 2019.
https://concordiauniversity.on.worldcat.org/oclc/1102387045.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze.
Introduction to Information Retrieval.
Cambridge University Press, 2008.
http://informationretrieval.org.