DD2476: Search Engines and Information Retrieval Systems

Johan Boye*
KTH

Lecture 6

* Many slides inspired by Manning, Raghavan and Schütze

Improving recall in search

- Relevance feedback (assignment 3.1)
  - "Give me more of this (and less of that)"
- Wildcard queries (3.3, 3.4)
  - "*colo*rful*" $\rightarrow$ "*colorful*", "*colourful*"
- Spelling correction (3.5, 3.6)
  - "*see you on the wki*" $\rightarrow$ "*see you on the wiki*"
- Query expansion
  - adding synonyms, etc. to the query
  - word vectors, multi-lingual retrieval

Relevance feedback

Relevance feedback: Basic idea

- The user issues a (short, simple) query.
- The search engine returns a set of documents.
- User marks some docs as relevant, some as nonrelevant.
- Search engine computes a new query that better (hopefully) represents the information need.
- Search engine runs new query and returns new results.
- New results have (hopefully) better recall.

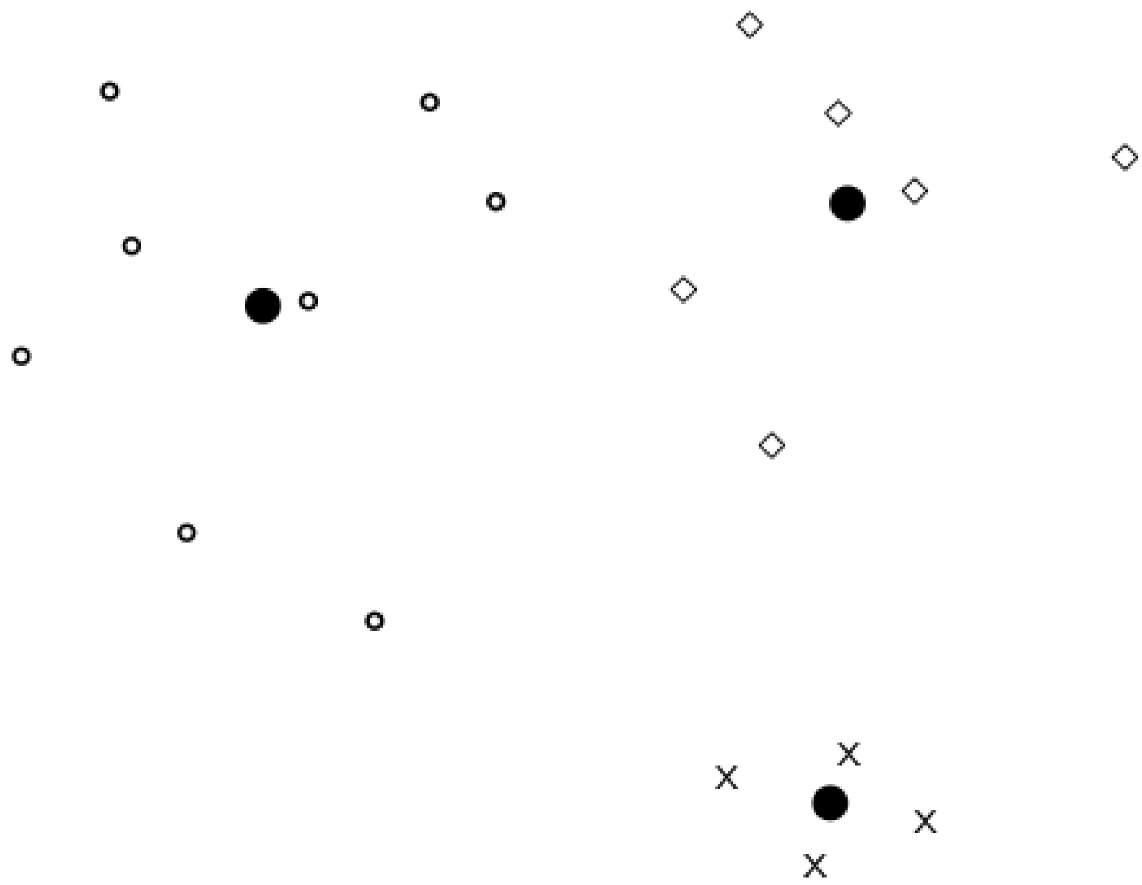
Key concept for relevance feedback: Centroid

- The centroid is the center of mass of a set of points.
- Recall that we represent documents as points in a high-dimensional space.
- Thus: we can compute centroids of documents.
- Definition:

$$\vec{\mu}(D) = \frac{1}{|D|} \sum_{d \in D} \vec{v}(d)$$

- where D is a set of documents, and $\vec{v}(d) = \vec{d}$ is the vector we use to represent document d.

Key concept for relevance feedback: Centroid



Rocchio algorithm

- The Rocchio algorithm implements relevance feedback in the vector space model.
- Rocchio chooses the query $\vec{q}_{opt}$ that maximizes

$$\vec{q}_{opt} = \arg\max_{\vec{q}} [\text{sim}(\vec{q}, \mu(D_r)) - \text{sim}(\vec{q}, \mu(D_{nr}))]$$

- Dr : set of relevant docs; Dnr : set of nonrelevant docs
- Intent: $\vec{q}_{opt}$ is the vector that separates relevant and nonrelevant docs maximally.
- We can rewrite this as:

$$\vec{q}_{opt} = \mu(D_r) + [\mu(D_r) - \mu(D_{nr})]$$

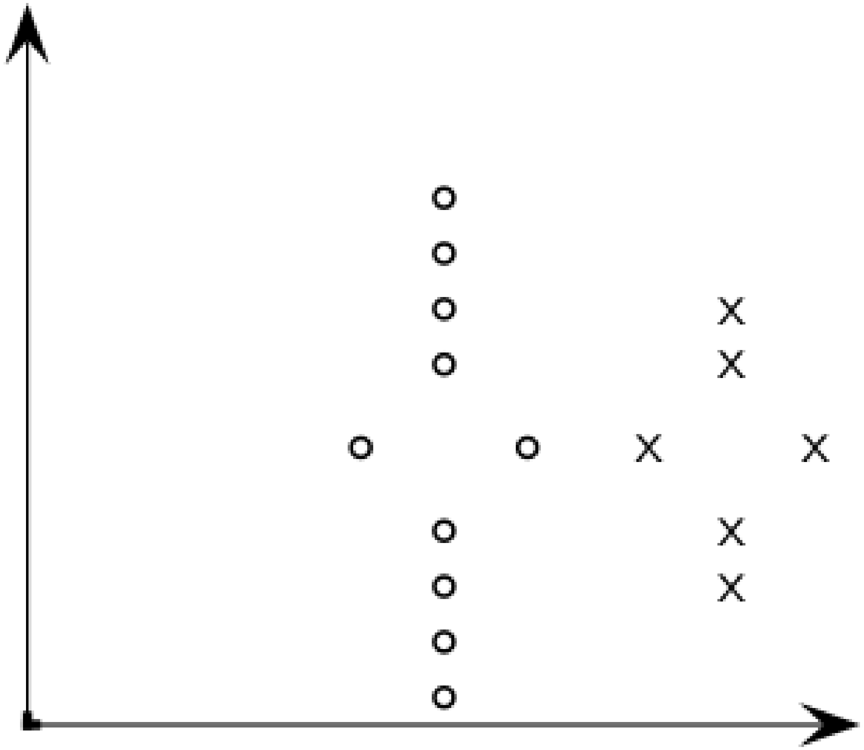
Rocchio algorithm

- The optimal query vector is:

$$
\begin{aligned}
\vec{q}_{opt} &= \mu(D_r) + [\mu(D_r) - \mu(D_{nr})] \\
&= \frac{1}{|D_r|} \sum_{\vec{d}_j \in D_r} \vec{d}_j + [\frac{1}{|D_r|} \sum_{\vec{d}_j \in D_r} \vec{d}_j - \frac{1}{|D_{nr}|} \sum_{\vec{d}_j \in D_{nr}} \vec{d}_j]
\end{aligned}
$$

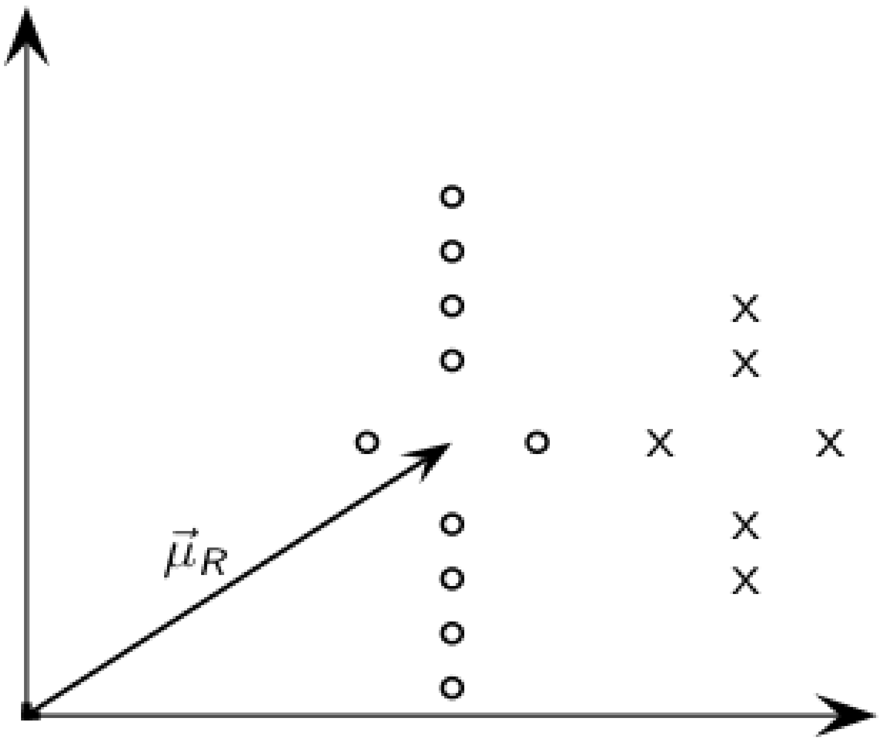
- We move the centroid of the relevant documents by the difference between the two centroids.

Example: Rocchio algorithm



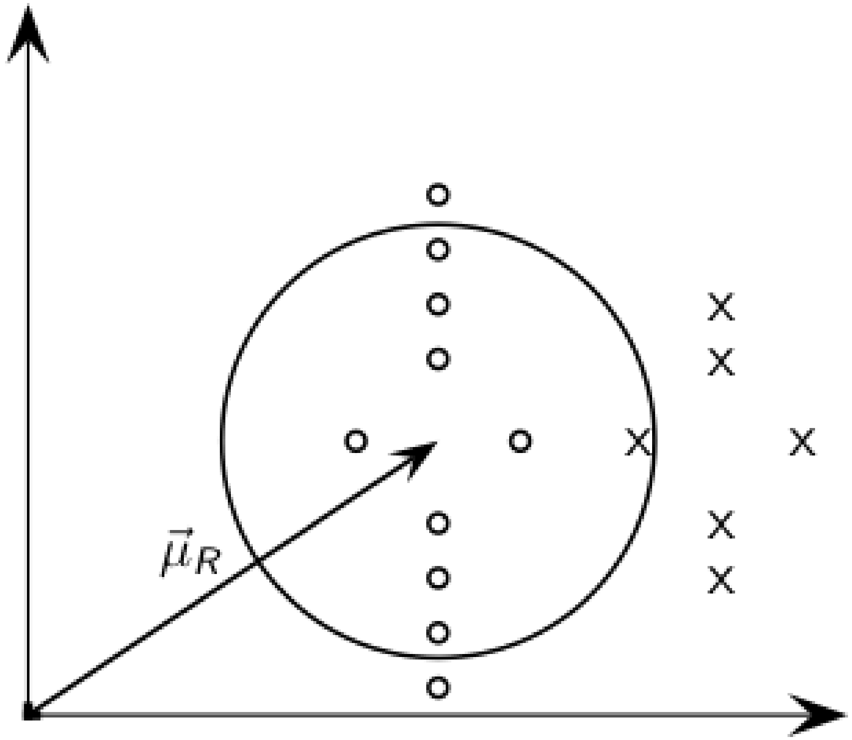
circles: relevant documents, Xs: nonrelevant documents

Rocchio illustrated



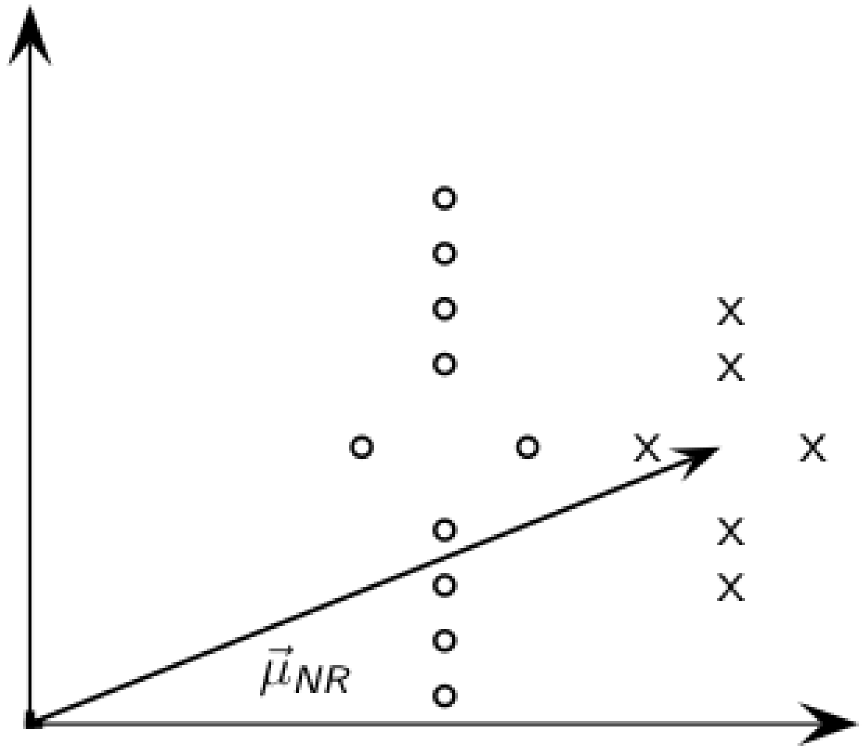
$\vec{\mu}_R$: centroid of relevant documents

Rocchio illustrated



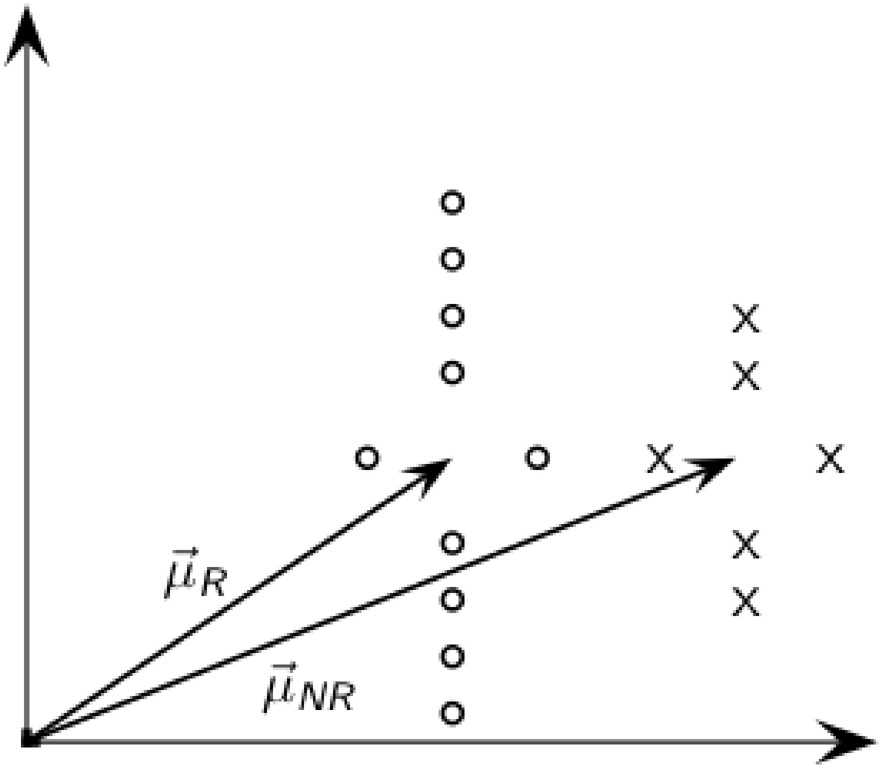
$\vec{\mu}_R$ does not separate relevant / nonrelevant.

Rocchio illustrated

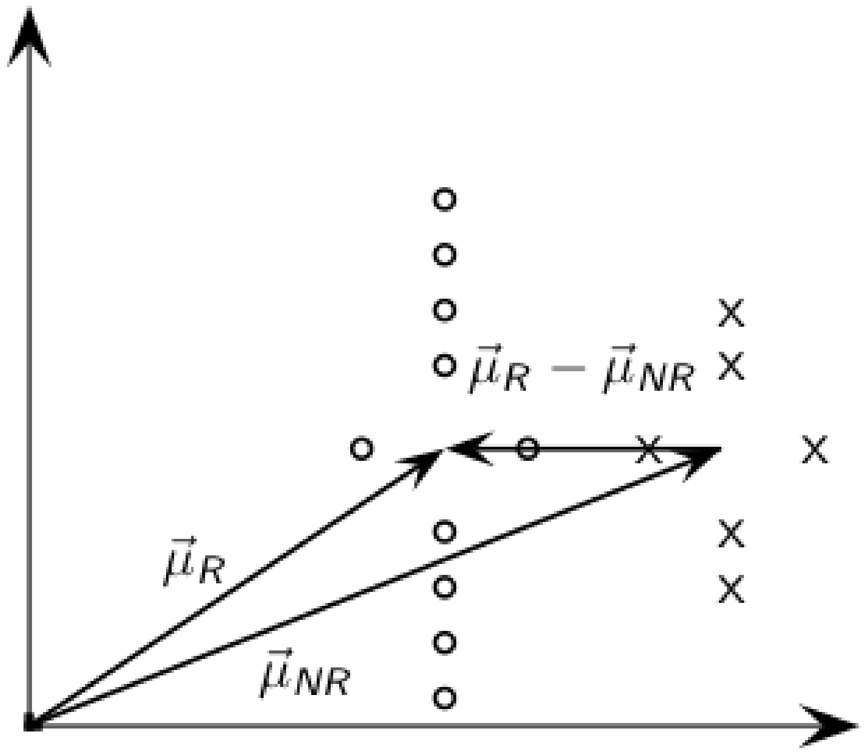


$\vec{\mu}_R$ centroid of nonrelevant documents.

Rocchio illustrated

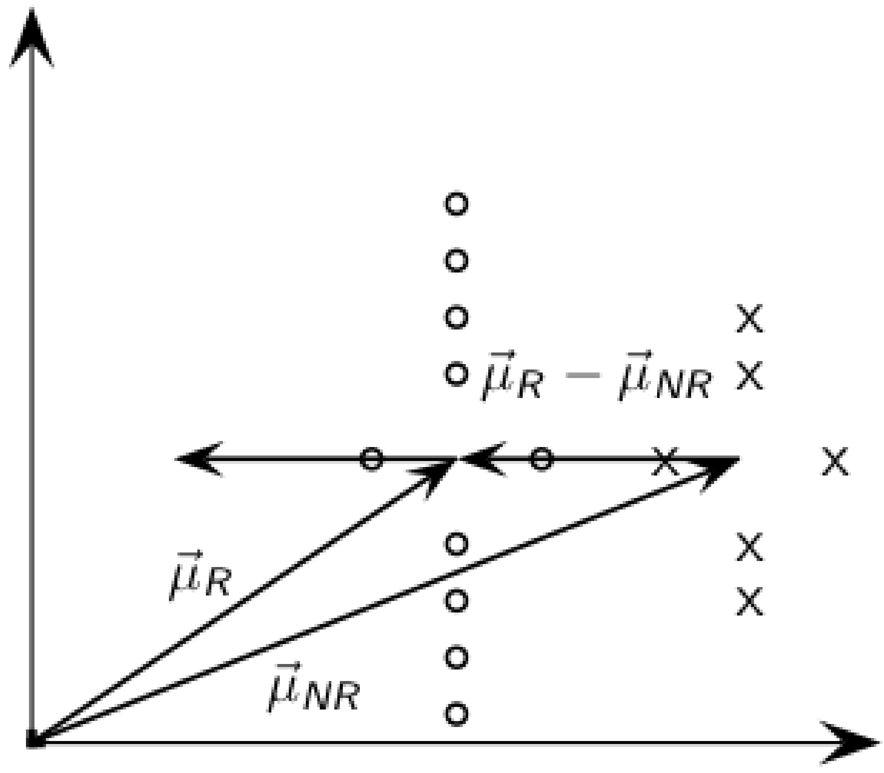


Rocchio illustrated



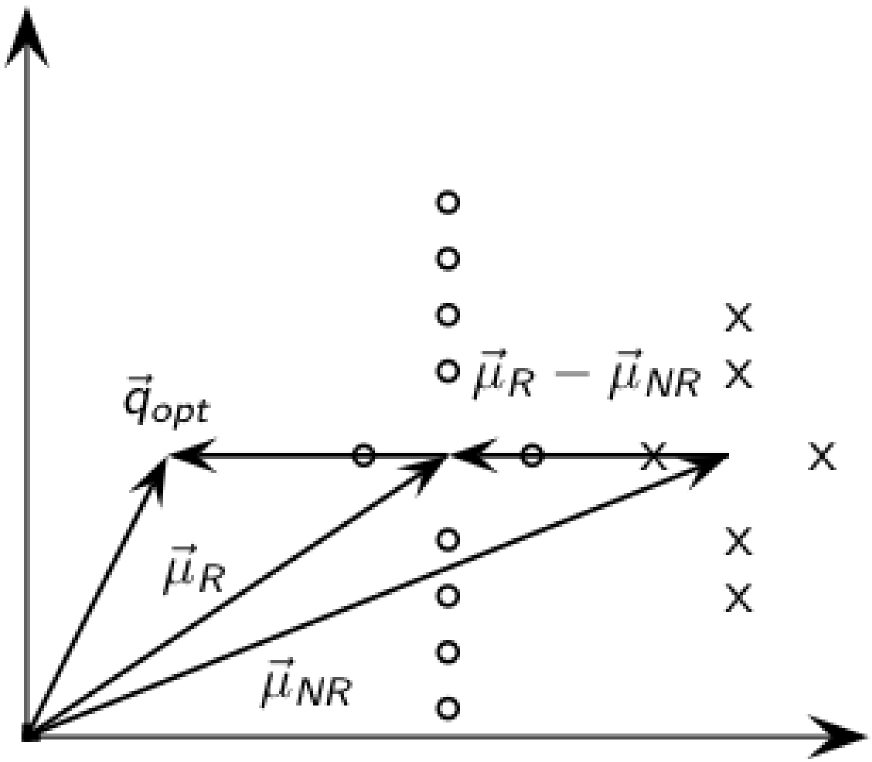
$\vec{\mu}_R - \vec{\mu}_{NR}$: difference vector

Rocchio illustrated



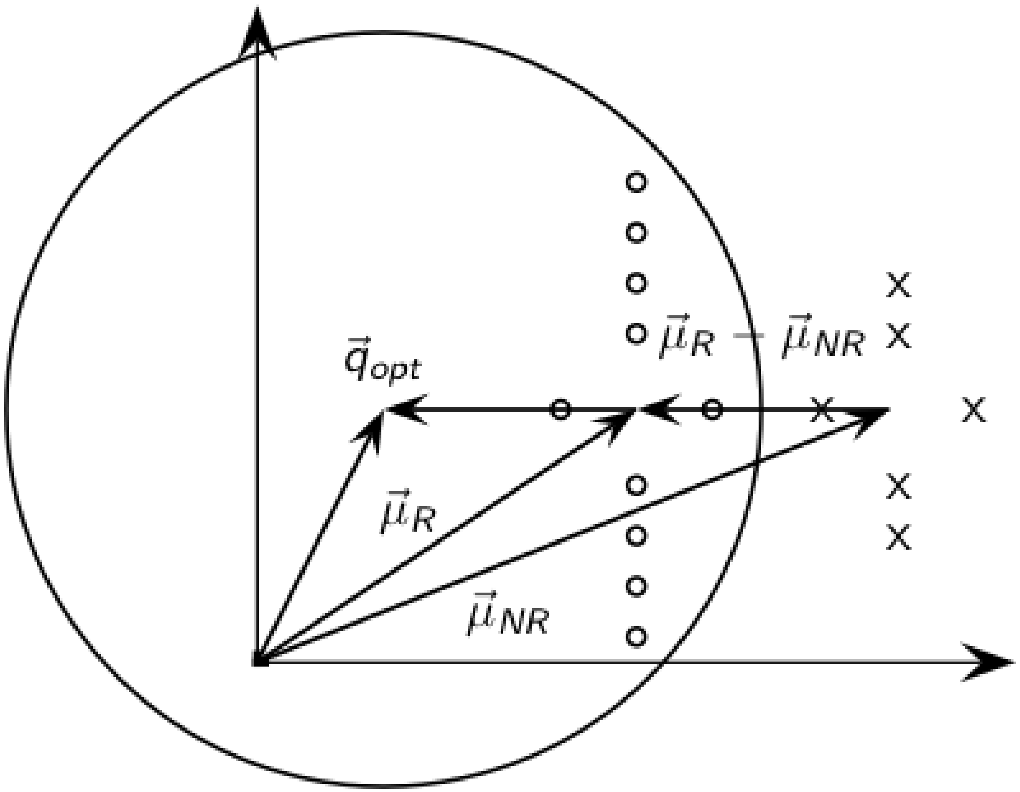
Add difference vector to $\vec{\mu}_R$ …

Rocchio illustrated



… to get $\vec{q}_{opt}$

Rocchio illustrated



$\vec{q}_{opt}$ separates relevant / nonrelevant perfectly.

Quiz

Four documents:

1. `cat` ✓

2. `cat dog` ✓

3. `cat horse horse` ✗

4. `horse` ✗

I find **1,2 relevant**, and **3,4 non-relevant**. What is the optimal query according to Rocchio?

Quiz

1	✓	`cat`	`(1,0,0)`
2	✓	`cat dog`	`(1,1,0)`
3	✗	`cat horse horse`	`(1,0,2)`
4	✗	`horse`	`(0,0,1)`

Relevant centroid R = (1, 0.5, 0)

Non-relevant centroid NR = (0.5, 0, 1.5)

Quiz

1	✓	`cat`	`(1,0,0)`
2	✓	`cat dog`	`(1,1,0)`
3	✗	`cat horse horse`	`(1,0,2)`
4	✗	`horse`	`(0,0,1)`

Relevant centroid R = (1, 0.5, 0)

Non-relevant centroid NR = (0.5, 0, 1.5)

Q = R + R – NR = (1.5, 1, -1.5)

Quiz

1	✓	`cat`	`(1,0,0)`
2	✓	`cat dog`	`(1,1,0)`
3	✗	`cat horse horse`	`(1,0,2)`
4	✗	`horse`	`(0,0,1)`

Q = R + R – NR = (1.5, 1, -1.5)

$$\cos(Q,1) = \frac{1.5\cdot 1 + 1\cdot 0 - 1.5\cdot 0}{\sqrt{5.5}\cdot\sqrt{1}} = 0.64$$

$$\cos(Q,3) = \frac{1.5\cdot 1 + 1\cdot 0 - 1.5\cdot 2}{\sqrt{5.5}\cdot\sqrt{5}} = -0.29$$

Rocchio 1971 algorithm (SMART)

Used in practice:

$$
\begin{aligned}
\vec{q}_m &= \alpha\vec{q}_0 + \beta\mu(D_r) - \gamma\mu(D_{nr}) \\
&= \alpha\vec{q}_0 + \beta\frac{1}{|D_r|}\sum_{\vec{d}_j\in D_r}\vec{d}_j - \gamma\frac{1}{|D_{nr}|}\sum_{\vec{d}_j\in D_{nr}}\vec{d}_j
\end{aligned}
$$

- q_m: modified query vector; q_0: original query vector; D_r and D_{nr} : sets of **known** relevant and nonrelevant documents respectively; α, β, and γ: weights
- New query moves towards relevant documents and away from nonrelevant documents.

Positive vs negative feedback

- Positive feedback is more valuable than negative feedback.
  - For example, set $\beta = 0.75$, $\gamma = 0.25$ to give higher weight to positive feedback.
- Many systems only allow positive feedback.
  - Why?
  - This is what we will do in assignment 3.

Relevance feedback - Assumption

- Relevance documents are similar
  - Term distribution in relevant documents will be similar
  - Term distribution in non-relevant documents will be different from those in relevant documents
  - Similarities between relevant and irrelevant documents are small

Violation of the assumption

- There are several clusters of relevant documents
- Examples:
  - Alternative terminology (Burma / Myanmar)
  - Disjunctive queries (“*Celebrities that use to work for Burger King*”)
  - Instances of general concepts(Feline $\rightarrow$ cat, tiger, lion, etc)

Relevance feedback: Problems

- Long queries are inefficient for typical IR engine.
  - Possible solution: Only reweight certain prominent terms
  - Perhaps top 20 by term frequency
- Users are often reluctant to provide explicit feedback
- It’s often harder to understand why a particular document was retrieved after applying relevance feedback

Pseudo-relevance feedback

- Users are often reluctant to provide explicit feedback
- Pseudo-relevance feedback automates the “manual” part of true relevance feedback.
- Pseudo-relevance algorithm:
  - Retrieve a ranked list of hits for the user’s query
  - Assume that the top k documents are relevant.
  - Do relevance feedback (e.g., Rocchio)
- Works very well on average
  - But can go horribly wrong for some queries.
  - Several iterations can cause **query drift**.

Wildcard queries

Tolerant retrieval

- Spelling correction
  - *“see you on the wki”* $\rightarrow$ *“see you on the wiki”*
- Wildcard queries
  - *“colo*rful”* $\rightarrow$ *“colorful”*, *“colourful”*
- In both cases, the search engine needs to
  - construct the intended query (or queries)
  - compute the results for those queries (intersection, phrase, ranked retrieval)
  - list the results

Wildcard queries: one word

- **`care*:`** find all docs containing any word beginning "care".
- **`*less:`** find words ending in "less"
- **`colo*r:`** find all words beginning "colo" and ending in "r"
- general case: any numbers of '**`*`**' placed anywhere in the word (we will **not** consider this case)
- special case: '**`*`**' matches all words (you don't need to consider this case)

Wildcard queries: several words

- **`b* colo*r:`** find all docs containing any word beginning "b", **and** any word beginning with "colo" ending in "r"

Wildcard queries

- How do we find all words matching **`care*`** ?
- **Idea:** Go through all words in the vocabulary, and check which words match the regular expression **`^care.*`**
  - e.g. using Java's regex library
- Would this work?

K-gram index

- For both wildcard queries and spelling correction we must **quickly** find words that
  - the user intended (for wildcard queries), or
  - the user probably intended (for spelling correction)

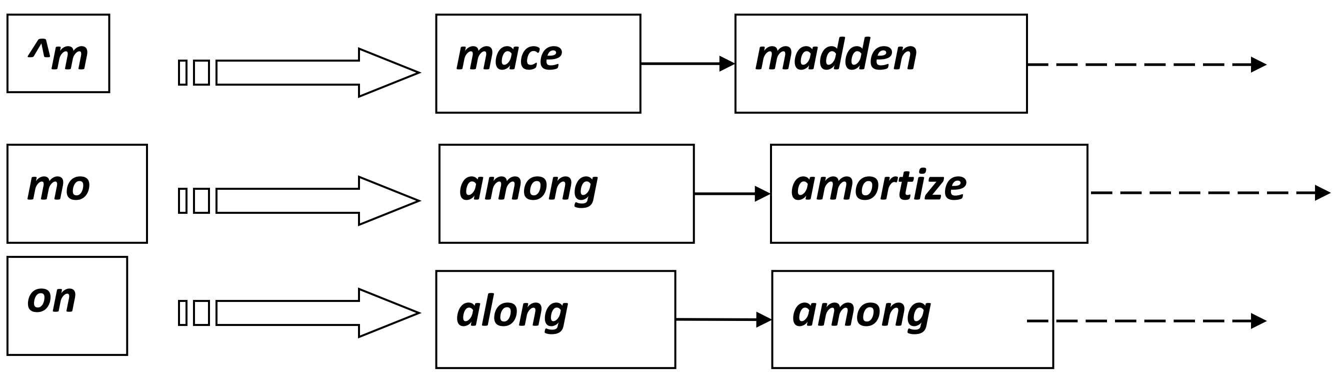
- A ***k*-gram index** is an index from k-grams (parts of words) to words.
  - **bigram** index when k=2
  - **trigram** index when k=3
  - etc.

K-grams

- The bigrams of **`december`**:
  - First add start and end symbol: **`^december$`**
  - Bigrams are all two-letter sequences:
    `^d, de, ec, ce, em, mb, be, er, r$`
- The trigrams of **`december`**:
  - **`^de, dec, ece, cem, emb, mbe, ber, er$`**
- A word of length *n* has *n+3–k* *k*-grams

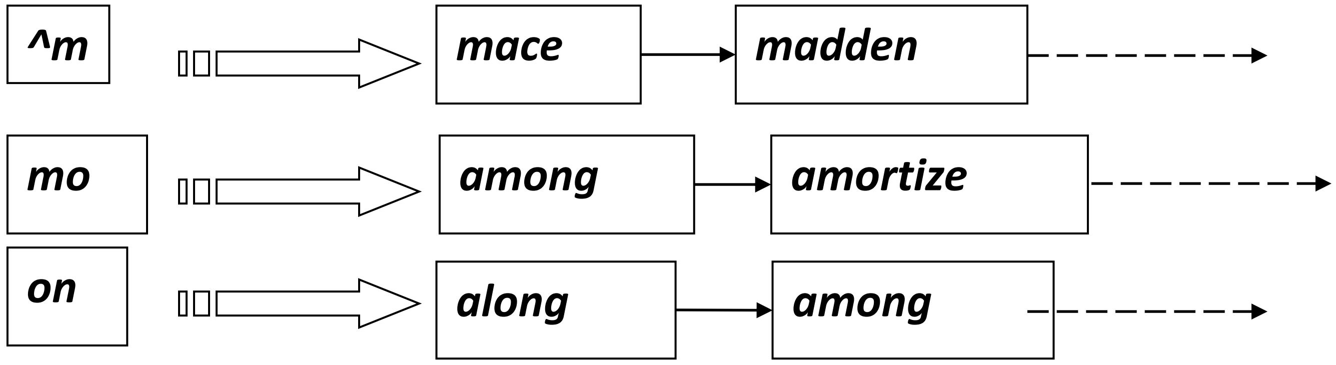
K-gram index

- For k-gram indexes, we can reuse a lot of ideas from our usual inverted indexes:
  - keys in a hashtable
  - values as arraylists



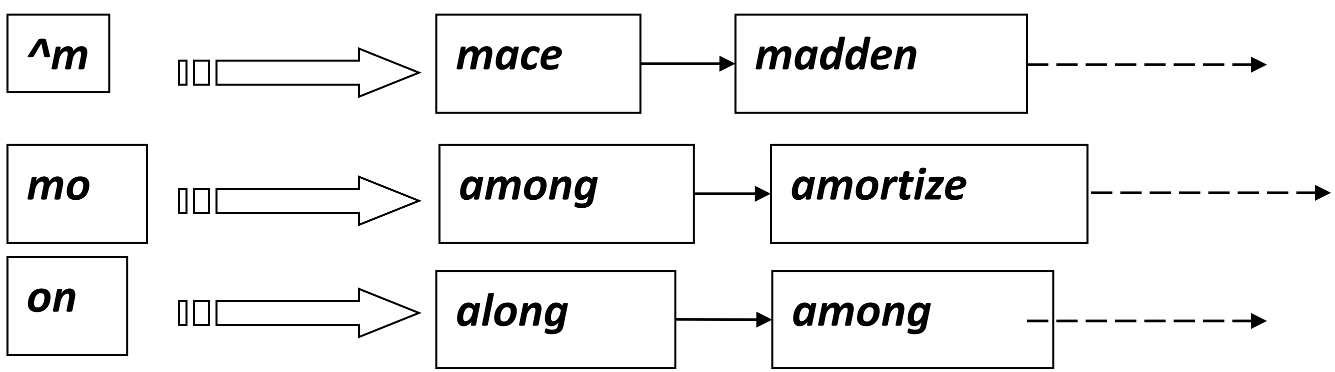
K-gram index and wildcards

- Suppose we want to find matches for **`mon*`**
- How would you search?



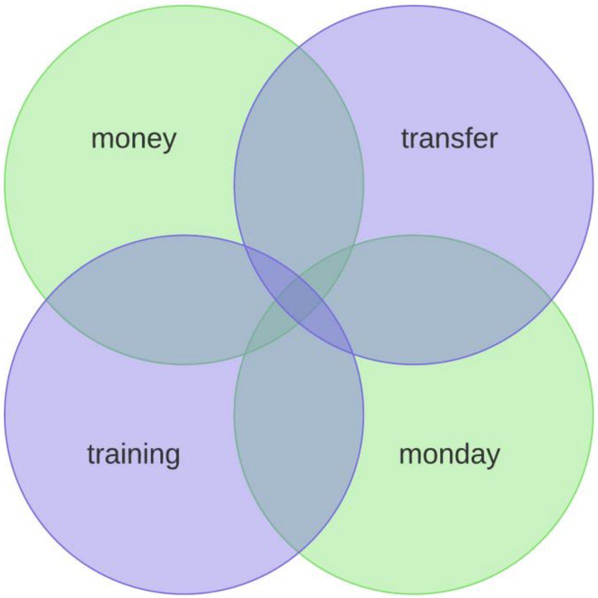
K-gram index and wildcards

- Suppose we want to find matches for **`mon*`**
- How would you search?
  - do an **intersection** search for **`^m mo on`** in k-gram index
  - post-process the results using the regex library
  - do a **union** search on the resulting words in the ordinary index



K-gram index and wildcards

- Suppose we want to find matches for `mon* tra*`
- Which documents in this Venn diagram are you looking for?



Spelling correction

Spelling correction

- Two principal uses
  - Correcting document(s) being indexed
  - Correcting user queries to retrieve “right” answers
    - Usually documents are left intact, but queries spell-checked
- Two main flavors:
  - Isolated word
    - Will not catch typos resulting in correctly spelled words, e.g., ***from*** $\rightarrow$ ***form***
  - Context-sensitive, e.g. ***I flew <u>form</u> Heathrow to Narita.***

Spelling correction

- Fundamental premise – there is a lexicon from which the correct spellings come
- Two basic choices for this
  - (**Grammatical approach**) A standard lexicon such as
    - Webster's English Dictionary
    - An "industry-specific" lexicon – hand-maintained
  - (**Data-driven approach**) The lexicon of the indexed corpus
    - E.g., all words on the web
    - All names, acronyms etc.
    - (Including the mis-spellings)

Spelling correction of a single word

- What we will do in assignment 3:
  - Data-driven approach (no lexicon)
  - Assumption: A word is **not** misspelt if it appears in at least 1 document.
  - If a word has 0 occurrences it **might** be misspelt, and the search engine should suggest corrections

Spelling correction of a single word

- Methods:
  - **Edit distance**
  - **Weighted edit distance**
  - ***n*-gram overlap**
- These can (should?) be combined

Levenshtein (edit) distance

- What is dist(*intention*, *execution*)?

`i n t e n t i o n`	← delete *i*
`n t e n t i o n`	← substitute *n* by *e*
`e t e n t i o n`	← substitute *t* by *x*
`e x e n t i o n`	← insert *u*
`e x e n u t i o n`	← substitute *n* by *c*
`e x e c u t i o n`	

- Cost 1+2+2+1+2 = 8
- Can be efficiently computed with dynamic programming

Computing Levenshtein distance

`above broke`

	#	a	b	o	v	e
#						
b						
r						
o						
k						
e						

Computing Levenshtein distance

`above broke`

	#	a	b	o	v	e
#	0	1	2	3	4	5
b	1					
r	2					
o	3					
k	4					
e	5					

Computing Levenshtein distance

above broke

	#	a	b	o	v	e
#	0	1	2	3	4	5
b	1	2				
r	2					
o	3					
k	4					
e	5					

Computing Levenshtein distance

`above broke`

	#	a	b	o	v	e
#	0	1	2	3	4	5
b	1	2	1	2	3	4
r	2	3	2	3	4	5
o	3	4	3	2	3	4
k	4	5	4	3	4	5
e	5	6	5	4	5	4

Weighted edit distances

- As above, but the weight of an operation depends on the character(s) involved
  - Meant to capture OCR or keyboard errors, e.g. ***m*** more likely to be mis-typed as ***n*** than as ***q***
  - Therefore, replacing ***m*** by ***n*** is a smaller edit distance than by ***q***
  - This may be formulated as a probability model
- Requires weight matrix as input
- Modify dynamic programming to handle weights

Using edit distances

- Given a misspelt word in the query, find all words in the index within a preset edit distance (e.g. 2)
    1. Show terms you found to user as suggestions, ***or***
    2. Look up all possible corrections in our inverted index and return all docs … slow, ***or***
    3. Run with a single most likely correction
- In assignment 3, we will opt for alternative 1.

Using edit distances

- Given a query, do we compute its edit distance to every dictionary term?
  - Expensive and slow
- How do we find the candidate dictionary terms?
  - One alternative: k-gram overlap
  - Use the k-gram index again!
  - Can also be used by itself for spelling correction

k-gram overlap

- Enumerate all the *k*-grams in the query string as well as in the lexicon
  - ***november:*** `^no, nov, ove, vem,` `emb, mbe, ber, er$`
  - ***december:*** `^de, dec, ece, cem,` `emb, mbe, ber, er$`
  - overlap 4/12 unique trigrams
  - the **Jacquard coefficient** = 4/12 = 0.33
  - generally, $\dfrac{|X \cap Y|}{|X \cup Y|}$ where X,Y are sets

Spelling correction of a single word

- E.g. **`wki`**
  - Do a union search in the k-gram index for **`^w wk ki i$`**
  - Calculate the Jacqard coefficient between **`wki`** and each of the resulting words
  - If the JC > some threshold for word *w*, calculate the Levenshtein distance between *w* and **`wki`**
  - If Levenstein distance < some other threshold, then *w* is a potential correction
  - Add *w* to the list of corrections

Spelling correction, multi-word queries

- E.g. ”`See yuoq on the wki`”
- Includes two (possibly) misspellt words: `yuoq` and `wki` with 0 postings
- Construct the lists of spelling suggestions for each word
  - Lists for words with > 0 postings will only contain themselves
  - Then merge the lists

Spelling correction, multi-word queries

- **`See youq on the wki`**

see	**you**	**on**	**the**	**wiki**
	your			**ki**
	youd			**wi**
	yous			**wk**
	youn			**waki**
	…			…
	…			…
	…			…

- Now the lists have to merged to produce final suggestions

Spelling correction, multi-word queries

- Final list of suggestions (for instance):
  - **`See you on the wiki`**
  - **`See you on the ki`**
  - **`See you on the wi`**
  - **`See you on the wk`**
  - **`See you on the waki`**

General issues in spell correction

- We enumerate several possible alternatives to misspelled queries – which ones should we present to the user?
- Use heuristics:
  - The alternative matching most documents (expensive)
  - **The alternative likely to match most documents** (using heuristics, cheaper)
  - Query log analysis – what have others been searching for? What has this user been searching for?

Query expansion

Query expansion

- improve retrieval results by adding synonyms / related terms to the query
- query-independent, ”global” method

Why are synonyms important?

- As an example consider query q: [aircraft] . . .
  - . . . and document d containing “plane”, but not containing “aircraft”
  - A simple IR system will not return d for q.
  - Even if d is the most relevant document for q!
- We want to change this:
  - Return relevant documents even if there is no term match with the (original) query

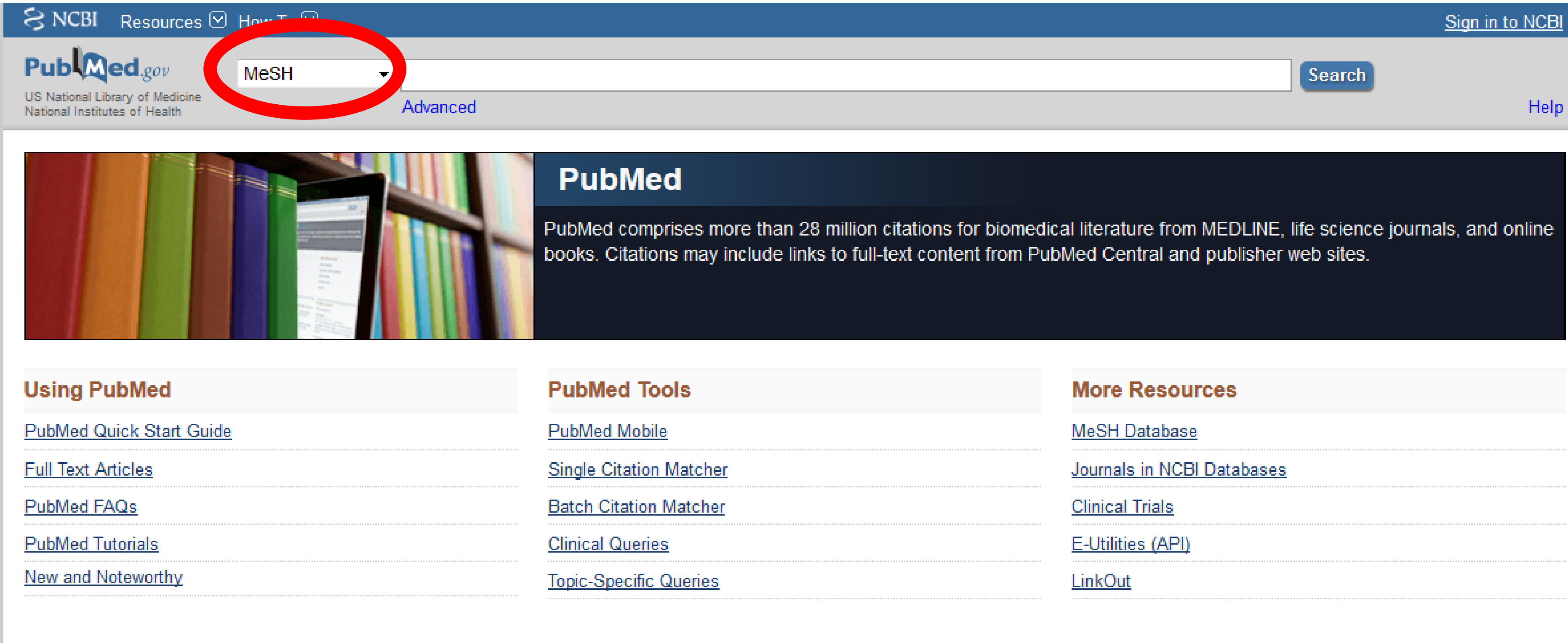
Query expansion

- In global query expansion, the query is modified based on some global resource, i.e. a resource that is **not query-dependent**.
- Main information we use: (near-)**synonymy**
- A publication or database that collects (near)-S synonyms is called a **thesaurus**.
- We will look at two types of thesauri: manually created and automatically created.

Thesaurus-based query expansion

- For each term t in the query, expand the query with words the thesaurus lists as semantically related with t.
  - E.g. CARDIAC → HEART
  - Generally increases recall
- May significantly decrease precision, particularly with ambiguous terms
  - INTEREST RATE → INTEREST RATE HOBBY
- Widely used in specialized search engines for science and engineering
- It’s very expensive to create a manual thesaurus and to maintain it over time.

PubMed: Manually curated thesaurus



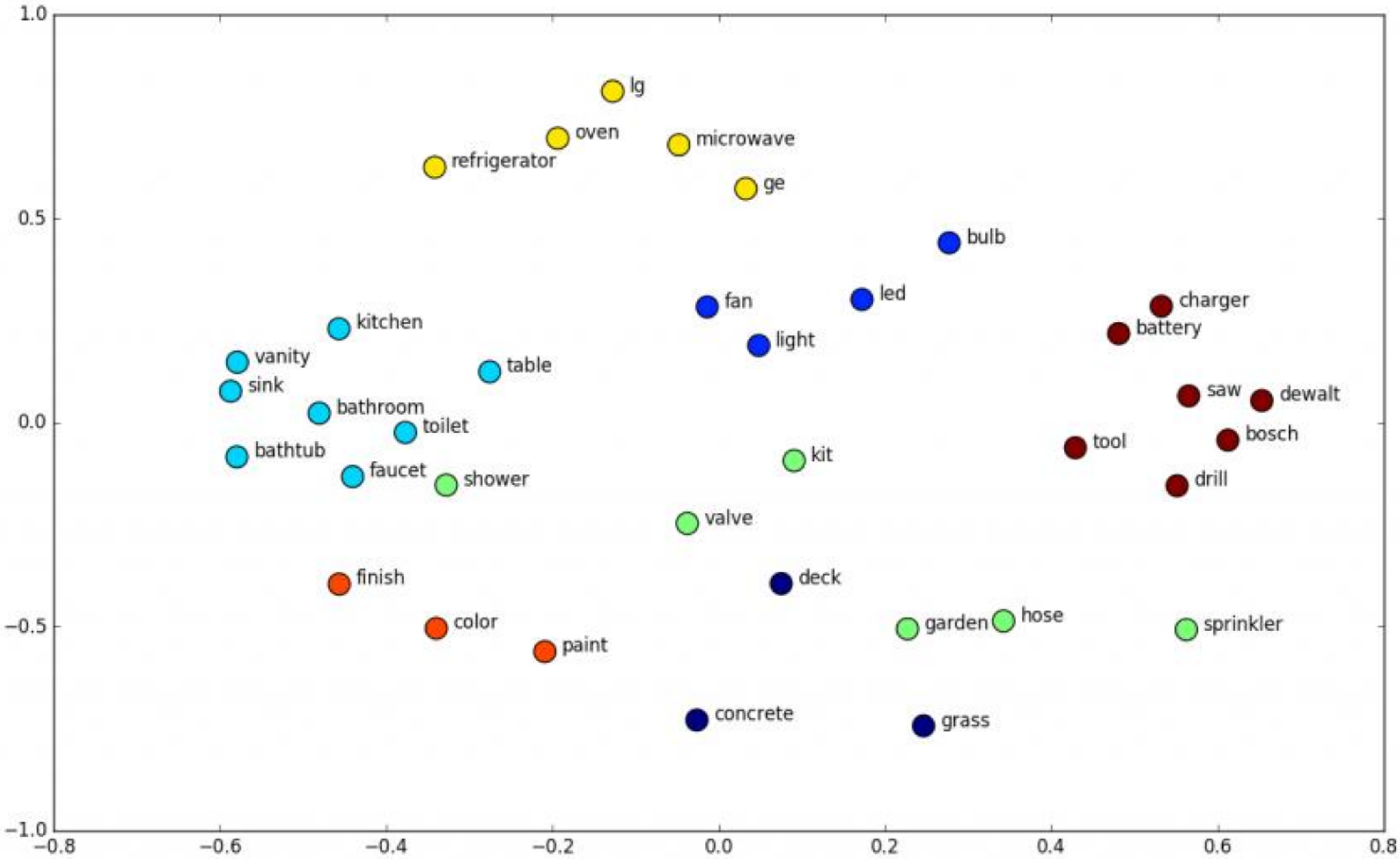
Automatic thesaurus construction

- Attempt to generate a thesaurus automatically by analyzing the distribution of words in documents
- Definition 1: Two words are similar if **they co-occur with similar words**.
  - “car” ≈ “motorcycle” because both occur with “road”, “gas” and “license”, so they must be similar.
- Definition 2: Two words are similar if **they occur in a given grammatical relation** with the same words.
  - You can harvest, peel, eat, prepare, etc. apples and pears, so apples and pears must be similar.
- Co-occurrence is more robust, grammatical relations are more accurate.

Word embedding approaches

- Mapping words to vectors of real numbers
- If w_1 and w_2 have similar meaning, then $vec(w_1)$ and $vec(w_2)$ are similar
- Many approaches exist:
  - Latent Semantic Analysis (LSA), Random Indexing, Word2Vec (2013), Glove (2014), Fasttext (2017), Elmo (2018)…
  - Vectors have typically 50-300 dimensions
  - Words with similar semantics can be retrieved with a Nearest Neighbor software package

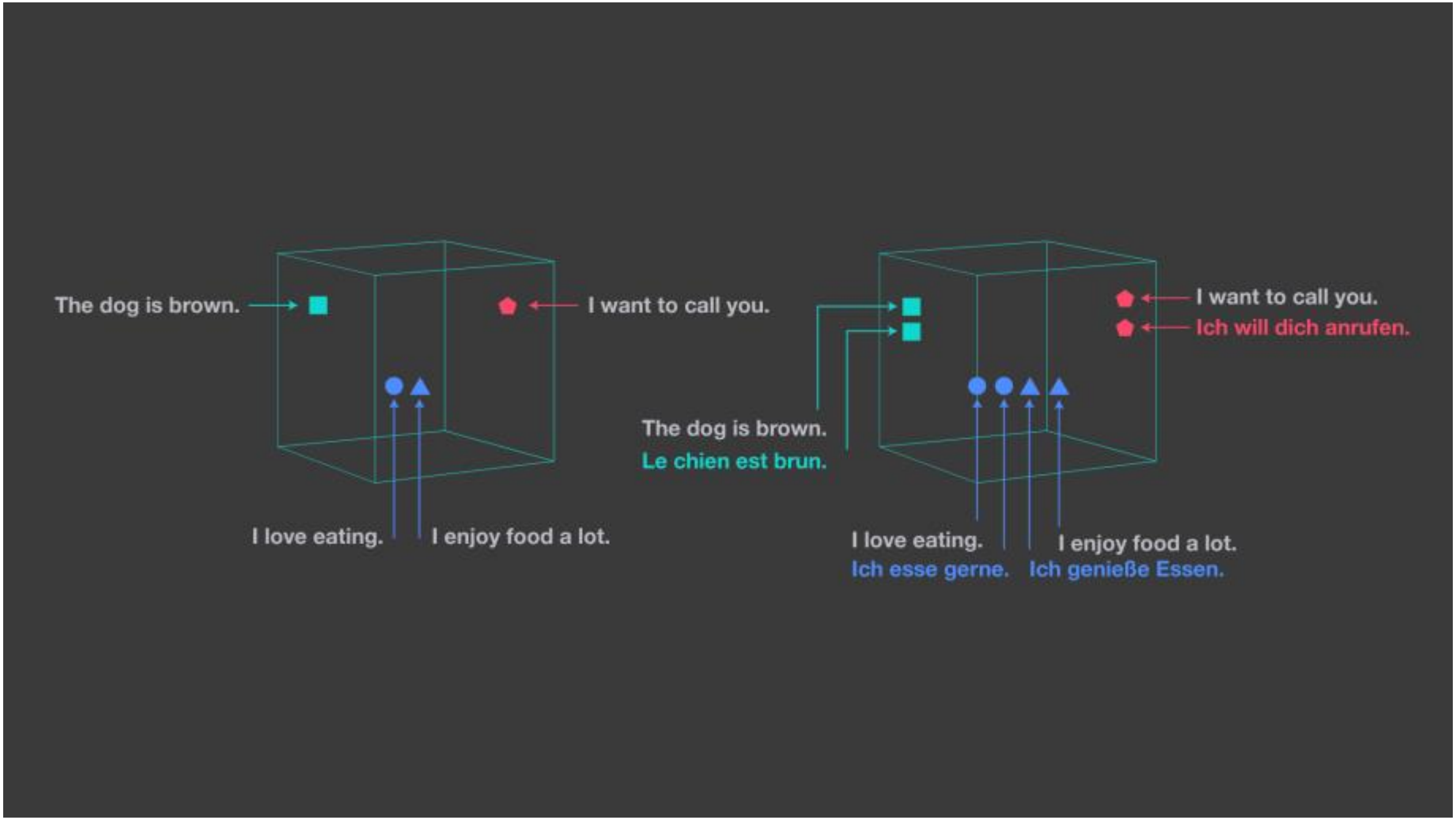
Word embedding approaches



Sentence embeddings

- Mapping sentences to vectors of real numbers
- If s_1 and s_2 have similar meaning, then vec(s_1) and vec(s_2) are similar
- Universal Sentence Encoder (2018), BERT (2018)
- LASER (2019) Multi-lingual sentence embeddings

LASER



The dog is brown.
I want to call you.
I love eating.
I enjoy food a lot.
I want to call you.
Ich will dich anrufen.
The dog is brown.
Le chien est brun.
I love eating.
Ich esse gerne.
I enjoy food a lot.
Ich genieße Essen.

Query expansion using query logs

- Example 1: After issuing the query [herbs], users frequently search for [herbal remedies].
  - → "herbal remedies" is potential expansion of "herb".
- Example 2: Users searching for [flower pix] frequently click on the URL photobucket.com/flower. Users searching for [flower clipart] frequently click on the same URL.
  - → "flower clipart" and "flower pix" are potential expansions of each other.

Summary

- Ways of improving recall in search:
  - **Relevance feedback** (assignment 3.1)
  - **Wildcard queries** (3.3, 3.4)
  - **Spelling correction** (3.5, 3.6)
  - **Query expansion**