Natural Language Processing

Prof. Dr. Jannik Strötgen
jannik.stroetgen@h-ka.de
Summer 2024

Hochschule Karlsruhe
University of
Applied Sciences

Fakultät für
Informatik und
Wirtschaftsinformatik

Recap

Named Entity Recognition

Named Entity Annotation

Person

Location

Organization

Number

Time / Date

Common Types of Named Entities

The core set (by definition):

- Persons (e.g., `Elon Musk`)
- Locations (e.g., `Karlsruhe`)
- Organizations (e.g., `EU`)

Further useful named entity types:

- Dates (e.g., `Friday, 13.05.`)
- Times (e.g., `13:37`)
- Numeric expressions (e.g., `$43bn`)

Domain-specific entity types:

- Chemicals (e.g., C_2H_5OH)
- Genes (e.g., `TP53`)
- Stock symbols (e.g., `AAPL`)
- Laws (e.g., `StGB`)
- URLs (e.g., `www.h-ka.de`)
- etc...

Named Entity Ambiguities

Language is ambiguous. Named entities are no exception…

Example:

The **Tesla** chief executive revealed on Thursday that he had launched a hostile takeover of Twitter.

$$T = \frac{Vs}{m^2}$$

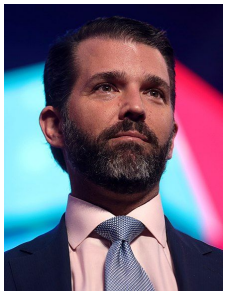
Company? Person? Unit?

Named Entity Normalization: Definitions

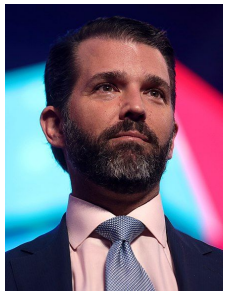
Donald John Trump Jr. is an American political activist, businessman, conspiracy theorist, author, and former television presenter. He is the eldest child of the 45th president of the United States, **Donald J. Trump**.

NE **disambiguation** (NED): The task of deciding whether two entity mentions refer to the same entity.

NE **linking** (NEL): The task of linking an entity mention to a unique identifier.



≠



Donald Trump Jr.

From Wikipedia, the free encyclopedia

Outline

Outline of the Course

1. Foundations and Pre-processing
2. Part-of-speech Tagging
3. Parsing
4. Named Entity Recognition and Linking
5. **Similarity and Search**
6. Language Models: Static Word Embeddings
7. Contextual Language Models
8. Text Mining (Classification, Clustering, and Topic Models)
9. Opinion Mining and Sentiment Analysis
10. Relation Extraction and Question Answering
11. Applications in Document Analysis

Outline for Today

Similarity and Search

1. Word Similarity
2. Surface Form Similarity
    - Phonological Similarity
    - Morphological Similarity
    - Spelling Similarity
3. Semantic Similarity
4. The Vector Space Model
    - Boolean retrieval
    - TF-IDF
5. Document Similarity
    - Document Vectors
    - Cosine Similarity

Word Similarity

What’s the difference?

What’s the difference between a **hippo** and a **zippo**?

One is really heavy and the other is a little lighter.

Word Similarity: Experimental Results

People were asked to rate the similarity of two words:

- Scale: 0 (not similar) - 10 (similar)
- Result: no consistent agreement between human annotators

⇨ When are two words similar?

word 1	word 2	sim
tiger	cat	7.35
tiger	tiger	10.00
book	paper	7.46
computer	keyboard	7.62
computer	internet	7.58
plane	car	5.77
train	car	6.31
telephone	communication	7.50
television	radio	6.77
media	radio	7.42
drug	abuse	6.85
bread	butter	6.19
cucumber	potato	5.92

Finkelstein et al. *Placing search in context: The concept revisited*. International Conference on World Wide Web (2001)

Types of Similarity in Language

Surface form similarity:

- Phonological similarity (e.g., `brake | break`)
- Morphological similarity (e.g., `respect | respectful`)
- Spelling similarity (e.g., `theater | theatre`)

Semantic similarity:

- Synonymy (e.g., `verbose | wordy`)
- Hyponymy (e.g., `color | red`)

Content similarity:

- Sentence similarity (e.g., paraphrases)
- Document similarity (e.g., two news stories on the same event)

Surface Form Similarity

Phonological Similarity: Names

Variations of `John / Jon`:

- `Gianni` (Italian)
- `Ivan` (Russian)
- `Jan` (Dutch)
- `Jean` (French)
- `Johann` (German)
- `Juan` (Spanish)

  - Uniform spelling is a (relatively) modern phenomenon.
  - Names tend to have high spelling variation
  - Variation is even higher across languages
  - This is problematic for statistical approaches (e.g., a census)

Soundex Algorithm: Outline

Soundex is a class of heuristics for expanding a query into phonetic equivalence classes. It is mainly used for names and is language-specific. Words with similar pronunciation are encoded to the same representation so they can be matched despite minor differences in spelling.

Idea:

- Turn every token into a 4-character reduced form
- Build an index on the reduced forms
- Apply the same encoding to query terms (= tokens in the query)
- Search the index for phonetically similar tokens that have the same encoding

Soundex Algorithm

1. Retain the first character of the word.
2. Replace all occurrences `A, E, I, O, U, H, W, Y` with the digit `0`
3. Change characters from the following sets into digits:
   - `1 ← B, F, P, V`
   - `2 ← C, G, J, K, Q, S, X, Z`
   - `3 ← D, T`
   - `4 ← L`
   - `5 ← M, N`
   - `6 ← R`
4. If two adjacent digits are identical, remove one of them.
5. Remove all zeros from the string
6. Return the first four characters of the string (pad with trailing `0`s if necessary)

Surname (asc)	Surname.Soundex
ADAMSKI	A352
ADIYATIPAR...	A331
AHMED	A530
AITKEN	A325
ALLAN	A450
ALLEN	A450
ALLERD	A463
AMPHLETT	A514
ANDERSON	A536
ANDERSON	A536
ANGUS	A522
ANWING	A552

Soundex: Disadvantages

- Language specificness: originally developed for English
- Homophonous names starting with a different character
  - `Craft  (C613)`
  - `Kraft  (K613)`
- Unable to discriminate between long words (4-character limit)
- Conflation of unrelated family names:
  - `Saint  (S530)`
  - `Sand   (S530)`
  - `Snead  (S530)`
  - `Sunday (S530)`

Morphological Similarity

Recall (from Slide Set 1) that inflectional and derivational **morphology** describe the modification of words, typically from a common **root**.

Inflectional Morphology:

- Variation in the form of a word, to express a grammatical contrast.
- Example: `run → runs | running`

Derivational Morphology:

- The formation of a new word or inflectable stem from another word or stem.
- Example: `compute → computer → computerization`

Morphological Similarity:

- We can consider words to be similar if they share a root
- We can use stemming or lemmatization to measure similarity

Spelling Similarity: Spell Checking

Principal uses

- Spell checking suggestions (e.g., `spell chekcing`)
- Correcting documents in a collection to improve corpus quality
- Retrieving matching documents when the query contains a spelling error

There are two flavors of spell checking:

- **Isolated words**
  - Check each word on its own for misspelling
  - Will not catch typos resulting in correctly spelled words (e.g., `from` and `form`)
- Context-sensitive
  - Look at the context provided by surrounding tokens
  - Example: `I flew form Heathrow to Zürich`

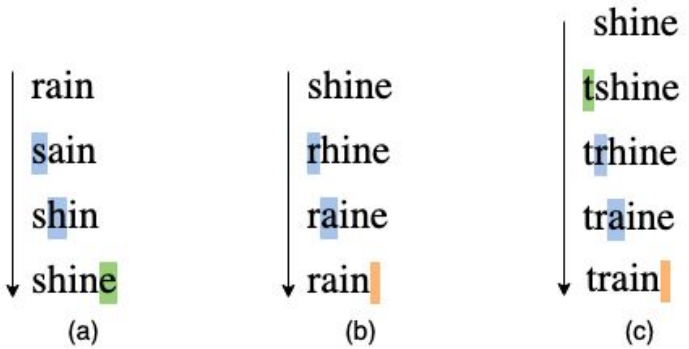
Isolated Word Correction

Word similarity for word correction:

- Given a lexicon and a word `w`, return the words in the lexicon that are closest to `w`
- But what do we mean by “closest”?

Instead of similarity, we often consider distance:

- Which words have the smallest distance? → Which words are similar?
- Example: what is the distance between:
  - `rain` and `shine`?
  - `shine` and `train`?



https://devopedia.org/levenshtein-distance

Levenshtein Distance

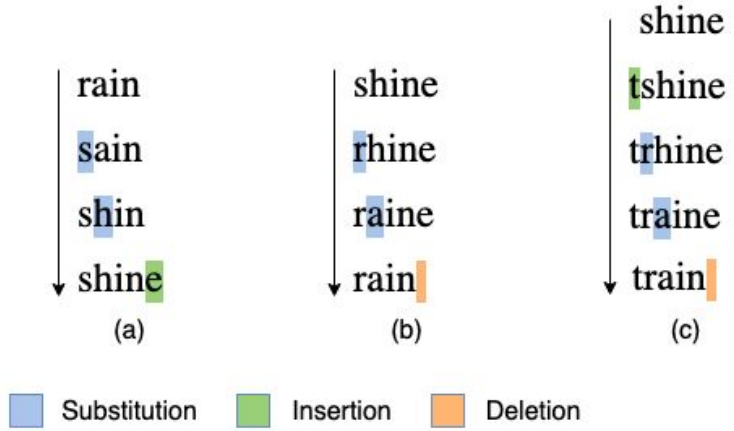
Idea behind the Levenshtein Distance (**edit distance**): Given two strings `S1` and `S2`, count the minimum number of basic operations to convert one string to the other.

Basic operations are typically character-level:

- Insert
- Delete
- Replace (i.e., substitute)

Example:

- The edit distance between `rain` and `shine` is 3
- We need to replace two characters and insert one character



https://devopedia.org/levenshtein-distance

Levenshtein Distance: Wagner-Fischer Algorithm (1)

Step 1: Setup and parameters
Set `n` to be the length of string `s`
Set `m` to be the length of string `t`
If `n=0`, return `m` and exit.
In `m=0`, return `n` and exit.
Construct a matrix containing `0…m` rows and `0…n` columns.

Step 2: Initialization
Initialize the zeroth row to `0…n`
Initialize the zeroth column to `0…m`

Example:

		c	a	t
	0	1	2	3
h	1			
a	2			
t	3			

`s = cat, n=3`

`t = hat, m=3`

Levenshtein Distance: Wagner-Fischer Algorithm (2)

Step 3: Iteration

For `j` from `1…n` // iterate over columns

  For `i` from `1…m` // iterate over rows

    if `s[i] = t[j]` then `subCost := 0` // retain character

    if `s[i] ≠ t[j]` then `subCost := 1` // replace characte

```
    d[i, j] := minimum(d[i-1,   j] + 1,        // deletion
                       d[i,   j-1] + 1,        // insertion
                       d[i-1, j-1] + subCost)  // substitution
```

return `d[m, n]`

Example:

		c	a	t
	0	1	2	3
h	1	1	2	3
a	2	2	1	2
t	3	3	2	1

Wagner-Fischer Algorithm: Example

		S	T	R	E	N	G	T	H
	0	1	2	3	4	5	6	7	8
T	1								
R	2								
E	3								
N	4								
D	5								

Wagner-Fischer Algorithm: Example

		S	T	R	E	N	G	T	H
	0	1	2	3	4	5	6	7	8
T	1	1							
R	2								
E	3								
N	4								
D	5								

Parameters and choices:

`[1,1] S ≠ T` → subCost = 1

delete: 1+1=2

insert: 1+1=2

substitute: 0+1=1

→ substitute is cheapest

Wagner-Fischer Algorithm: Example

		S	T	R	E	N	G	T	H
	0	1	2	3	4	5	6	7	8
T	1	1	1						
R	2								
E	3								
N	4								
D	5								

Parameters and choices:

`[1,2] T = T` → subCost = 0

delete: 2+1=3

insert: 1+1=2

substitute: 1+0=1

→ substitute is cheapest

Wagner-Fischer Algorithm: Example

		S	T	R	E	N	G	T	H
	0	1	2	3	4	5	6	7	8
T	1	1	1	2					
R	2								
E	3								
N	4								
D	5								

Parameters and choices:

`[1,3] R ≠ T` → subCost = 1

delete: 3+1=4

insert: 1+1=2

substitute: 2+1=3

→ insert is cheapest

Wagner-Fischer Algorithm: Example

		S	T	R	E	N	G	T	H
	0	1	2	3	4	5	6	7	8
T	1	1	1	2	3	4	5	6	7
R	2	2							
E	3								
N	4								
D	5								

Parameters and choices:

`[2,1] S ≠ R` → subCost = 1

delete: 1+1=2

insert: 2+1=3

substitute: 1+1=2

→ delete and substitute are both equally cheap

Wagner-Fischer Algorithm: Example

		S	T	R	E	N	G	T	H
	0	1	2	3	4	5	6	7	8
T	1	1	1	2	3	4	5	6	7
R	2	2	2	1					
E	3								
N	4								
D	5								

Parameters and choices:

`[2,3] R = R` → subCost = 0

delete: 2+1=3

insert: 2+1=3

substitute: 1+0=1

→ substitute is cheapest

Wagner-Fischer Algorithm: Example

		S	T	R	E	N	G	T	H
	0	1	2	3	4	5	6	7	8
T	1	1	1	2	3	4	5	6	7
R	2	2	2	1	2	3	4	5	6
E	3	3	3	2	1	2	3	4	5
N	4	4	4	3	2	1	2	3	4
D	5	5	5	4	3	2	2	3	4

Output:
The Levenshtein distance is 4

`STrend`: insert S at pos 0
`Strend`: retain T at pos 1
`Strend`: retain R at pos 2
`Strend`: retain E at pos 3
`Strend`: retain N at pos 4
`Streng`: replace D with G at pos 5
`Strengt`: insert T at pos 6
`Strength`: insert H at pos 7

Weighted Edit Distance

Core idea:

- Different costs for insert / delete operations
- Cost matrices for replace
- Can use domain-specific knowledge

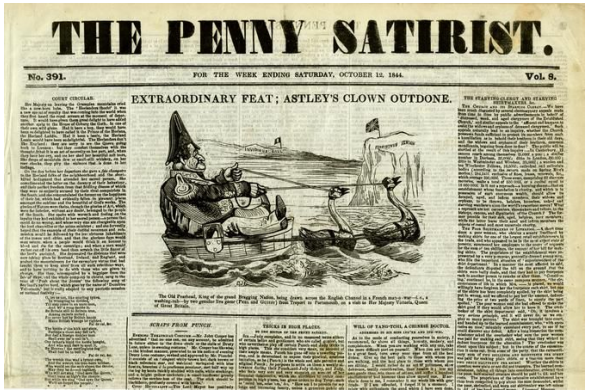
Damerau modification:

- **Swaps** of two adjacent characters also have a cost of 1
- Example:
  - `Lev(cats, cast) = 2`
  - `Dam(cats, cast) = 1`

Specialized Edit Distances

Consider these edit distances. Why (or where) could they make sense?

`DistA(sit down, sit clown) = 1`



`DistA` models typical OCR errors that occur when processing old newspapers.

`DistB(qeather, weather) = 1`
`DistB(leather, weather) = 2`

`DistB` models typing errors as a result of using a keyboard (= fat fingers)

Semantic Similarity

Semantic Similarity: Example

The **S&P 500** **climbed** 6.9%, to 1,237, its **best close** since June 12, 2001.

The **Nasdaq** **gained** 12.2%, to 2,123 for its **best showing** since June 8, 2001.

The **DJIA** **rose** 68.46%, to 10,723, its **highest level** since March 15.

Semantic Relations

Synonymy:

- Different words with similar meaning (e.g., `big | large`)
- Synonyms differ in their frequency of use and the context

Antonymy:

- Words that are near opposites (e.g., `raise | lower`)

Hypernymy:

- Supertype of a word (e.g., `red` is a `color`)

Hyponymy:

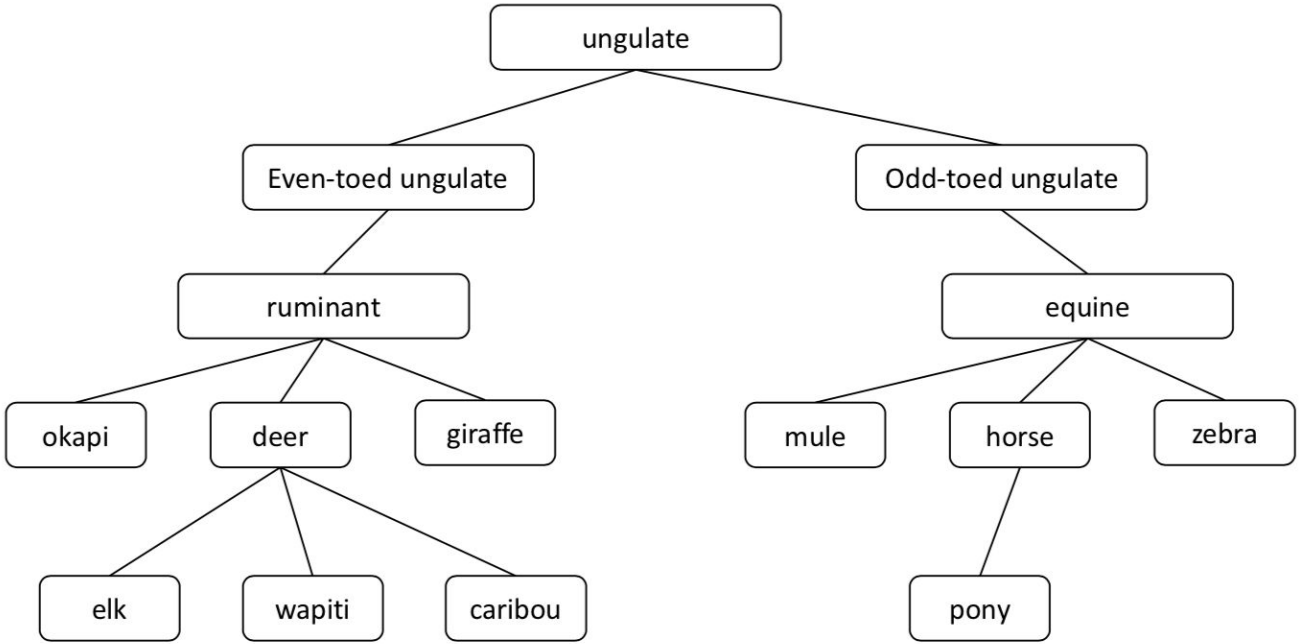
- Subtype of a word (inverse of hypernymy)

Meronymy:

- A word is part of a larger whole (e.g., a `finger` is a meronym of `hand`)

WordNet

WordNet is a database of words and semantic relations between them. The main relation is hypernymy, so the overall structure is tree-like.



WordNet: Example

The **noun** `bar` has 11 senses:

- `Barroom, bar, saloon, ...` (a room where drinks are served)
- `Bar` (counter where you can purchase food or drink)
- `bar` (unit of pressure)
- `...`

The **verb** `bar` has 4 senses:

- `barricade, block, ...`
- `bar, debar, exclude, ...`
- `...`

WordNet: Structure

Parent hierarchy of the different meanings of `bar`:

- `Barroom, bar → room → area → structure → artifact ...`
- `Bar → counter → table → furniture → ... → artifact ...`
- `Bar → implement → instrumentation → artifact ...`
- `Bar → musical notation → notation → writing → ...`

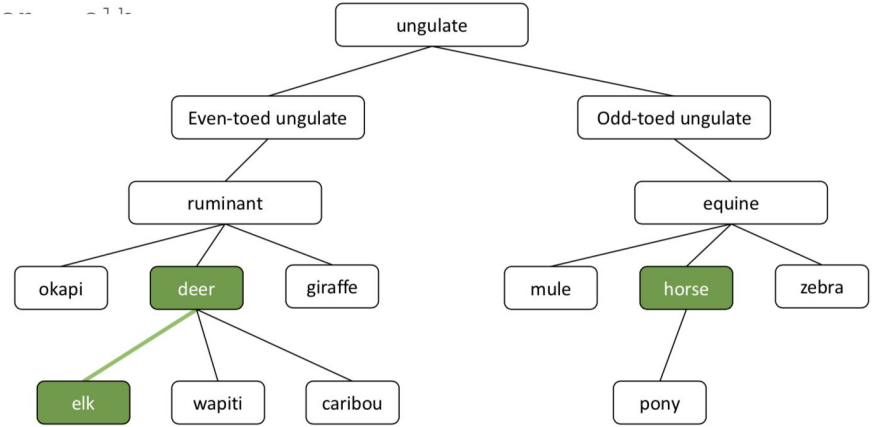
However:

- Not all words share the same root
- WordNet has multiple roots
- WordNet is a forest, not just a tree

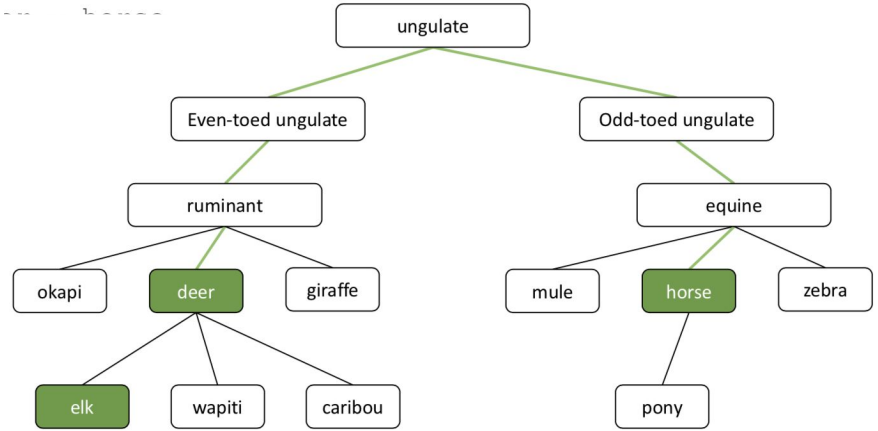
How is this helpful for computing word similarities?

WordNet: Tree-based Similarity

Which pair is more similar? `Deer - elk` or `deer - horse`?



`TreeDist(deer, elk) = 1`



`TreeDist(deer, horse) = 6`

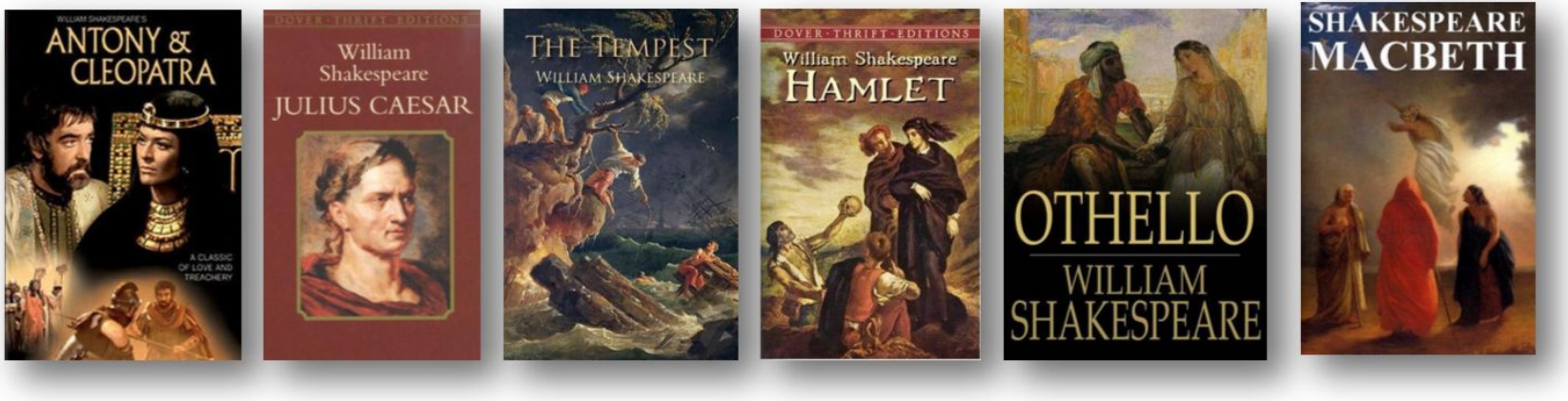
In principle, **the higher the distance the lower the similarity**! But there are problems:

- A specific word may not be in any tree
- Hypernymy edges are not all equally apart in similarity space
- Many more detailed graph-based semantic similarity measures have been developed.

The Vector Space Model

Comparing and Querying Documents

So far, we have discussed similarity between words. But what about documents?



For example, if we have a corpus with all plays by Shakespeare, how can we identify all plays that contain the words `Brutus` and `Caesar`?

→ How can we **search**?

Boolean Retrieval

The simplest information retrieval system:

- Create an index of all words in the documents
- Model queries as Boolean expressions (`Caesar AND Brutus`)
- The retrieval engine returns all documents that match the expression
- Stemming and lemmatization can help to improve recall

Term-Document Incidence Matrix

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Mcbeth	...
Antony	1	1	0	0	0	1	...
Brutus	1	1	0	1	0	0	...
Caesar	1	1	0	1	1	0	...
Calpurnia	0	1	0	0	0	0	...
Cleopatria	1	0	0	0	0	0	...
mercy	1	0	1	1	1	1	...
worser	1	0	1	1	1	1	...

Indexing and retrieval:

- Create a matrix of the document collection that contains all distinct terms
- Set the value to `1` if the corresponding document contains the given term
- Return all documents that have a value of `1` in all cells corresponding to query terms

Scoring for Ranked Retrieval

But what happens if many documents match?

→ We want to return documents in an order that is 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 to each document (typically in the range `[0,1]`)
- The score measures how well the document and the query "match"
- We need a way to assign a score to a query/document pair

Term Frequency Matrix

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Mcbeth	...
Antony	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	...
Cleopatria	57	0	0	0	0	0	...
mercy	2	0	3	8	5	8	...
worser	2	0	1	1	1	5	...

To have data for a scoring function, term frequency information is helpful:

- We can store term frequency counts instead of binary values in the matrix
- Each document is now represented by a **count vector**
- Note: word order is not retained in a vector. This approach is called **bag of words**.

Term Frequency (TF)

The **term frequency** $tf_{t,d}$ of term `t` in document `d` is defined as the number of times that `t` occurs in `d`.

- A document with 10 occurrences of the term `Hamlet` is more relevant for a query containing `Hamlet` than a document with just one occurrence
- But probably not 10 times more relevant

⇨ Relevance does not increase proportionally with term frequency

How can we correct for this?

Rarity of Terms

Query example: `Are spider enthusiasts arachnocentric?`

Rare terms:

- Consider a term in the query that is rare in the corpus (e.g., `arachnocentric`)
- A document containing this term is very likely to be relevant

⇨ We want large positive weights for rare terms.

Frequent terms:

- Consider a term in the query that is frequent in the corpus (e.g., `spider`)
- A document containing this term is more likely to be relevant than a document that does not, but it is less of an indicator of relevance

⇨ We want positive weights for frequent terms (but lower than for rare terms)

Inverse Document Frequency (IDF)

The **document frequency** df_t of term t is defined as
the number of documents in the corpus in which t occurs.

- We can use df_t to account for the rarity of t when computing the matching score
- The document frequency is an inverse measure of the informativeness of a term

⇨ So we need to invert it

The **inverse document frequency** idf_t is a measure of the informativeness of the term.
We define the idf_t weight of term t as:

$$idf_t = \log_{10} \frac{N}{df_t}$$

Where N is the number of documents. We use $\log N/df_t$ instead of N/df_t as a heuristic to “dampen” the effect of the inverse document frequency.

Term Frequency-Inverse Document Frequency (TF-IDF)

We combine the term frequency and the inverse document frequency to assign a `tf-idf` weight `w` to each term `t` in each document `d`:

$$w_{t,d} = (1+\log_{10} tf_{t,d}) \log_{10} \frac{N}{df_t}$$

The weight:

- Increases with the number of occurrences of a term within a document
- Increases with the rarity of the term in the collection

Note: we use log-scaling of term frequencies and assume $tf_{t,d} > 0$ (otherwise, we set $w_{t,d} = 0$). Other approaches are possible and there is no rigorous formal reasoning behind this choice. In practice, experiments help to determine suitable scaling methods.

TF-IDF Weight Matrix

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Mcbeth	…
Antony	5.25	3.18	0	0	0	0.35	…
Brutus	1.21	6.10	0	1.0	0	0	…
Caesar	8.59	2.54	0	1.51	0.25	0	…
Calpurnia	0	1.54	0	0	0	0	…
Cleopatria	2.85	0	0	0	0	0	…
mercy	1.51	0	1.90	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

- Weights encode frequency information of terms in documents
- Frequencies of terms in a document are normalized by the number of documents in which the term occurs

Document Similarity

Documents as Vectors

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

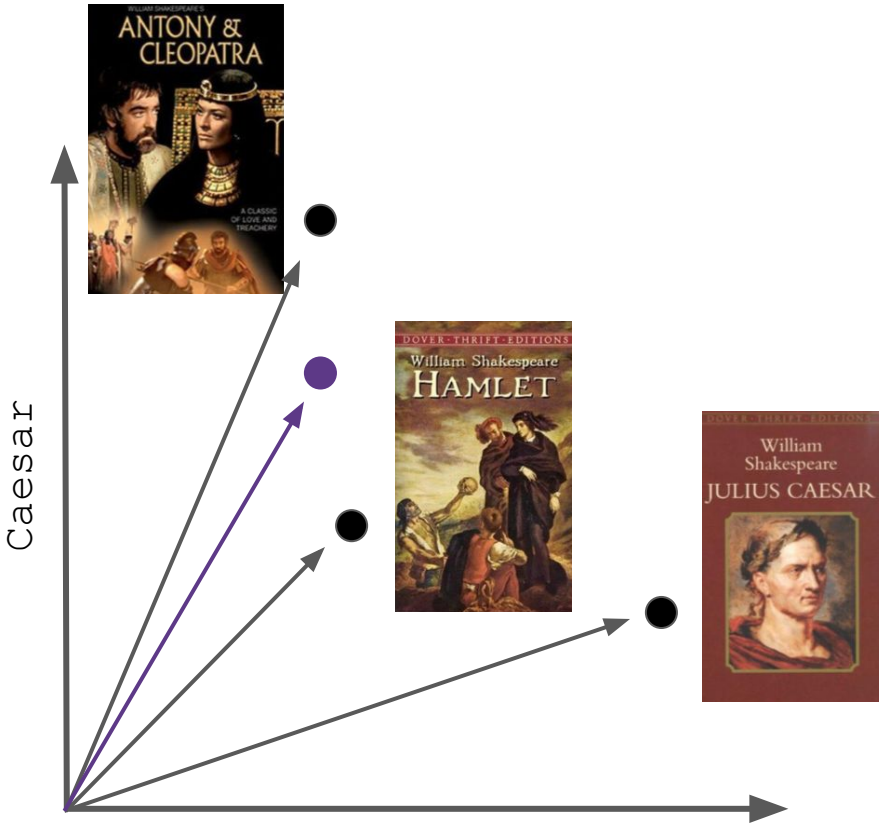
- Terms are **dimensions** (= axes) of the corresponding vector space
- Documents are points or **vectors** in this space
- The vector space is very **high-dimensional** due to the vocabulary size:
  Tens of millions of dimensions when building search engine at web scale
- Individual vectors are very **sparse**: most entries are zero

How is this helpful in querying?

- Idea: Find document vectors that are similar to the query vector in this space

⇨ Finding close points in a vector space is a well-researched problem

Documents in Vector Space



	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet
Antony	5.25	3.18	0	0
Brutus	1.21	6.10	0	1.0
Caesar	8.59	2.54	0	1.51
Calpurnia	0	1.54	0	0
Cleopatria	2.85	0	0	0
mercy	1.51	0	1.90	0.12
worser	1.37	0	0.11	4.15

Query:

Brutus killed Caesar

Measuring Distances in Vector Space

Intuitively: Euclidean distance between query vector and and document vector.
However, this is a bad idea...

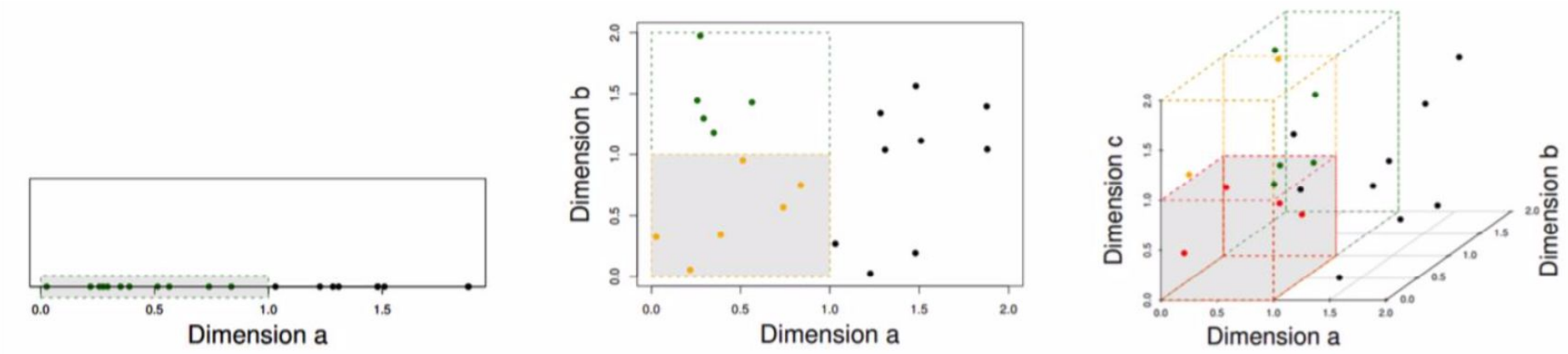
- The Euclidean distance between query and document will be large
- This is true even if the distribution of terms in the query and the distribution of terms in the document are very similar
- Reason: Euclidean distance is large for vectors of different lengths

⇨ Curse of dimensionality

Curse of Dimensionality

Problem:

- When the **dimensionality** increases, the **volume** of the space increases exponentially
- Word vector space have very high dimensionality



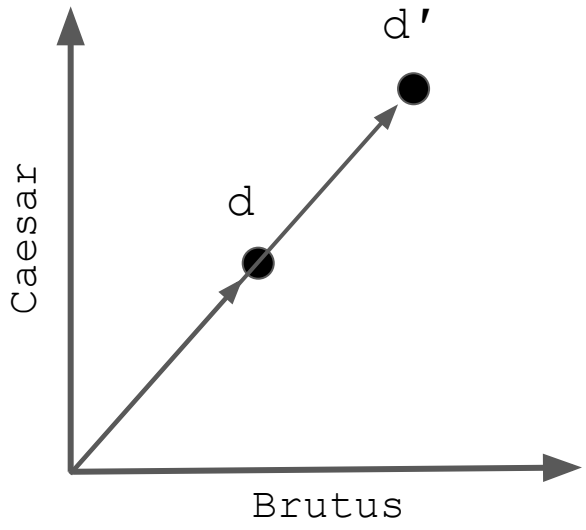
Using Angular Similarity

Solution:

- Rank documents according to their **angular distance** from the query

Thought experiment:

- Take a document `d` and append it to itself.
  Call this document `d'`
- "Semantically" `d` and `d'` have exactly the same content
- The angle between the two documents is `0°`,
  corresponding to maximal similarity
- The Euclidean distance between `d` and `d'`
  scales with the number of tokens in document `d`



Cosine Similarity

Since **cosine** is a monotonically decreasing function for the interval `[0°, 180°]`, the following two notions are equivalent:

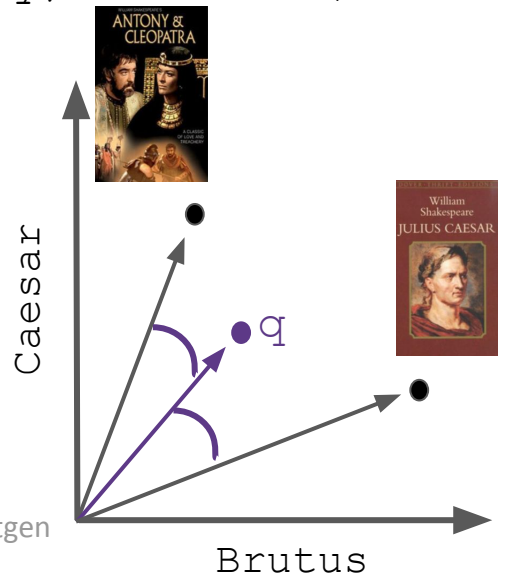
- Ranking documents in decreasing order of the angle between query and document
- Ranking documents in increasing order of `cosine(query, document)`

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

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 $\vec{d}$.



In a Nutshell: A Simple Search Engine

We can now construct a simple (yet surprisingly effective) search engine:

- Compute tf-idf weights for all terms in all documents in the corpus
- Represent the corpus as a tf-idf weighted incidence matrix
- Compute a tf-idf weighted vector representations of the query
- Compute the cosine distance between the query vector and all document vectors
- Rank the documents non-decreasingly by cosine distance score
- Return the top 10 results as the **10 blue links**

⇨ More in the assignment

Online Resources

Levenshtein Distance demo

https://phiresky.github.io/levenshtein-demo/

WordNet

https://wordnet.princeton.edu/

Further-Watching Material

Minimum Edit Distance

https://youtu.be/jhSdB36RYWk?list=PLoROMvodv4rOFZnDyrlW3-nI7tMLtmiJZ

https://youtu.be/Q7QQCNM7AJ4?list=PLoROMvodv4rOFZnDyrlW3-nI7tMLtmiJZ

Semantic Similarity

https://youtu.be/PxgkddPbjrM

Term-Document Incidence Matrix

https://youtu.be/b6cjKqTE04s?list=PLoROMvodv4rOFZnDyrlW3-nI7tMLtmiJZ

Vector Space Model

https://youtu.be/yvNt_qDRbDQ?list=PLoROMvodv4rOFZnDyrlW3-nI7tMLtmiJZ

Bonus Watching Material

https://youtu.be/Cxqca4RQd_M

https://youtu.be/9LMr5XTgeyI