Maschinelle Sprachverarbeitung

Retrieval Models and Implementation

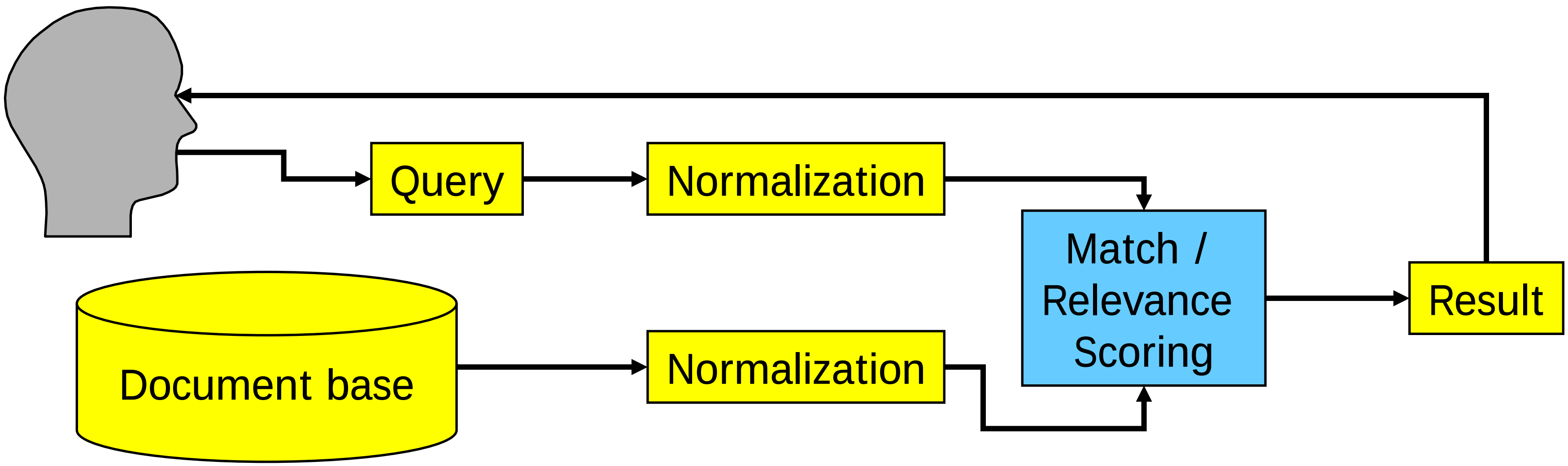
Ulf Leser

Content of this Lecture

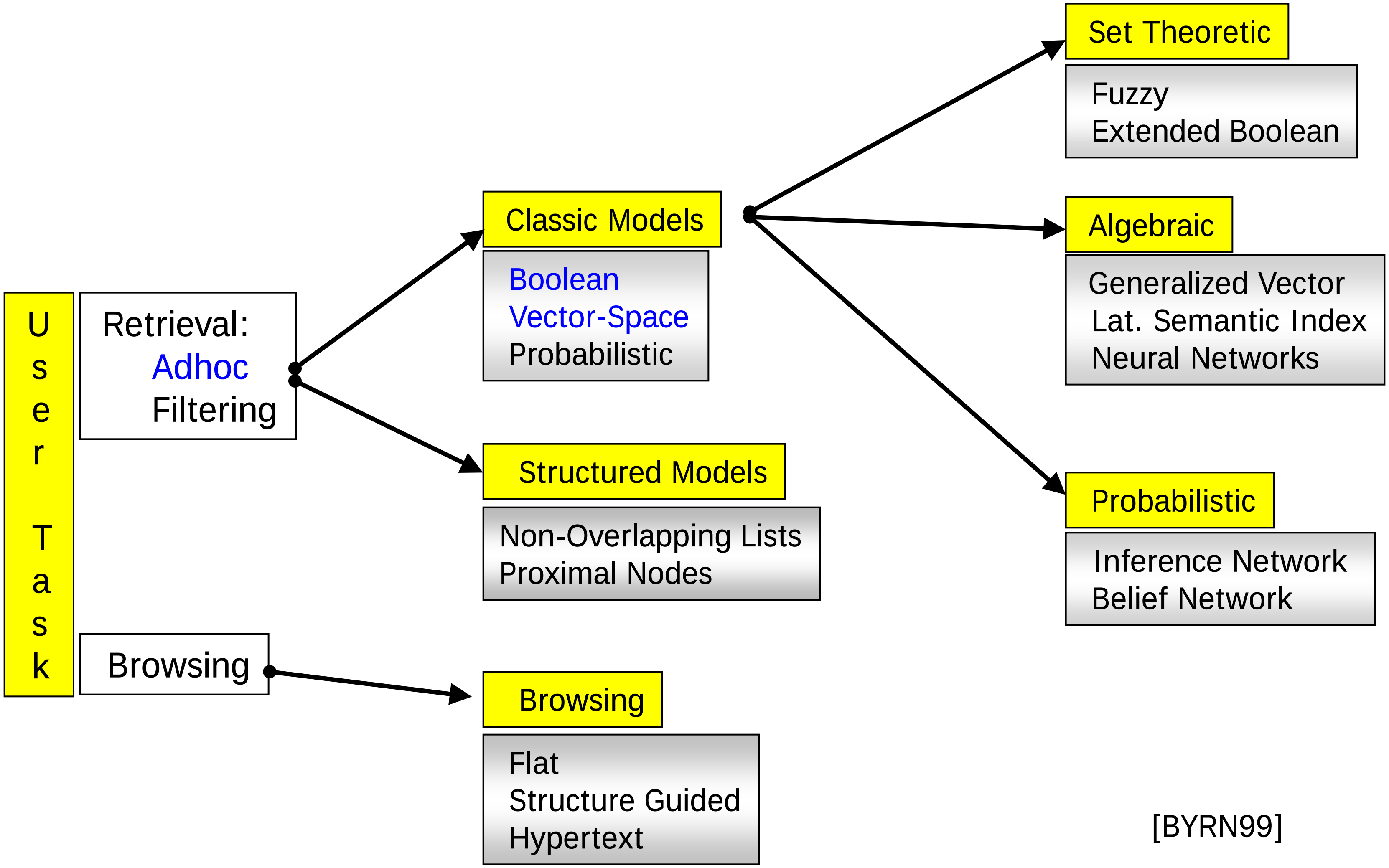
- Information Retrieval Models
  - Boolean Model
  - Vector Space Model
- Inverted Files

Information Retrieval Core

- The core question in IR:
  Which of a given set of (normalized) documents is relevant for a given query?
- Ranking: How relevant for a given query is each document?



How can Relevance be Judged?



[BYRN99]

Notation

- Most of the models we discuss use the "Bag of Words"
- Definition
  - *Let D be the set of all normalized documents, $d \in D$ is a document*
  - *Let K be the set of all terms in D, $k_i \in K$ is a term*
    - *Can as well be tokens*
  - *Let w be the function that maps a given d to its multiset of distinct terms in K (its bag-of-words)*
  - *Let v_d by a vector of size $|K|$ for d with*
    - $v_d[i]=0$ *iff* $k_i \notin w(d)$
    - $v_d[i]=1$ *iff* $k_i \in w(d)$
  - *Often, we use weights instead of a Boolean membership function*
    - *Let $w_{ij} \geq 0$ be the weight of term k_i in document d_j ($w_{ij}=v_j[i]$)*
    - $w_{ij}=0$ *if* $k_i \notin d_j$

Boolean Model

- Simple model based on set theory
- Queries are specified as Boolean expressions over terms
  - Terms connected by AND, OR, NOT, (XOR, ...)
  - Parenthesis are possible (but ignored here)
- Relevance of a document is either 0 or 1
  - Let q contain the atoms (terms) $<k_1, k_2, ...>$
  - An atom k_i evaluates to true for a document d iff $v_d[k_i]=1$
  - Compute truth values of all atoms for each d
  - Compute truth of q for d as logical expression over atom values
- Example: “(kaufen AND rad) OR NOT wir”
  - “wir kaufen ein rad” – <(T AND T) OR NOT T> = T
  - “sei kaufen ein auto” - <(T AND F) OR NOT F> = T

Properties

- Simple, clear semantics, widely used in (early) systems
- Disadvantages
  - No partial matching
    - Suppose query $\mathbf{k_1 \wedge k_2 \wedge \ldots \wedge k_9}$
    - A doc d with $\mathbf{k_1 \wedge k_2 \ldots k_8}$ is as irrelevant as one with none of the terms
  - No ranking
  - Terms cannot be weighted
  - No synonyms, homonyms, semantically close words
  - Lay users don't understand Boolean expressions
- Results: Often unsatisfactory
  - Too many documents (too few restrictions, many OR)
  - Too few documents (too many restrictions, many AND)

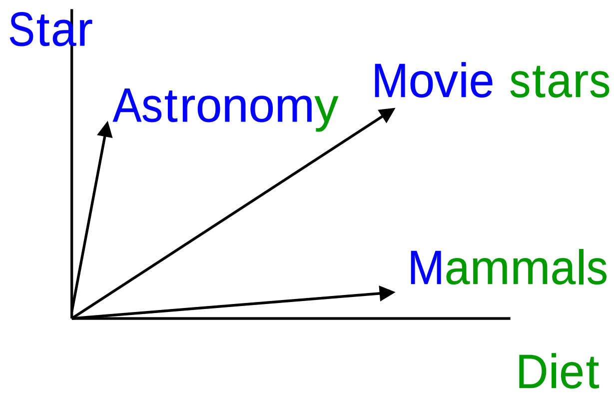
Content of this Lecture

- Information Retrieval Models
  - Boolean Model
  - Vector Space Model
- Inverted Files

Vector Space Model

- Salton, G., Wong, A. and Yang, C. S. (1975). "A Vector Space Model for Automatic Indexing." *Communications of the ACM* **18**(11): 613-620.
  - A breakthrough in IR
- General idea
  - Fix vocabulary K (the dictionary)
  - View each doc (and the query) as point in a |K|-dimensional space
  - Rank docs according to distance from the query in that space
- Main advantages
  - Inherent ranking (according to distance)
  - Naturally supports partial matching (increases distance)

Vector Space



- Each term is one dimension
  - Different suggestions for determining co-ordinates, i.e., term weights
- The closest docs are the most relevant ones
  - Rationale: Vectors correspond to themes which are loosely related to sets of terms
  - Distance between vectors ~ distance between themes
  - Different suggestions for defining distance

The Angle between Two Vectors

- Recall: The scalar product between two vectors v and w of equal dimension is defined as

$$v \circ w = |v| * |w| * \cos(v, w)$$

- This gives us the angle

$$\cos(v, w) = \frac{v \circ w}{|v| * |w|}$$

  - With

$$|v| = \sqrt{\sum v_i^{\,2}} \qquad v \circ w = \sum_{i=1..n} v_i * w_i$$

Distance as Angle

Distance = cosine of the angle between doc d and query q

$$sim(d,q) = \cos(v_d, v_q) = \frac{v_d \circ v_q}{|v_d| * |v_q|} = \frac{\sum (v_q[i] * v_d[i])}{\sqrt{\sum v_d[i]^2} * \sqrt{\sum v_q[i]^2}}$$

Length normalization

Can be dropped for ranking

Example

- Assume stop word removal, stemming, and binary weights

	Text	verkauf	haus	italien	gart	miet	blüh	woll
1	Wir verkaufen Häuser in Italien	1	1	1				
2	Häuser mit Gärten zu vermieten		1		1	1		
3	Häuser: In Italien, um Italien, um Italien herum		1	1				
4	Die italienschen Gärtner sind im Garten			1	1			
5	Der Garten in unserem italienschen Haus blüht		1	1	1		1	
Q	Wir wollen ein Haus mit Garten in Italien mieten		1	1	1	1		1

Ranking

1	1	1	1				
2		1		1	1		
3		1	1				
4			1	1			
5		1	1	1		1	
Q		1	1	1	1		1

$$sim(d,q) = \frac{\sum \left(v_q[i] * v_d[i] \right)}{\sqrt{\sum v_d[i]^2}}$$

- $sim(d_1,q) = (1*0+1*1+1*1+0*1+0*1+0*0+0*1) / \sqrt{3}$ ~ 1.15
- $sim(d_2,q) = (1+1+1) / \sqrt{3}$ ~ 1.73
- $sim(d_3,q) = (1+1) / \sqrt{2}$ ~ 1.41
- $sim(d_4,q) = (1+1) / \sqrt{2}$ ~ 1.41
- $sim(d_5,q) = (1+1+1) / \sqrt{4}$ ~ 1.5

Rg	Q: Wir wollen ein **Haus** mit **Garten** in **Italien mieten**
1	d_2: **Häuser** mit **Gärten** zu **vermieten**
2	d_5: Der **Garten** in unserem **italienschen Haus** blüht
3	d_4: Die **italienschen Gärtner** sind im **Garten**
	d_3: **Häuser**: In **Italien**, um **Italien**, um **Italien** herum
5	d_1: Wir verkaufen **Häuser** in **Italien**

Introducing Term Weights

- Definition
  Let D be a document collection, K be the set of all terms in D, $d \in D$ and $k \in K$
  - *The relative term frequency tf_{dk} is the relative frequency of k in d*
  - *The document frequency df_k is the frequency of docs in D containing k*
    - *This should rather be called "corpus frequency"*
    - *May also be defined as the frequency of occurrences of k in D*
    - *Both definitions are valid and both are used*
  - *The inverse document frequency is defined as $idf_k = |D| / df_k$*
    - *In practice, one usually uses $idf_k = \log(|D| / df_k)$*
  - *The tf*idf score w_{dk} of a term k in a document d is defined as*

$$w_{dk} = \frac{tf_{dk}}{|d|} * idf_k$$

Example TF*IDF

$$sim(d,q) = \frac{\sum \left(v_q[i] * v_d[i]\right)}{\sqrt{\sum v_d[i]^2}}$$

IDF	5	5/4	5/4	5/3	5	5	DIV-0
1 (tf)	1/3	1/3	1/3				
2 (tf)		1/3		1/3	1/3		
3 (tf)		1/4	3/4				
4 (tf)			1/3	2/3			
5 (tf)		1/4	1/4	1/4		1/4	
Q		1	1	1	1		1

- $sim(d_1,q)=(5/4*1/3 +5/4*1/3) / \sqrt{3.13}$ ~ 1.51
- $sim(d_2,q)=(5/4*1/3 +5/3*1/3+5*1/3) / \sqrt{3.26}$ ~ 4,80
- $sim(d_3,q)=(5/4*1/4+5/4*3/4) / \sqrt{0.98}$ ~ 1,57
- $sim(d_4,q)=(5/4*1/3 +5/3*2/3) / \sqrt{1.41}$ ~ 2,08
- $sim(d_5,q)=(5/4*1/4 +5/4*1/4+5/3*1/4) / \sqrt{1.93}$ ~ 2,08

wollen ein **Haus** mit **Garten** in **Italien mieten**
Häuser mit **Gärten** zu **vermieten**
Der **Garten** in unserem **italienschen Haus** blüht Die **italienschen Gärtner** sind im **Garten**
Häuser: In **Italien**, um **Italien**, um **Italien** herum
Wir verkaufen **Häuser** in **Italien**

TF*IDF in Short

- Give terms in a doc d high weights which are ...
  - frequent in d and
  - infrequent in D
- IDF deals with the consequences of Zipf’s law
  - The few very frequent (and unspecific) terms get lower scores
  - The many infrequent (and specific) terms get higher scores
- Interferes with stop word removal
  - If stop words are removed, IDF might not be necessary any more
  - If IDF is used, stop word removal might not be necessary any more

BoW Shortcomings

- No treatment of synonyms (query expansion, ...)
- No treatment of homonyms
  - Different senses = different dimensions
  - We would need to disambiguate terms into their senses (later)
- No consideration of term order
  - But order carries semantic meaning
- Assumes that all terms are independent
  - Clearly wrong: some terms are semantically closer than others
    - Their co-appearance doesn’t mean more than only one appearance
    - The appearance of “red” in a doc with “wine” doesn’t mean much
  - Extension: Topic-based Vector Space Model
    - Latent Semantic Indexing (see IR lecture)

Content of this Lecture

- Information Retrieval Models
  - Boolean Model
  - Vector Space Model
- Inverted Files

Full-Text Indexing

- Fundamental operation for all IR models: find(k, D)
  - Given a query term k, find all docs from D containing it
- Can be implemented using online search
  - Search all occurrence of k in all docs from D
  - Algorithms: Boyer-Moore, Knuth-Morris-Pratt, etc.
- But
  - We generally assume that D is stable (compared to k)
  - We only search for discrete terms (after tokenization)
- Consequence: Better to pre-compute a term index over D
  - Also called “full-text index”

Inverted Files (or Inverted Index)

- Simple and effective index structure for terms
- Builds on the Bag of words approach
  - We give up the order of terms in docs (see positional index later)
  - We cannot reconstruct docs based on index only
- Start from “docs containing terms” (~ “docs”) and invert to “terms appearing in docs” (~ “inverted docs”)

```
d1: t1,t3
d2: t1
d3: t2,t3
d4: t1
d5: t1,t2,t3
d6: t1,t2
d7: t2
d8: t2
```

```
t1: d1,d2,d4,d5,d6
t2: d3,d5,d6,d7,d8
t3: d1,d3,d5
```

Building an Inverted File [Andreas Nürnberger, IR-2007]

Doc1:
Now is the time for all good men to come to the aid of their country.

Doc2:
It was a dark and stormy night in the country manor. The time was past midnight.

Term	Doc ID
now	1
is	1
the	1
time	1
for	1
all	1
good	1
men	1
to	1
come	1
to	1
the	1
aid	1
of	1
their	1
country	1
it	2
was	2
a	2
dark	2
and	2
stormy	2
night	2
in	2
the	2
country	2
manor	2
the	2
time	2
was	2
past	2
midnight	2

Sort

Term	Doc ID
a	2
aid	1
all	1
and	2
come	1
country	1
country	2
dark	2
for	1
good	1
in	2
is	1
it	2
manor	2
men	1
midnight	2
night	2
now	1
of	1
past	2
stormy	2
the	1
the	1
the	2
the	2
their	1
time	1
time	2
to	1
to	1
was	2
was	2

Merge

a	2
aid	1
all	1
and	2
come	1
country	1,2
dark	2
for	1
good	1
in	2
is	1
it	2
manor	2
men	1
midnight	2
night	2
now	1
of	1
past	2
stormy	2
the	1,2
their	1
time	1,2
to	2
was	2

Boolean Retrieval

- For each query term k_i, look-up doc-list D_i containing k_i
- Evaluate query in the usual order
  - $k_i \wedge k_j$: $D_i \cap D_j$
  - $k_i \vee k_j$: $D_i \cup D_j$
  - NOT k_i: $D \setminus D_i$
- Example

a	2
aid	1
all	1
and	2
come	1
country	1,2
dark	2
for	1
good	1
in	2
is	1
it	2
manor	2
men	1
midnight	2
night	2
now	1
of	1
past	2
stormy	2
the	1,2
their	1
time	1,2
to	2
was	2

(time AND past AND the) OR (men)

$$= (D_{time} \cap D_{past} \cap D_{the}) \cup D_{men}$$
$$= (\{1,2\} \cap \{2\} \cap \{1,2\}) \cup \{1\}$$
$$= \{1,2\}$$

Necessary and Obvious Tricks

- How do we support union and intersection efficiently?
  - Naïve algorithm requires $O(|D_i|*|D_j|)$
  - Better: Keep doc-lists sorted
  - Intersection $D_i \cap D_j$: Sort-Merge in $O(|D_i| + |D_j|)$
  - Union $D_i \cup D_j$: Sort-Merge in $O(|D_i| + |D_j|)$
  - If $|D_i| << |D_j|$, use binsearch in D_j for all terms in D_i
    - Whenever $|D_i| + |D_j| > |D_i|*\log(|D_j|)$
- How do we efficiently look-up doc-list D_i?
  - Inverted file is very large – store on disc
  - Bin-search on inverted file: $O(\log(|K|))$ access to disk
  - Inefficient: Random access

Dictionary and Posting List

- Split up inverted file into dictionary and posting list
  - Dictionary is not very large – keep in memory
  - Each entry maintains a pointer to its posting list
  - Posting lists are on disk
  - 1 IO for finding posting list for a given term

Term	docIDs
a	2
aid	1
all	1
and	2
come	1
country	1,2
dark	2
…	…
time	1,2
to	1
was	2

Term
a
aid
all
and
come
country
dark
…
time
to
was

Dictionary

Posting
2
1
1
2
1
1,2
2
…
1,2
1
2

Postings

Adding Term Weighting

- VSM with TF*IDF requires (relative) term frequencies
  - Dictionary stored IDF (or DF) per term
  - For each docID entry in posting list, add (relative) term frequency

Term	DF
a	1
aid	1
all	1
and	1
come	1
country	2
dark	1
…	…
time	2
to	1
was	1

Dictionary

Posting
(2,1)
(1,1)
(1,1)
(2,1)
(1,1)
(1,1), (2,1)
(2,1)
…
(1,1), (2,1)
(1,2)
(2,2)

Postings

Searching in VSM

- Assume we want to retrieve the top-r docs
- Algorithm
  - Initialize an empty doc-list S (as hash table or priority queue)
  - Iterate through query terms k_i
    - Walk through posting list (elements (docID, TF))
      - If docID∈S: S[docID] =+ IDF[k_i]*TF
      - else: S = S.append((docID, IDF[k_i]*TF))
    - Length-normalize value and compute cosine
  - Return top-r docs in S
- S contains all and only those docs containing at least one k_i

Space Usage

- Size of dictionary: O(|K|)
  - Zipf's law: From a certain corpus size on, new terms appear only very infrequently
    - But there are always new terms, no matter how large D
    - Example: 1GB text (TREC-2) generates only 5MB dictionary
  - Typically: <1 Million
    - Many more in multi-lingual corpora, web corpora, etc.
- Size of posting list
  - Theoretic worst case: O(|K|*|D|)
  - Practical: A few hundred entries for each doc in D

Storing the Dictionary

- Dictionary as array (keyword, DF, ptr)
- Since keywords have different lengths: Implementation will be (ptr1, DF, ptr2)
  - ptr1: To string (the keyword)
  - ptr2: To posting list
- Search: Compute log(|K|) memory addresses, follow ptr1, compare strings: $O(\log(|K|)*|k|)$
- Construction: Essentially for free
- Alternatives: Hashing, Keyword Trees

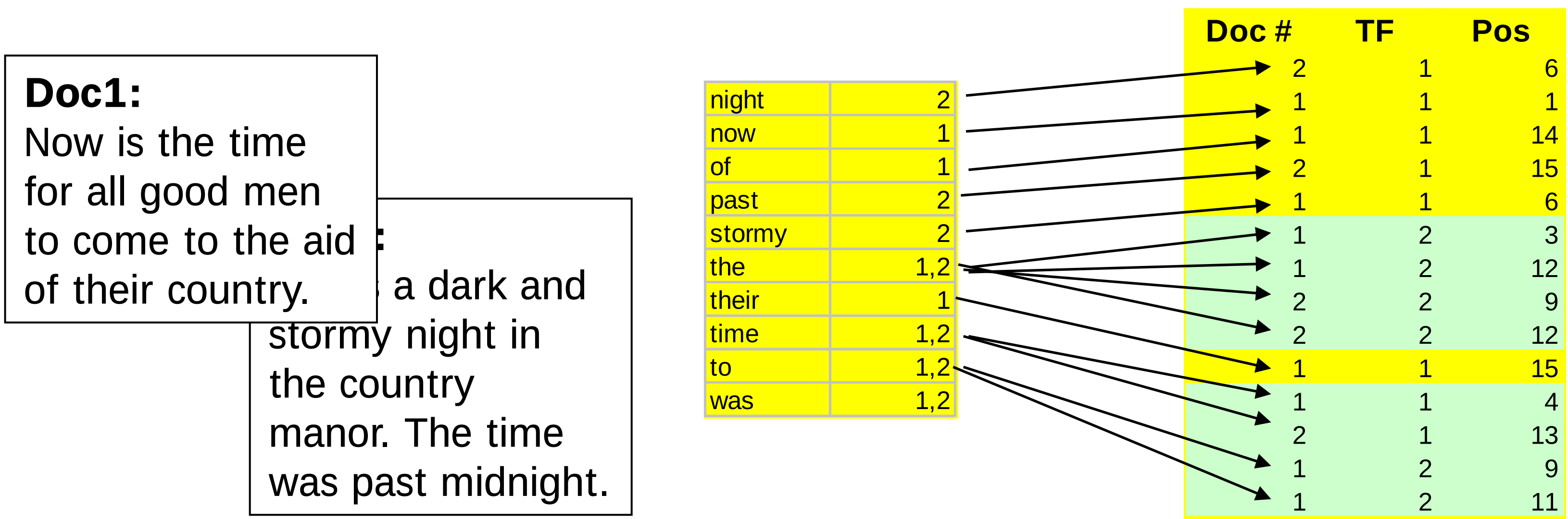
Term	DF	
a	1	ptr
aid	1	ptr
all	1	ptr
and	1	ptr
come	1	ptr
country	2	ptr
dark	1	ptr
for	1	ptr
good	1	ptr
in	1	ptr
is	1	ptr
it	1	ptr
manor	1	ptr
men	1	ptr
midnight	1	ptr
night	1	ptr
now	1	ptr

Storing the Posting File

- Posting file is usually kept on disk
- Thus, we need an IO-optimized data structure
- Static
  - Store posting lists one after the other in large file
  - Posting-ptr is (large) offset in this file
- Prepare for inserts
  - Reserve additional space per posting
    - Good idea: Large initial posting lists get large extra space
    - Many inserts can be handled internally
  - Upon overflow, append entire posting list at the end of the file
    - Place pointer at old position – at most two access per posting list
  - Can lead to many holes – requires regular reorganization

Positional Information

- What if we search for phrases: “Bill Clinton”, “Ulf Leser”
  - ~10% of web searches are phrase queries
- What if we search by proximity “car AND rent/5”
  - “We rent cars”, “cars for rent”, “special care rent”, “if you want to rent a car, click here”, “Cars and motorcycles for rent”, ...
- We need positional information



Answering Phrase Queries

- Search posting lists of all query terms
- During intersection, also positions must fit

Effects

- Dictionary is not affected
- Posting lists get much larger
  - Store many <<docID, pos>,TF> instead of few <docID,TF>
  - Index with positional information typically 30-50% larger than the corpus itself
  - Especially frequent words require excessive storage
- Use compression

Self Assessment

- Explain the vector space model
- How is the size of K (vocabulary) influenced by pre-processing?
- Describe some variations of deducing term weights
- How could we extend the VSM to also consider the order of terms (to a certain degree)?
- Explain idea and structure of inverted files?
- What are possible data structures for the dictionary? Advantages / disadvantages?
- What decisions influence the size of posting lists?