Crash Course IR – Fundamentals

Sebastian Hofstätter

sebastian.hofstaetter@tuwien.ac.at
/s_hofstaetter

Today

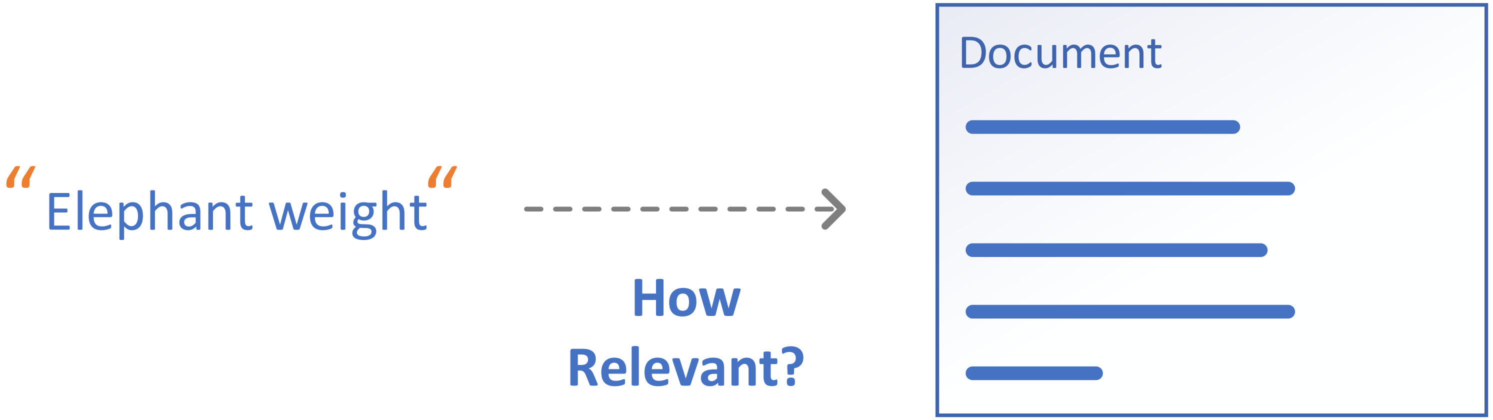
Crash Course IR – Fundamentals

1. **Inverted Index**
   - Creation Process
   - Data structure

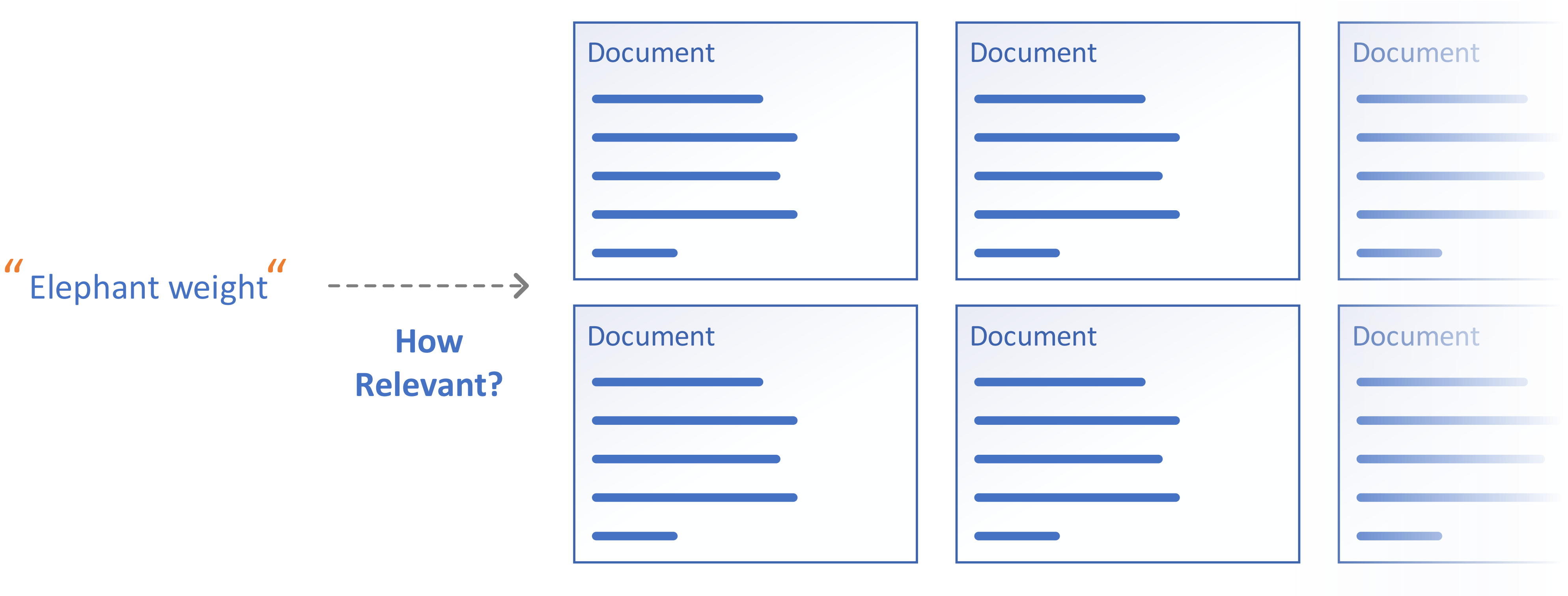
2. **Search & Relevance Scoring**
   - Search Workflow
   - Scoring Models: TF-IDF & BM25

Based on: Lecture 1 - Foundations of IR + Lecture 3 - Relevance & Scoring
https://github.com/sebastian-hofstaetter/teaching/tree/master/introduction-to-information-retrieval/lectures

Information Retrieval



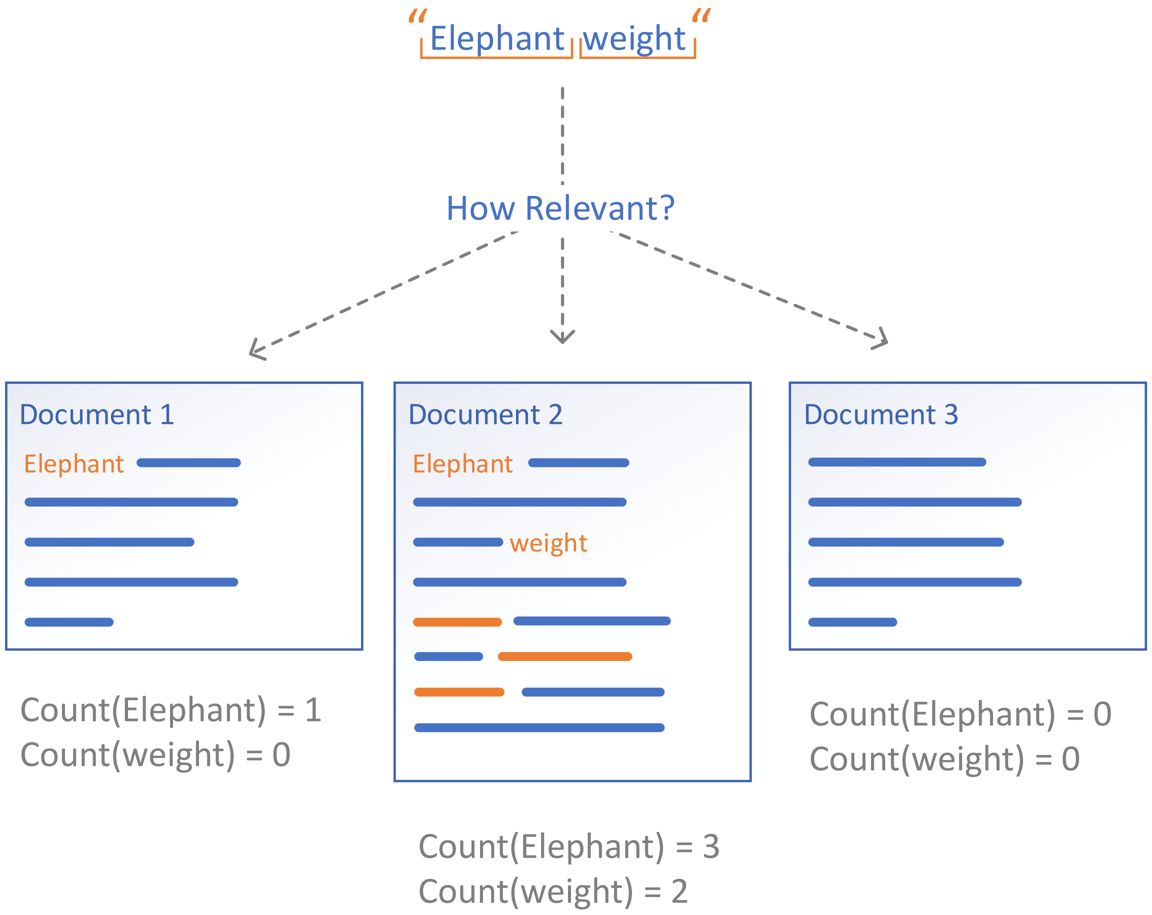
Information Retrieval (Finding the needle in the haystack)



Notes on terminology

- **Documents** can be anything: a web page, word file, text file, article …
  (we assume it to be text for the moment)
  - A lot of details to look out for: encoding, language, hierarchy, fields, …
- **Collection**: A set of documents (we assume it to be static for the moment)
- **Relevance**: Does a document satisfy the information need of the user and does it help complete the user’s task?

Relevance (based on text content)



- If a word appears more often -> more relevant
- Solution: count the words
- If a document is longer, words will tend to appear more often -> take into account the document length
- Counting only when we have a query is inefficient

Inverted Index

Transforming text-based information

Inverted Index

- Inverted index allows to efficiently retrieve documents from large collections
- Inverted index stores all statistics per term (that the scoring model needs)
  - **Document frequency:** how many documents contain the term
  - **Term frequency per document:** how often does the term appear per document
  - Document length
  - Average document length

- Save statistics in a format that is accessible **by a given term**
- Save metadata of a document (Name, location of the full text, etc..)

Inverted Index

Document data

```
Document Ids & Metadata:

[0] = (”Wildlife”, ”location”,...)
[1] = (”Zoo Vienna” ,...)
...

Document Lengths:

[0] = 231 [1] = 381 ...
```

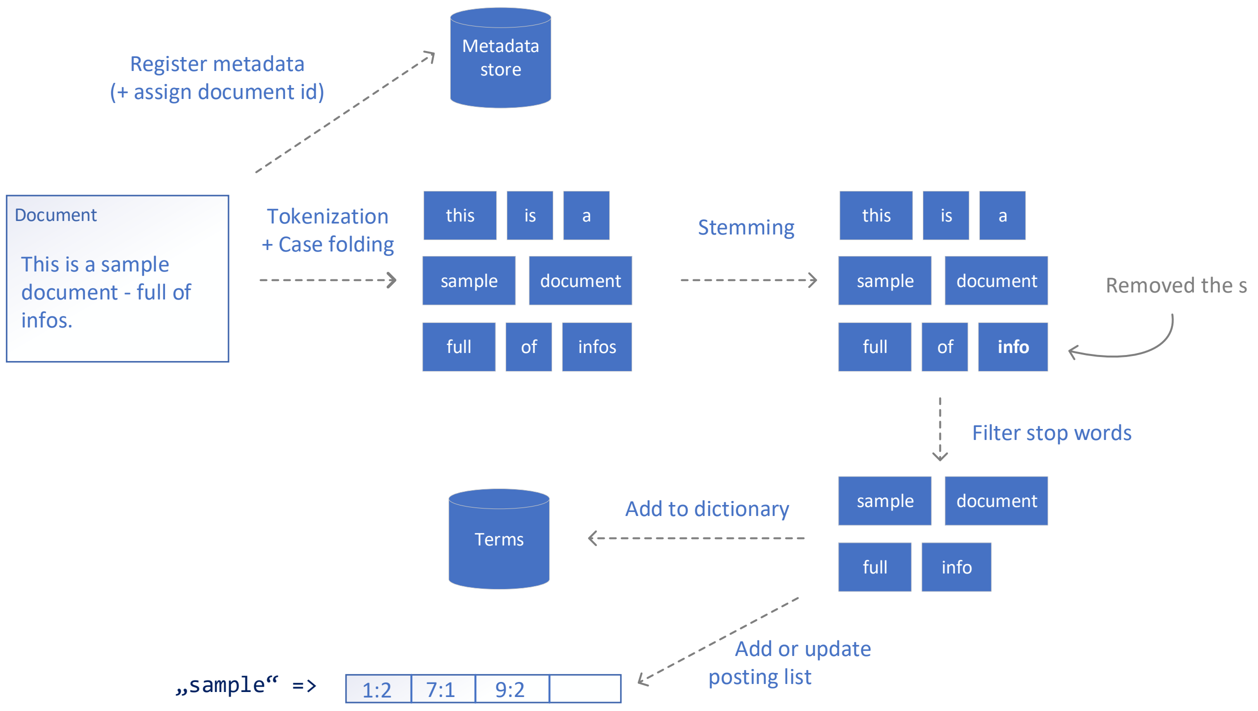
Term data

```
”elephant“ => | 1:5 | 2:1 | 3:5 | 4:5 | … |
”lion“ =>     | 1:2 | 7:1 | 9:2 | … |
”weight“ =>   | 4:1 | 6:4 | … |
...
```

DocId Term Frequency

- Every document gets an internal document id
- Term dictionary is saved as a search friendly data structure (more on that later)
- Term Frequencies are stored in a “posting list” = a list of doc id, frequency pairs

Creating the Inverted Index



- Simplified example pipeline
- Linguistic models are language dependent
- A query text and a document text both have to undergo the same steps

Tokenization

- Naïve baseline: split on each whitespace and punctuation character
  - This splits U.S.A to [U,S,A] or 25.9.2018 to [25,9,2018]
  - Still a good baseline for English

- Improvement: keep abbreviations, names, numbers together as one token
  - Open source tools like Stanford tokenizer https://nlp.stanford.edu/software/tokenizer.shtml
  - Can also handle emoji 👌👍

Stemming

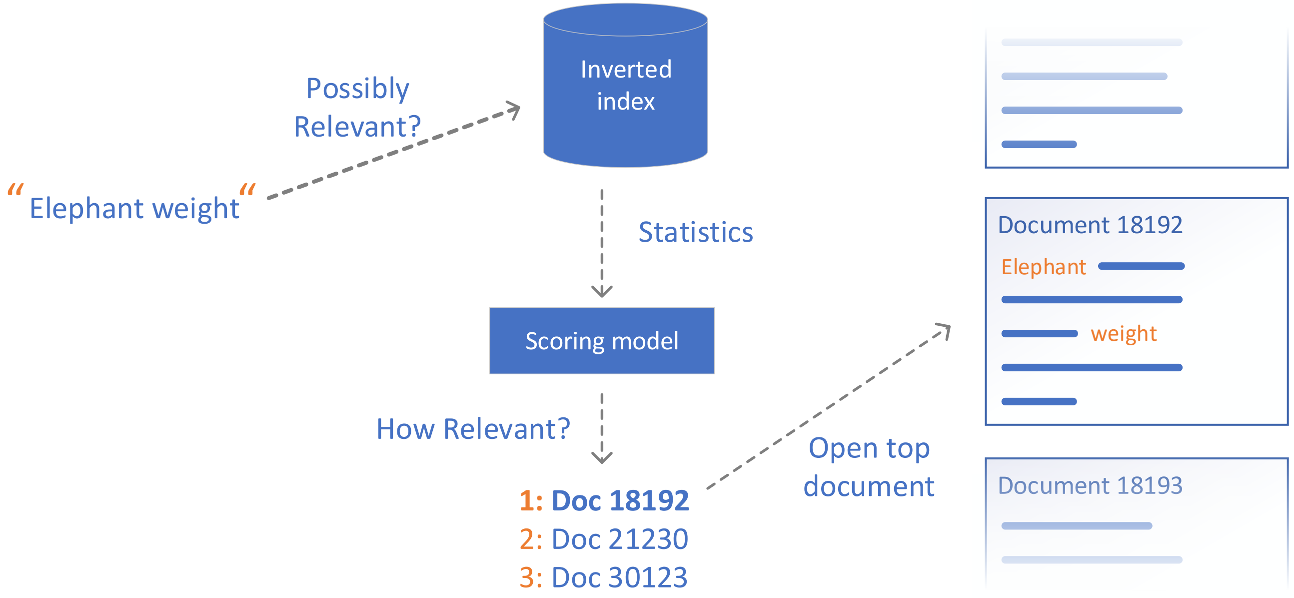
- Reduce terms to their "roots" before indexing
- "Stemming" suggests crude affix chopping
  - language dependent
  - ***automate(s), automatic, automation*** all reduced to ***automat***.

- More advanced form: **Lemmatization**: Reduce inflectional/variant forms to base form (*am, are, is* $\rightarrow$ *be*)
  - Computationally more expensive

Search

Efficiently searching with the Inverted Index

Querying the Inverted Index



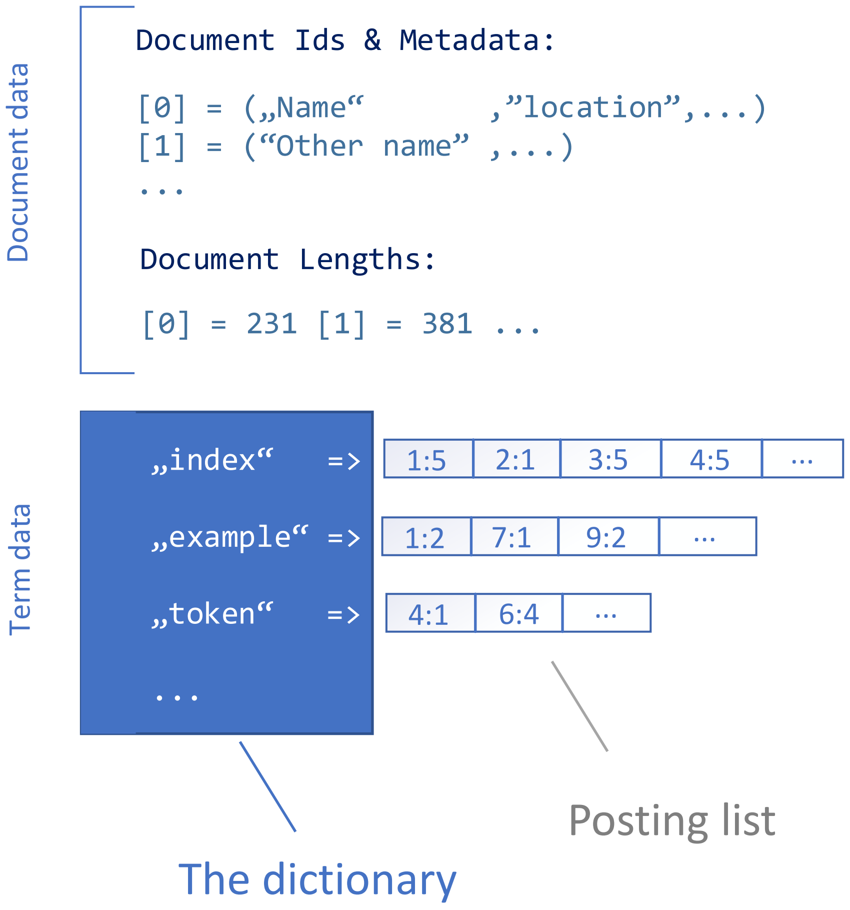
- No need to read full documents
- Only operate on frequency numbers of potentially relevant documents*
- Sort documents based on relevance score – retrieve most relevant documents

* it's not that easy because a document could be relevant without containing the exact query terms – but for now keep it simple

Types of queries (including, but not limited to)

- **Exact matching**: match full words and concatenate multiple query words with “or”
- **Boolean queries**: “and” / “or” / “not” operators between words
- **Expanded queries**: automatically incorporate synonyms and other similar or relevant words into the query
- **Wildcard queries, phrase queries, phonetic queries** (e.g. Soundex) …

Inverted Index: Dictionary



- Dictionary<T> maps text to T
  - T is a posting list or potentially other data about the term depending on the index
- Wanted properties:
  - Random lookup
  - Fast (creation & especially lookup)
  - Memory efficient (keep the complete dictionary in memory)
- Naturally, there are a lot of choices

Dictionary data structures

- **Hash table**: Maps the hash value of a word to a position in a table

- **Trie** (or Prefix Tree): stores alphabet per node and path forms word
- **B-Tree:** Self balancing tree, can have more than two child nodes

- **Finite State Transducer** (FST): Memory friendly automaton

Related:

- **Bloom Filter**: Test if an element is in a set (false positives possible)

Hash table

- Uses a hash function to quickly map a key to a value
  - Collisions possible, have to be dealt with (quite a few options)
- Allows for fast lookup: O(1) (this doesn’t mean it is free!)
- No sorting or sorted sequential access
- Does only a direct mapping
  - No wildcards – no autocomplete

Spell-checking

- Two principal uses
  - Correcting documents being indexed
  - Correcting user queries to retrieve correct answers – e.g. did you mean .. ?
- Two main flavors:
  - Isolated word
    - Check each word on its own for misspelling
    - Will not catch typos resulting in correctly spelled words
    - e.g., ***from*** $\rightarrow$ ***form***
  - Context-sensitive
    - Look at surrounding words,
    - e.g., ***I flew form Heathrow to Narita.***

Spell-checking by Peter Norvig

- Simple isolated spell-checking in a few lines of code
- Uses a text file of ~1 million words (from books)
  - For correct spelling information
  - Probability of each word occurring, if multiple correctly spelled candidates are available
- Get set of candidate words with: deletion or insertion of 1 char, swapping two adjacent chars, replace 1 char with 1 other
- Select most probable correct spelling from available candidates

Details (and implementation in various languages) here: https://norvig.com/spell-correct.html

Relevance

How to select top 10 out of 1 million?

Scoring model

- Input: statistics, Output: floating point value (i.e. the score)

- Evaluated pairwise – 1 query, 1 document: $score(q, d)$

- Capture the notion of relevance in a mathematical model

Today we focus on free-text queries & „ad-hoc“ document retrieval

(document content only)

Search algorithm

float *Scores*={}

for each query term *q*

  fetch posting list for *q*

  for each pair(d, $tf_{t,d}$) in posting list

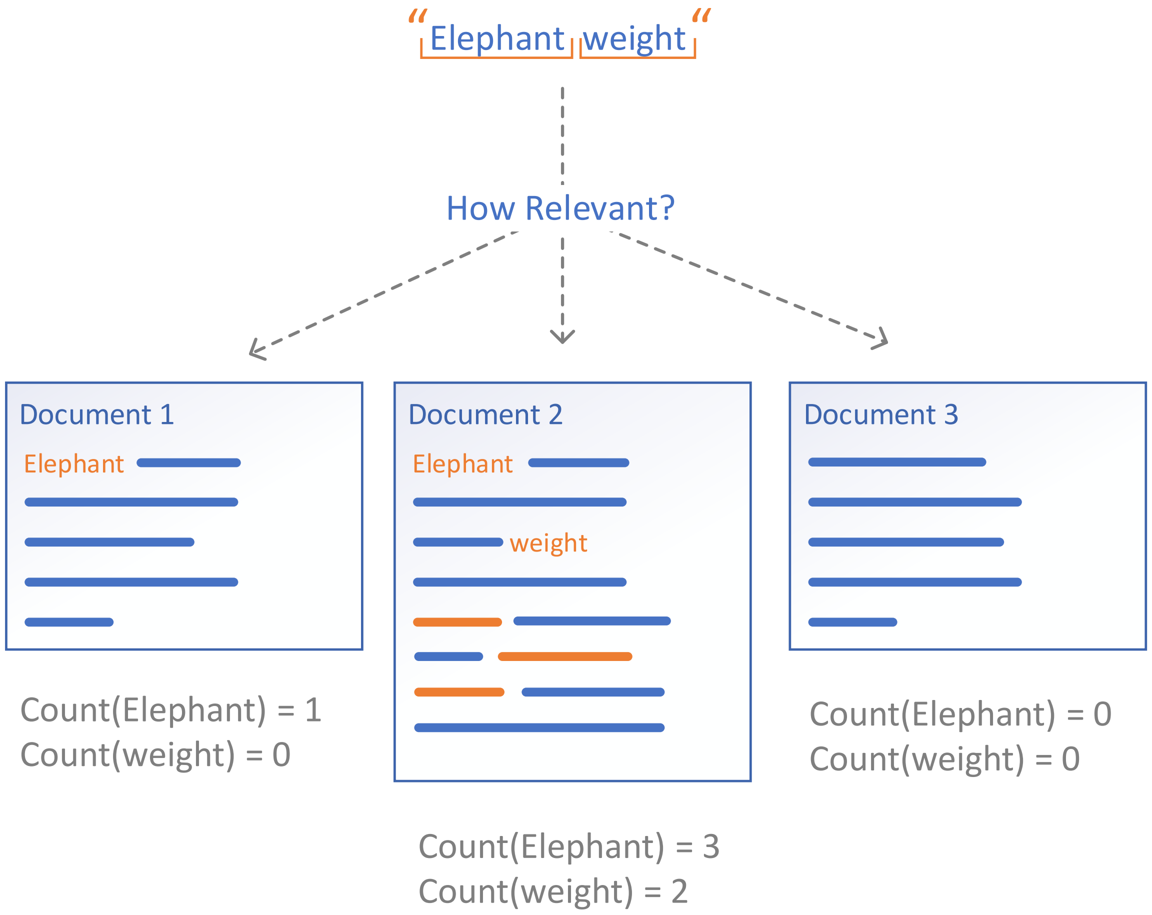
    if *d* not in *Scores* **do** *Scores*[*d*]=0

    Scores[*d*] += score(q, d, $tf_{t,d}$, …)

return Top *K* entries of *Scores*

We transform information back to a document centric view (from the term centric view in the inverted index)

Relevance



- If a word appears more often → more relevant
- Solution: **count the words**
- If a document is longer, words will tend to appear more often → take into account the document length

Relevance

- Words are meaningless – we see them as discrete symbols
- Documents are therefore a stream of meaningless symbols

- We try to find patterns or trends

- Understanding of relevance probably requires deep understanding of language and/or the human brain
  - A step in this direction → using neural networks for relevance computation

Relevance limitations

- “Relevance” means relevance to the need rather than to the query
  - “Query” is shorthand for an instance of information need, its initial verbalized presentation by the user
- Relevance is assumed to be a binary attribute
  - A document is either relevant to a query/need or it is not
- We need these oversimplifications to create & evaluate mathematical models

From: A probabilistic model of information retrieval: development and comparative experiments, Spärck Jones et al. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.134.6108&rep=rep1&type=pdf

TF-IDF

Term Frequency – Inverse Document Frequency

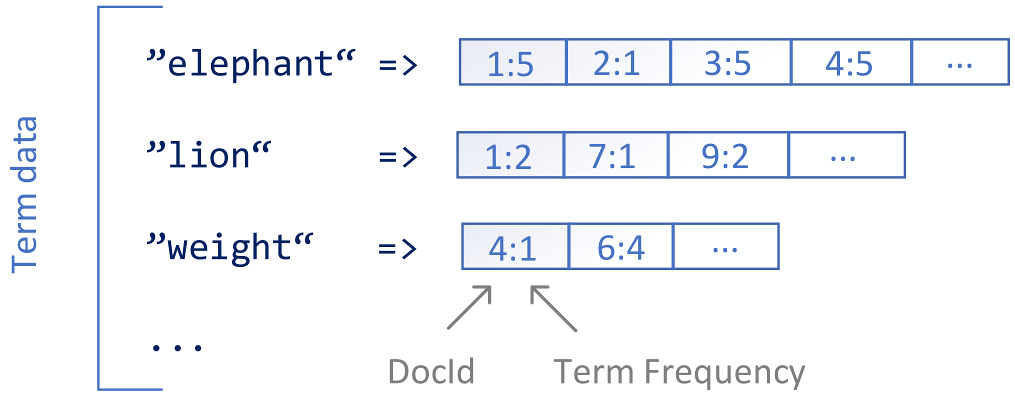
Term Frequency – conceptional data view

- **Bag of words**: word order is not important
- First step for a retrieval model: number of occurrences counts!
- $tf_{t,d}$ number of occurrences of term *t* in document *d*

		Documents					
		Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Terms	Antony	157	73	0	0	0	0
	Brutus	4	157	0	1	0	0
	Caesar	231	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

Term Frequency – actual data storage

- Inverted index saves only non-0 entries, not the whole matrix
  - Otherwise we would waste a lot of storage capacity
- Therefore not good at random lookups into the document column
  - Needs to iterate through the posting list to find the correct document
  - However, for scoring models $tf_{t,d}$ with 0 can be skipped



TF - Term Frequency

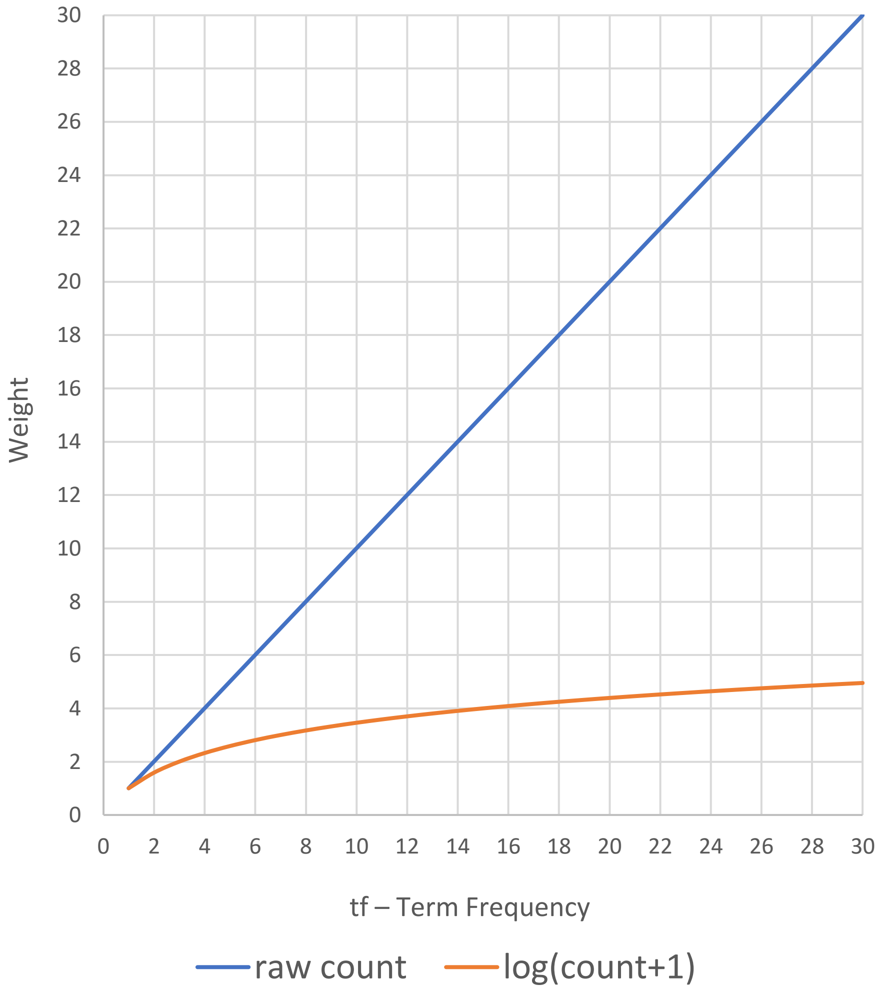
- $tf_{t,d}$ = how often does term t appear in document d

- Powerful starting point for many retrieval models

- Main point of our intuition at the beginning

- Using the raw frequency is not the best solution
  - Use relative frequencies
  - Dampen the values with logarithm

Term Frequency & Logarithm



- In long documents, a term may appear hundred of times.
- Retrieval experiments show that using the logarithm of the number of term occurrences is more effective than raw counts.
- Commonly used approach: apply logarithm

$$\log(1 + tf_{t,d})$$

Document Frequency

- df_t = in how many documents does term t appear in

- 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., *TUWIEN in a news corpora*
- A document containing this term is very likely to be relevant to the query *TUWIEN*

→ We want a high weight for rare terms like *TUWIEN*.

IDF – Inverse Document Frequency

- A common way of defining the inverse document frequency of a term is as follows:

$$\mathrm{idf}(t) = log\,\frac{|D|}{df_t}$$

- df_t is an inverse measure of the "informativeness" of the term
- $df_t \leq |D|$
- Logarithm is used also for *idf* to "dampen" its effect.

$\lvert D\rvert$	Total # of documents
df_t	# of Documents with $tf_{t,d} > 0$

TF-IDF

$$TF\_IDF(q,d) = \sum_{t \in T_d \cap T_q} \underbrace{log(1 + tf_{t,d})}_{\text{increases with the number of occurrences within a document}} * \underbrace{log(\frac{|D|}{df_t})}_{\text{increases with the rarity of the term in the collection}}$$

$\sum_{t \in T_d \cap T_q}$	Sum over all query terms, that are in the index
$tf_{t,d}$	Term frequency
$\|D\|$	Total # of documents
df_t	# of Documents with $tf_{t,d} > 0$

- A rare word (in the collection) appearing a lot in one document creates a high score
- Common words are downgraded

For more variations: https://en.wikipedia.org/wiki/Tf-idf

TF-IDF – Usage

- Useful not only as a standalone model in document retrieval

- Weights used as a base for many other retrieval models
  - Example: Vector Space Model (VSM) works better with tf-idf weights

- Also useful as a generic word weighting mechanism for NLP
  - Task agnostic importance of a word in a document in a collection
  - Assign every word in a collection its tf-idf score
  - Example: Latent Semantic Analysis (LSA) works better with tf-idf weights

BM25

“BestMatch25”

BM25

- Created 1994 by Robertson et al.
- Grounded in probabilistic retrieval

- In general, BM25 improves on TF-IDF results

- But only set as a default scoring in Lucene in 2015

Original paper: http://www.staff.city.ac.uk/~sb317/papers/robertson_walker_sigir94.pdf

TF-IDF vs BM25 in Lucene https://opensourceconnections.com/blog/2015/10/16/bm25-the-next-generation-of-lucene-relevation/

BM25 (as defined by Robertson et al. 2009)

$$BM25(q,d) = \sum_{t \in T_d \cap T_q} \frac{tf_{t,d}}{k_1((1-b) + b\frac{dl_d}{avgdl}) + tf_{t,d}} * log\frac{|D| - df_t + 0.5}{df_t + 0.5}$$

- Assuming we have no additional relevance information
  - If we do: Use RSJ (Robertson/Spärck Jones) weight instead of IDF
- Simpler than the original formula
  - Over time it was shown that more complex parts not needed

Details (a lot of them): The Probabilistic Relevance Framework: BM25 and Beyond
http://www.staff.city.ac.uk/~sb317/papers/foundations_bm25_review.pdf

$\sum_{t \in T_d \cap T_q}$	Sum over all query terms, that are in the index
$tf_{t,d}$	Term frequency
dl_d	Document length
$avgdl$	Average document length in index
$\|D\|$	Total # of documents
df_t	# of Documents with $tf_{t,d} > 0$
k_1, b	Hyperparameters

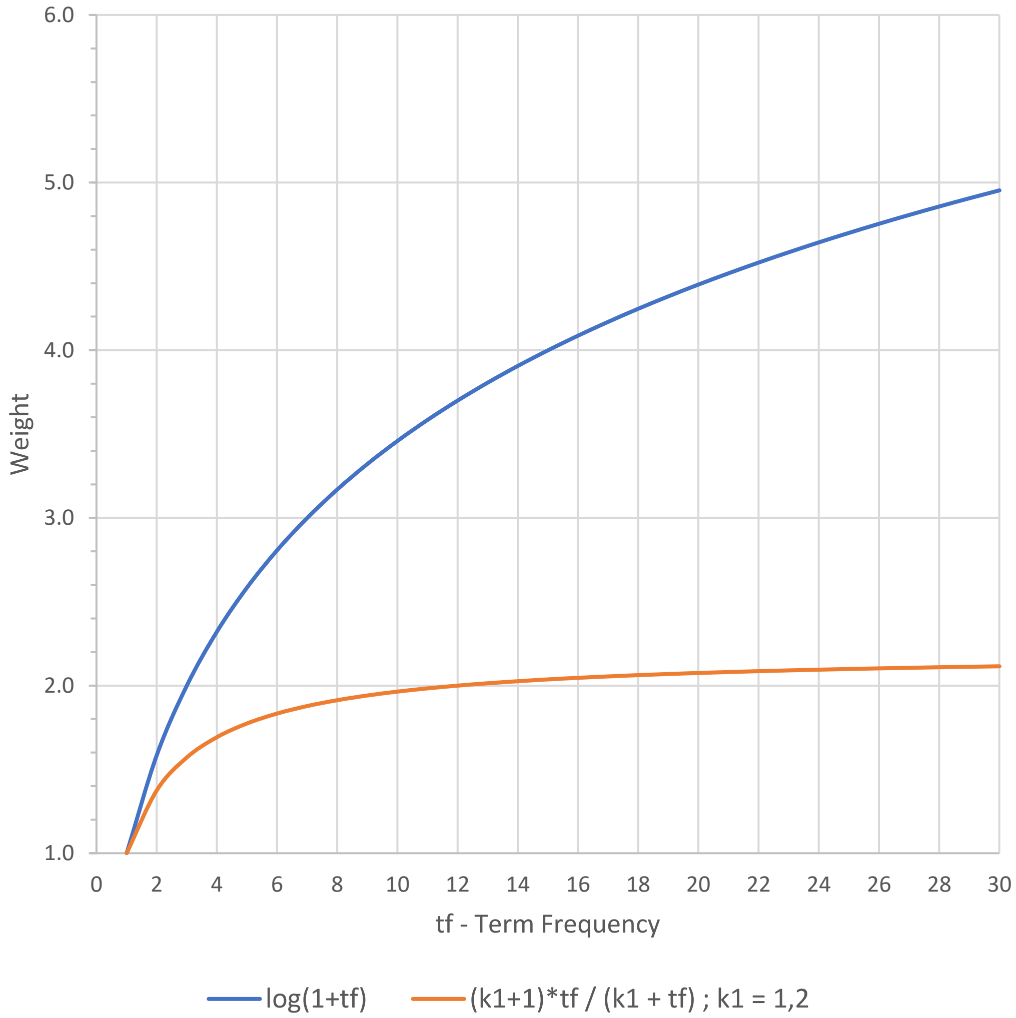
BM25 vs. TF-IDF

- Simple case of BM25 looks a lot like TF-IDF

- 1 main difference: BM25 tf component contains saturation function
  - Therefore works better in practice

- BM25 variants can be adapted to:
  - Incorporate additional reference information
  - Long(er) queries
  - multiple fields

BM25 vs. TF-IDF - Saturation



- **TF-IDF:** weight is always increasing (even with log)
- **BM25:** diminishing returns quickly = asymptotically approaches $k_1 + 1$

Note: we added (k_1+1) to the numerator to make tf@1 = 1, but it does not change the ranking because it is added to every term

Note: we assume the doc length = avgdl

BM25 vs. TF-IDF - Example

- Suppose your query is "machine learning"
- Suppose you have 2 documents with term counts:
  - doc1: learning 1024; machine 1
  - doc2: learning 16; machine 8

- TF-IDF: $\log(\mathrm{tf}) * \log(|\mathrm{D}|/\mathrm{df})$
  - doc1: $11 * 7 + 1 * 10 = \mathbf{87}$
  - doc2: $5 * 7 + 4 * 10 = \mathbf{75}$
- BM25: $\mathrm{k}_1 = 2$
  - doc1: $7 * 3 + 10 * 1 = \mathbf{31}$
  - doc2: $7 * 2.67 + 10 * 2.4 = \mathbf{42.7}$

Example taken from: https://web.stanford.edu/class/cs276/

Hyperparameters

- k_1, b are hyperparameters = they are set by us, the developers

- k_1 controls term frequency scaling
  - k_1 = 0 is binary model; k_1 large is raw term frequency
- b controls document length normalization
  - b = 0 is no length normalization; b = 1 is relative frequency (fully scale by document length)

- Common ranges: $0.5 < b < 0.8$ and $1.2 < k_1 < 2$

BM25F

- BM25 only covers the document as 1 unstructured heap of words
- Real world use case: documents have at least some structure
  - Title, abstract, infobox, headers …
  - Anchor text in web pages describing a page

- BM25**F** allows for multiple fields (or “streams”) in a document
  - For example - 3 streams per doc: title/abstract/body

- BM25**F** allows to assign different weights to the individual streams

BM25F (as defined by Robertson et al. 2009)

$$BM25F(q,d) = \sum_{t \in T_d \cap T_q} \frac{\widetilde{tf}_{t,d}}{k_1 + \widetilde{tf}_{t,d}} * log \frac{|D| - df_t + 0.5}{df_t + 0.5}$$

$$\widetilde{tf}_{t,d} = \sum_{s=1}^{S_d} w_s \frac{tf_{t,s}}{(1 - b_s) + b_s \frac{sl_s}{avgsl})}$$

- Assuming we have no additional relevance information – if we do use RSJ
- Shared IDF might be problematic, could be improved

Details (a lot of them): The Probabilistic Relevance Framework: BM25 and Beyond
http://www.staff.city.ac.uk/~sb317/papers/foundations_bm25_review.pdf

$\sum_{t \in T_d \cap T_q}$	Sum over all query terms, that are in the index
$\sum_{s=1}^{S_d}$	Sum over streams for one doc
$tf_{t,s}$	Term frequency in the stream s
sl_s	Stream length
$avgsl$	Average stream length in index
$\|D\|$	Total # of documents
df_t	# of Documents with $tf_{t,d} > 0$
w_s	Stream weight
k_1, b_s	Hyperparameters

BM25F

- BM25F first combines streams and then terms
  - This is different than chaining together BM25 results on different streams
  - The saturation function is applied at the stream level
- This follows the property that the f.e. the title and body of 1 document are not independent from each other
  - And may not be treated as independent
- Naturally, one assigns a higher stream weight to titles and abstracts
  - Exact values have to be found again with evaluating different settings and relevance judgements

1998: Google

- Started as a research project at Stanford
- Obviously, a lot of good ideas
- Information retrieval as a problem of context and scale

Original paper: (highly recommended reading)

The Anatomy of a Large-Scale Hypertextual Web Search Engine, Brin and Page
http://ilpubs.stanford.edu:8090/361/

Summary: Crash Course – Fundamentals

1. We save statistics about terms in an inverted index

2. The statistics in the index can be access by a given term (query)

3. TF-IDF & BM25 use term and document frequencies to score a query & doc

1. We save statistics about terms in an inverted index
2. The statistics in the index can be access by a given term (query)
3. TF-IDF & BM25 use term and document frequencies to score a query & doc

Thank You