

IBM Watson Information Retrieval

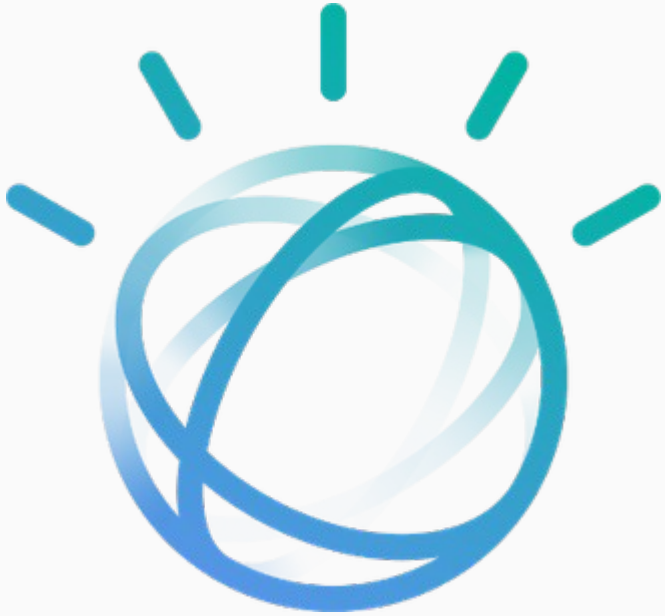


IBM Watson

2011: IBM Watson beats humans @ Jeopardy!



What is IBM Watson



Watson is a **question answering computer system** capable of answering questions posed in natural language. Watson was named after IBM's first CEO, industrialist Thomas J. Watson.

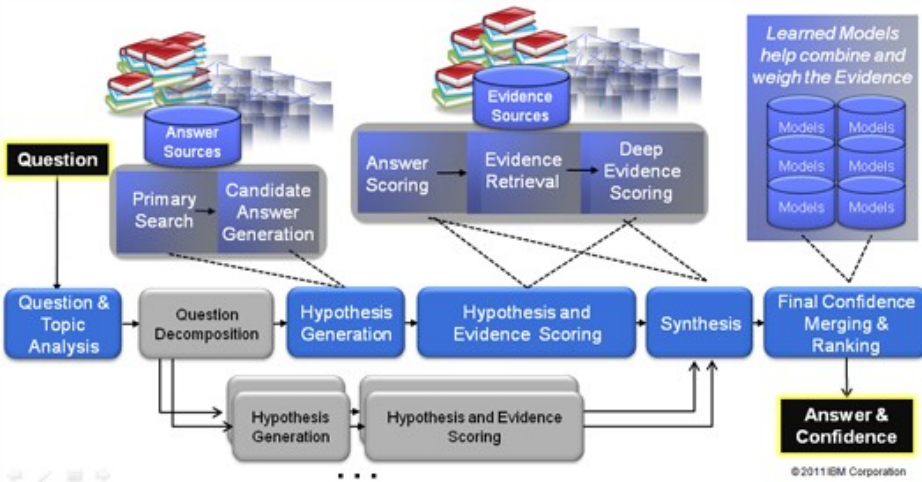
Software: IBM's DeepQA software, Apache UIMA framework, various programming languages, SUSE Linux over Apache Hadoop.

Hardware: cluster of 90 IBM Power 750 servers, 2.880 POWER7 processor threads and 16 terabytes of RAM. Cost: about 3 million \$

Data: millions of documents including encyclopedias, dictionaries, thesauri, newswire articles, and literary works. Watson also used databases, taxonomies, and ontologies (DBPedia, WordNet, and Yago). For Jeopardy! all data was stored in RAM for Watson to be competitive with humans. Watson parses this data to build its knowledge.

Operation: Watson parses questions and employs various technologies (Natural Language Processing, Information Retrieval, Knowledge Representation and Reasoning, and Machine Learning) on its data to spot probable answers.

The DeepQA Framework



1. Question decomposition: When the question is presented, Watson parses it in order to extract its major features.

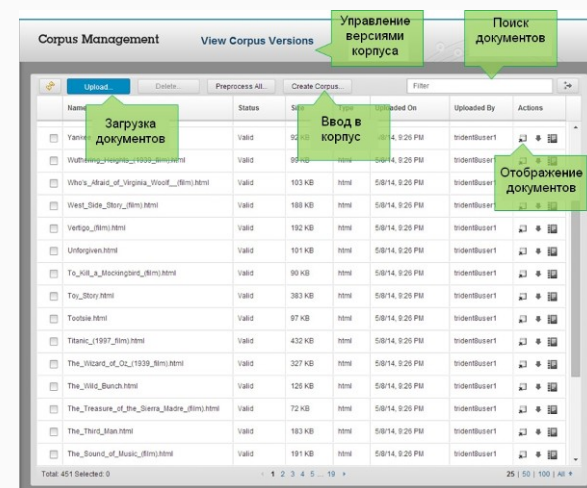
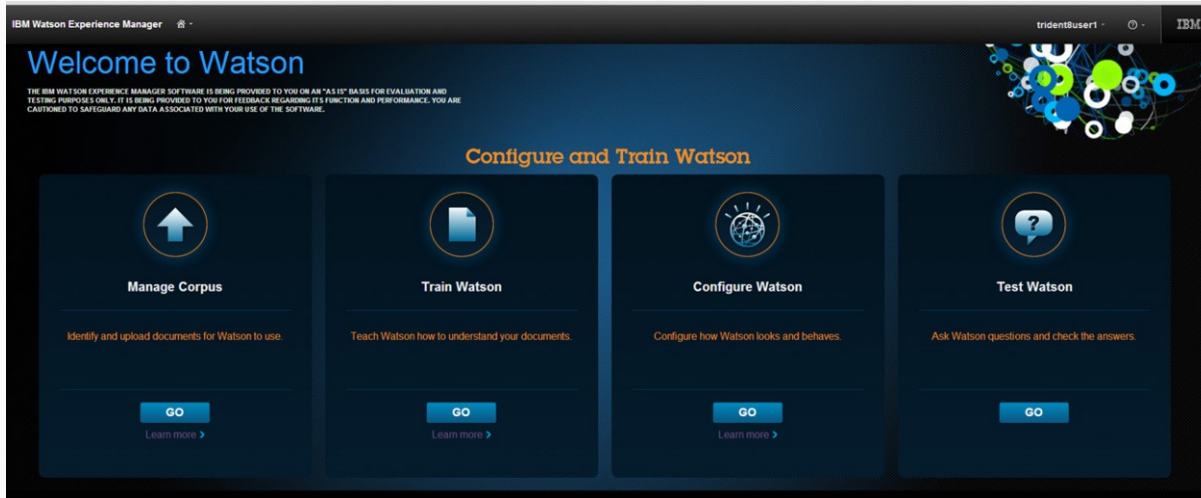
2. Hypothesis Generation: Watson searches the corpus (which consists of structured & unstructured knowledge) for passages that might contain a valuable response.

3. Hypothesis and evidence scoring: Watson compares the text of the question and the text of all potential responses with specific reasoning algorithms. Each one of these algorithms executes a different comparison (e. g: search for the matching of terms and synonyms, examine the temporal and spatial features) and then produces one or more scores that indicate the response's degree of relevance (inference) to the question.

4. Synthesis: Each resulting score is weighted against a statistical model that captures the algorithm's performance at the establishment of inference between two similar passages for that domain, during Watson's training period. This statistical model is then used to summarize a level of confidence as Watson's metric of evidence that the candidate answer is inferred by the question.

5. Final Confidence Merging and Ranking: The process (steps 1-4) is repeated for all of the candidate

Watson Experience Manager (Data Science class of 2016)



For a given topic - subject:

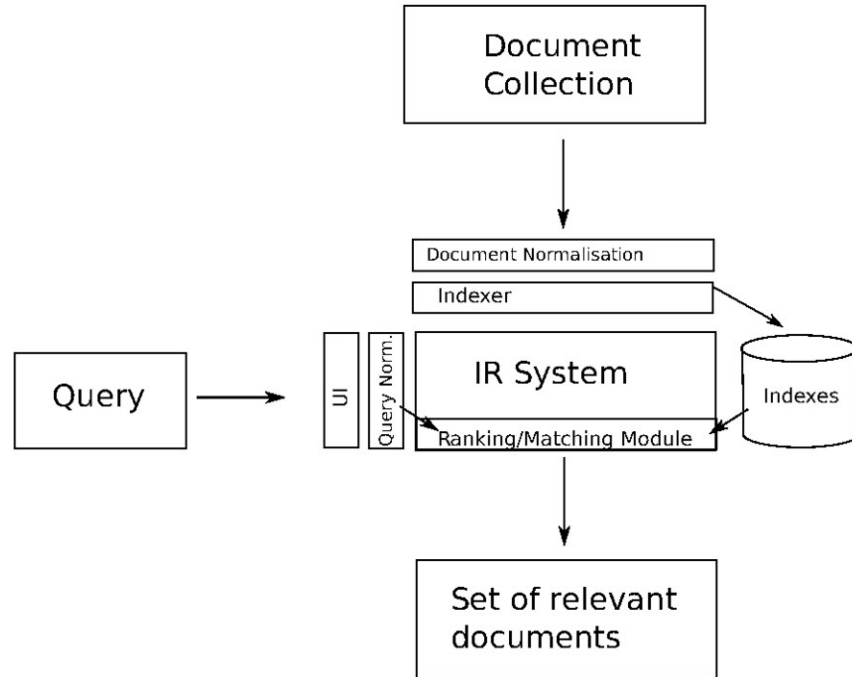
- A team collects possible questions (in natural language) on the topic
- A team creates a corpus of documents (html pdf etc) that potentially contain answers to these questions
- Another team matches each question to text excerpts inside the documents that are potential answers to the question
- An expert team evaluates the question answer pairs
- Watson is automatically trained and learns a topic model
- Watson is ready to be deployed as a web service and answer questions on the topic
- The process can be iterated to increase knowledge on the topic or add more

Some serious drawbacks:

- Huge number of questions, answers and documents needed (order of thousands)
- Black-box algorithms and training
- Not obvious integration as a service

Information Retrieval

IR System Components



QUERY: "march health awareness"

D1: "the Health Observances for March "

D2: "the Health oriented Calendar"

D3: "the Awareness News for March Awareness"

$D=3$; ; $IDF = \log\left(\frac{D}{df_j}\right)$ df_j = number of documents containing term j

TERMS	Q	COUNTS $TF_{i,j}$			df_j	D/df_j	IDF_j	WEIGHTS, $W_{I=}$	WEIGHTS $W_{I=F_{I,j}} * IDF_j$		
		D1	D2	D3				$F_{Q,j} * IDF_j$	D1	D2	D3
Health	1	1	1	0	2	$3/2=1.5$	0.1761	0.1761	0.1761	0.1761	0
Observances	0	1	0	0	1	$3/1=3$	0.4771	0	0.4771	0	0
For	0	1	0	1	2	$3/2=1.5$	0.1761	0	0.1761	0	0.0881
March	1	1	0	1	2	$3/2=1.5$	0.1761	0.1761	0.1761	0	0.0881
Awareness	1	0	0	2	1	$3/1=3$	0.4771	0.4771	0	0	0.4771
Oriented	0	0	1	0	1	$3/1=3$	0.4771	0	0	0.4771	0
Calendar	0	0	1	0	1	$3/1=3$	0.4771	0	0	0.4771	0
News	0	0	0	1	1	$3/1=3$	0.4771	0	0	0	0.2285
the	0	1	1	1	3	$3/3=1$	0	0	0	0	0

- **Ranking** search results: why it is important (as opposed to just presenting a set of unordered Boolean results)
- **Term frequency**: This is a key ingredient for ranking.
- **Tf-idf ranking**: best known traditional ranking scheme
- And one explanation for why it works: **Zipf's Law**
- **Vector space model**: One of the most important formal models for information retrieval (along with Boolean and probabilistic models)

- Thus far, our queries have been **Boolean**.
 - Documents either match or don't.
- **Good for expert users** with precise understanding of their needs and of the collection.
- Also **good for applications**: Applications can easily consume 1000s of results.
- **Not good for the majority of users**
- Don't want to write Boolean queries or wade through 1000s of results.
- This is particularly true of web search.

Problem with Boolean search: Feast or famine

- Boolean queries often have either too few or too many results.

Query 1

standard AND user AND dlink AND 650

→ 200,000 hits **Feast!**

Query 2

standard AND user AND dlink AND 650
AND no AND card AND found

→ 0 hits **Famine!**

- In Boolean retrieval, it takes a lot of skill to come up with a query that produces a manageable number of hits.
- In ranked retrieval, “feast or famine” is less of a problem.
- Condition: **Results that are more relevant are ranked higher than results that are less relevant.** (i.e., the ranking algorithm works.)

Scoring as the basis of ranked retrieval

- Rank documents in the collection according to how relevant they are to a query
- Assign a score to each query-document pair, say in $[0, 1]$.
- This score measures how well document and query “match”.
- If the query consists of just one term ...

lioness

- Score should be 0 if the query term does not occur in the document.
- The more frequent the query term in the document, the higher the score
- We will look at a number of alternatives for doing this.

Take 1: Jaccard coefficient

- A commonly used measure of overlap of two sets
- Let A and B be two sets
- Jaccard coefficient:

$$\text{JACCARD}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

($A \neq \emptyset$ or $B \neq \emptyset$)

- $\text{JACCARD}(A, A) = 1$
- $\text{JACCARD}(A, B) = 0$ if $A \cap B = \emptyset$
- A and B don't have to be the same size.
- Always assigns a number between 0 and 1.

Jaccard coefficient: Example

- What is the query-document match score that the Jaccard coefficient computes for:

Query

"ides of March"

Document

"Caesar died in March"

- $JACCARD(q, d) = 1/6$

What's wrong with Jaccard?

- It doesn't consider **term frequency** (how many occurrences a term has).
 - Rare terms are more informative than frequent terms.
 - Jaccard does not consider this information.
- We need a more sophisticated way of **normalizing** for the length of a document.
 - Later in this lecture, we'll use $|A \cap B| / \sqrt{|A \cup B|}$ (cosine) ...
 - ... instead of $|A \cap B| / |A \cup B|$ (Jaccard) for length normalization.

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** $\in \{0, 1\}^{|V|}$.

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** $\in \mathbb{N}^{|V|}$.

- We do not consider the **order** of words in a document.
- Represented the same way:

John is quicker than Mary
Mary is quicker than John

- This is called a **bag of words model**.
- In a sense, this is a step back: The positional index was able to distinguish these two documents.
- We will look at “recovering” positional information later in this course.
- For now: bag of words model

- The term frequency $tf_{t,d}$ of term t in document d is defined as the number of times that 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 because:
- A document with $tf = 10$ occurrences of the term is more relevant than a document with $tf = 1$ occurrence of the term.
- But not 10 times more relevant.
- Relevance does not increase proportionally with term frequency.

Instead of raw frequency: Log frequency weighting

- The log frequency weight of term t in d is defined as follows

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

$\text{tf}_{t,d}$	$w_{t,d}$
0	0
1	1
2	1.3
10	2
1000	4

- Score for a document-query pair: sum over terms t in both q and d :
 $\text{tf-matching-score}(q, d) = \sum_{t \in q \cap d} (1 + \log \text{tf}_{t,d})$
- The score is 0 if none of the query terms is present in the document.

- 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.
- Now: excursion to an important statistical observation about language.

- How many frequent vs. infrequent terms should we expect in a collection?
- In natural language, there are a few very frequent terms and very many very rare terms.

Zipf's law

The i^{th} most frequent term has frequency cf_i proportional to $1/i$:

$$cf_i \propto \frac{1}{i}$$

- cf_i is collection frequency: the number of occurrences of the term t_i in the collection.

Zipf's law

The i^{th} most frequent term has frequency cf_i proportional to $1/i$:

$$cf_i \propto \frac{1}{i}$$

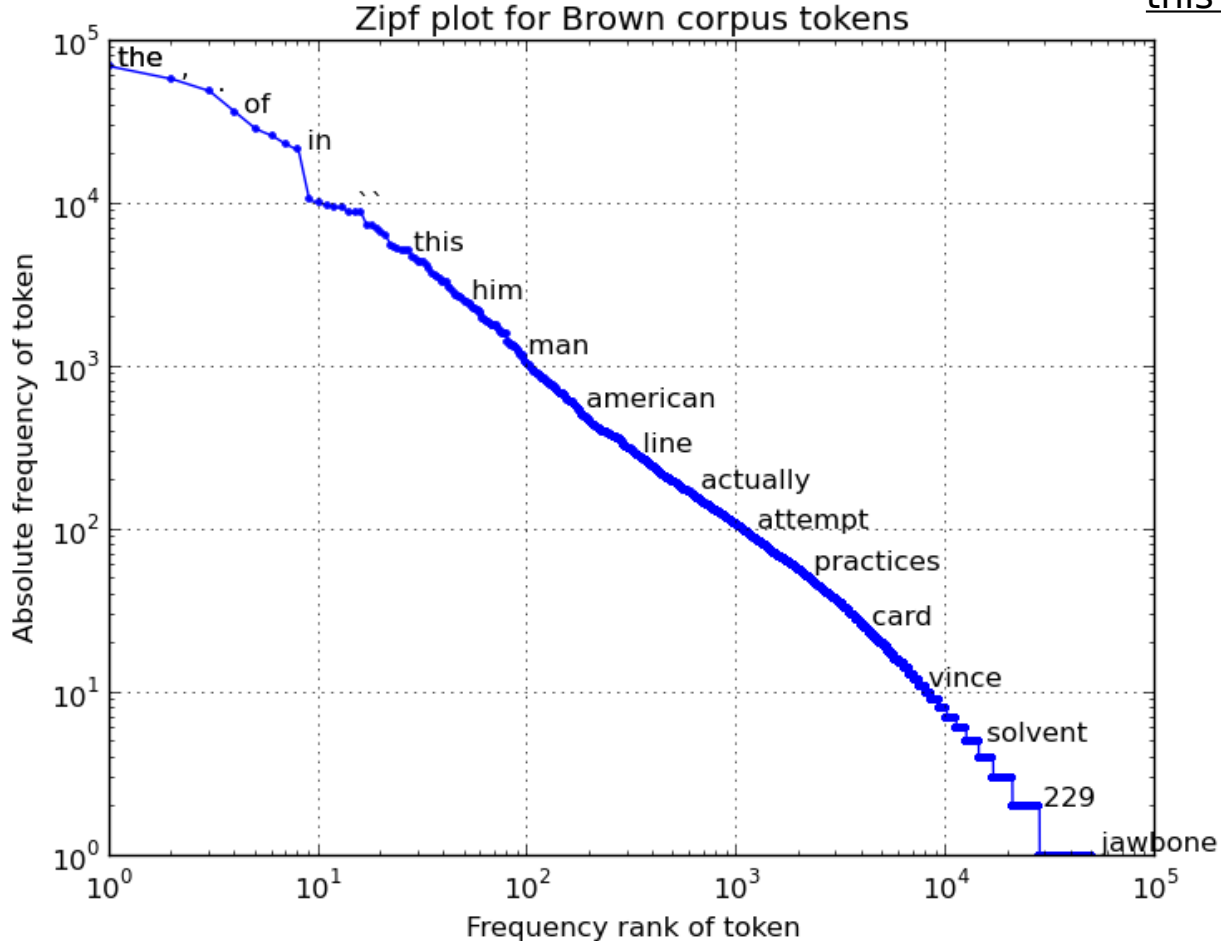
- So if the most frequent term (*the*) occurs cf_1 times, then the second most frequent term (*of*) has half as many occurrences $cf_2 = \frac{1}{2}cf_1 \dots$
- ... and the third most frequent term (*and*) has a third as many occurrences $cf_3 = \frac{1}{3}cf_1$ etc.
- Equivalent: $cf_i = ci^k$ and $\log cf_i = \log c + k \log i$ (for $k = -1$)
- Example of a power law

Zipf's Law: Examples from 5 Languages

Top 10 most frequent words in a large language sample:

English		German		Spanish		Italian		Dutch						
1	the	61,847	1	der	7,377,879	1	que	32,894	1	non	25,757	1	de	4,770
2	of	29,391	2	die	7,036,092	2	de	32,116	2	di	22,868	2	en	2,709
3	and	26,817	3	und	4,813,169	3	no	29,897	3	che	22,738	3	het/'t	2,469
4	a	21,626	4	in	3,768,565	4	a	22,313	4	è	18,624	4	van	2,259
5	in	18,214	5	den	2,717,150	5	la	21,127	5	e	17,600	5	ik	1,999
6	to	16,284	6	von	2,250,642	6	el	18,112	6	la	16,404	6	te	1,935
7	it	10,875	7	zu	1,992,268	7	es	16,620	7	il	14,765	7	dat	1,875
8	is	9,982	8	das	1,983,589	8	y	15,743	8	un	14,460	8	die	1,807
9	to	9,343	9	mit	1,878,243	9	en	15,303	9	a	13,915	9	in	1,639
10	was	9,236	10	sich	1,680,106	10	lo	14,010	10	per	10,501	10	een	1,637

this is a log-scale plot



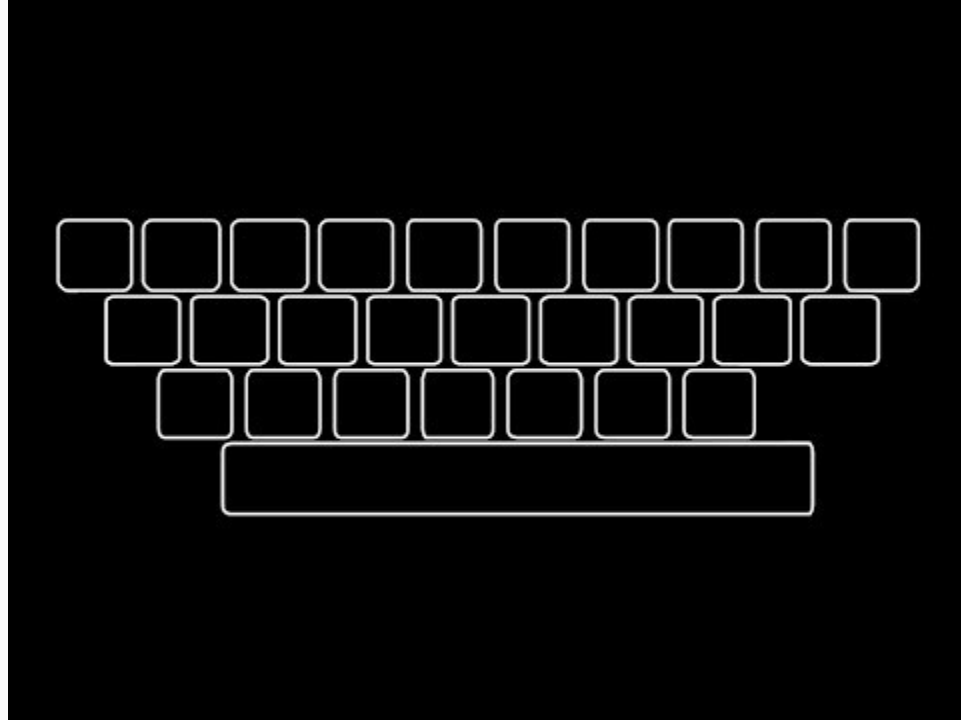
Zipf's law: Rank \times Frequency \sim Constant

English:	Rank R	Word	Frequency f	$R \times f$
	10	he	877	8770
	20	but	410	8200
	30	be	294	8820
	800	friends	10	8000
	1000	family	8	8000

German:	Rank R	Word	Frequency f	$R \times f$
	10	sich	1,680,106	16,801,060
	100	immer	197,502	19,750,200
	500	Mio	36,116	18,059,500
	1,000	Medien	19,041	19,041,000
	5,000	Miete	3,755	19,041,000
	10,000	vorläufige	1.664	16,640,000

- Sizes of settlements
- Frequency of access to web pages
- Income distributions amongst top earning 3% individuals
- Korean family names
- Size of earth quakes
- Word senses per word
- Notes in musical performances
- ...

The Zipf mystery



- 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.
- → We want *high weights for rare terms* like ARACHNOCENTRIC.

- 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.
- → **For frequent terms** like GOOD, INCREASE, and LINE, we want positive weights ...
- ... but **lower weights** than for rare terms.

- 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.**

- 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:

idf weight

$$\text{idf}_t = \log_{10} \frac{N}{df_t}$$

(N is the number of documents in the collection.)

- idf_t is a measure of the **informativeness** of the term.
- $\log \frac{N}{df_t}$ instead of $\frac{N}{df_t}$ to “dampen” the effect of idf
- Note that we use the log transformation for both term frequency and document frequency.

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

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

Collection frequency vs. Document frequency

Term	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
- Clearly, INSURANCE is a more discriminating search term and should get a higher weight.
- This example suggests that df (and idf) is better for weighting than cf (and "icf").

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

tf-idf weight

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

- tf-weight
- idf-weight
- Best known weighting scheme in information retrieval
- Alternative names: tf.idf, tf x idf

- Assign a tf-idf weight for each term t in each document d :
$$w_{t,d} = (1 + \log \text{tf}_{t,d}) \cdot \log \frac{N}{\text{df}_t}$$
- 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** $\in \{0, 1\}^{|V|}$.

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 $\in \mathbb{R}^{|V|}$.

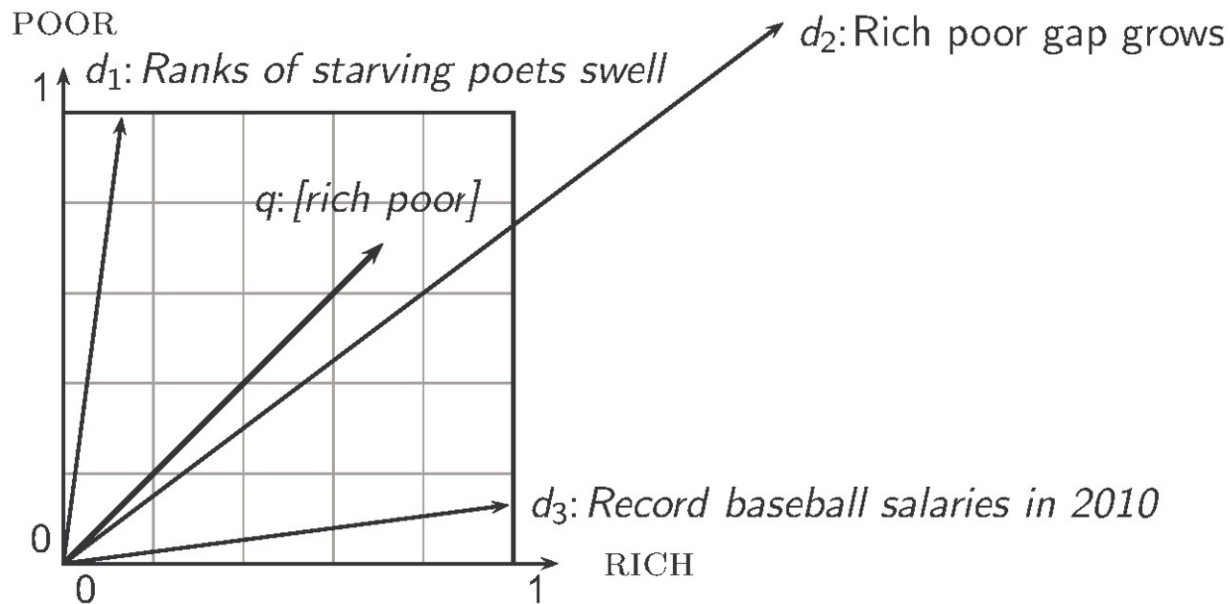
- Each document is now represented as a real-valued vector of tf-idf weights $\in \mathbb{R}^{|V|}$.
- So we have a $|V|$ -dimensional real-valued vector space.
- Terms are **axes** of the space.
- Documents are **points** or **vectors** in this space.
- Very high-dimensional: tens of millions of dimensions when you apply this to web search engines
- Each vector is very sparse - most entries are zero.

- **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 \approx negative distance
- This allows us to 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**.

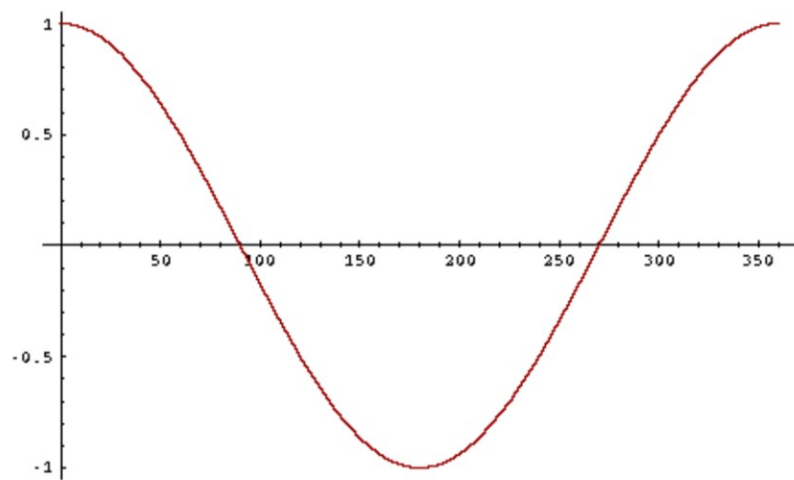
Why distance is a bad idea



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.

- 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.

- 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 $[0^\circ, 180^\circ]$



- 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.

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

- 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} .

- 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).

Components of tf-idf weighting

Term frequency		Document frequency		Normalization	
n (natural)	$tf_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(tf_{t,d})$	t (idf)	$\log \frac{N}{df_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2 + w_2^2 + \dots + w_M^2}}$
a (augmented)	$0.5 + \frac{0.5 \times tf_{t,d}}{\max_i(tf_{t,d})}$	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)	$1/CharLength^\alpha$, $\alpha < 1$
L (log ave)	$\frac{1 + \log(tf_{t,d})}{1 + \log(\text{ave}_{t \in d}(tf_{t,d}))}$				

Best known combination of weighting options

Default: no weighting

- We often use **different weightings** for queries and documents.
- Notation: ddd.qqq

Example: **Inc.ltn**

Document:

Ilogarithmic tf
no df weighting
cosine normalization

Query:

Ilogarithmic tf
t – means idf
no normalization

tf-idf example: Inc.ltn

Query: "best car insurance". Document: "car insurance auto insurance".

word	query					document				product
	tf-raw	tf-wght	df	idf	weight	tf-raw	tf-wght	weight	n'lized	
auto	0	0	5000	2.3	0	1	1	1	0.52	0
best	1	1	50000	1.3	1.3	0	0	0	0	0
car	1	1	10000	2.0	2.0	1	1	1	0.52	1.04
insurance	1	1	1000	3.0	3.0	2	1.3	1.3	0.68	2.04

Key to columns: **tf-raw**: raw (unweighted) term frequency, **tf-wght**: logarithmically weighted term frequency, **df**: document frequency, **idf**: inverse document frequency, **weight**: the final weight of the term in the query or document, **n'lized**: document weights after cosine normalization, **product**: the product of final query weight and final document weight

$$\sqrt{1^2 + 0^2 + 1^2 + 1.3^2} \approx 1.92$$

$$1/1.92 \approx 0.52$$

$$1.3/1.92 \approx 0.68$$

Final similarity score between query and document: $\sum_i w_{qi} \cdot w_{di} = 0 + 0 + 1.04 + 2.04 = 3.08$

Summary: Ranked retrieval in the 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