# Indexing

and

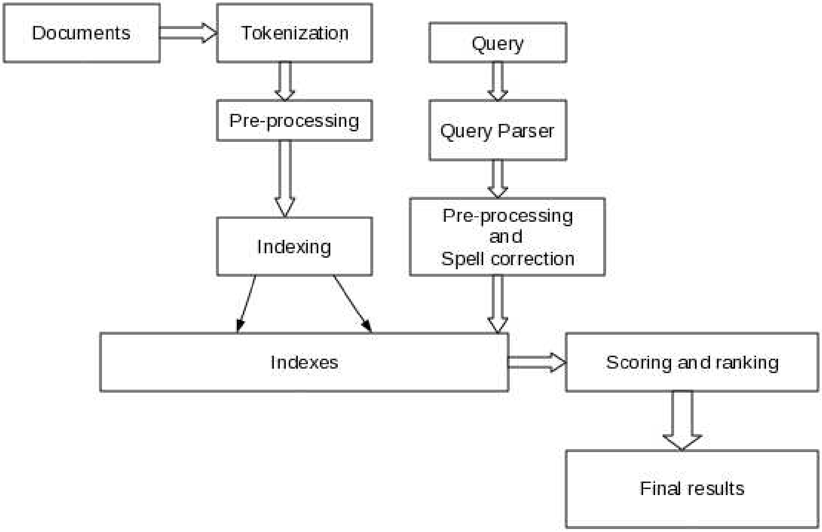
# Vector Space Model

**Jiaul Paik**
Indian Statistical Institute , Kolkata

# Plan of the Lecture

1. Tokenization
2. Pre-processing
3. Data Structures for Storing Dictionary
4. Index Constructon
5. Term Weighting
6. Vector Space Model

# IR system architecture : Overview



# Basic indexing steps

- Collect the documents.
- Tokenize the input text.
- Do pre-processing (stopwords, stemming etc.) of tokens.
- Construct the inverted index.

# Tokenization

- General strategy : chop on white spaces and throw away punctuation characters.
- Input: “Friends, Romans and Countrymen”.
- Output: Tokens
  - Friends
  - Romans
  - and
  - Contrymen
- A token is an instance of a sequence of characters.
- Each such token is now a candidate for an index entry, after further processing.

# Tokenization : Issues

- **O'Neill** $\longrightarrow$ neill, oneill, o'neill ?
- **Hewlett-Packard** $\longrightarrow$ Hewlett and Packard as two tokens?
- lower-case, lowercase or lower case?
- San Francisco : One token or two?

# Tokenization : language issues

- Compound words.
  - German IR benefits significantly (15%) from compound splitting.
    - Compound nouns without space : *Computerlinguistik* (computational linguistic)
    - Solution : Subdivide a word into multiple words that appear in vocabulary.

# Tokenization : language issues

- Chinese and japanese has no spaces between words.
- Arabic (or Hebrew) is basically written right to left, but with certain items like numbers written left to right.

# So far . . . and now

*Documents*

$\downarrow$

*Tokenization*

# So far . . . and now

*Documents*

$\downarrow$

*Tokenization*

$\downarrow$

*Pre processing*

# Pre-processing

- Case folding
- Stopwords
- Spelling correction and normalization
- Stemming
- Lemmatization

# Case folding

- Reduce all letters to lower case.
- exceptions
  - General Motors.
  - SAIL vs. sail

# Stopwords

- Remove the words/terms from the index which have no or negligable information
- Pros : Disk space save. (25% to 30%)
- Cons : Not good if the system has to handle phrase query, song names, etc.
  - As we may think.
  - To be or not to be.
- Common approach
  - Sort the terms based on descending collection frequency.
  - Take top few terms.
  - Hand filter the list.

# Spelling normalization and correction

- **Normalization**
  - color vs. colour.
- **Correction**
  - Isolated term : attempts to correct single query term at a time.
    - Edit distance
    - k-gram overlap
  - Context sensitive : *flew form Heathrow*
    - Using query log : frequent query

# Normalizing morphological variants

- Inflectional morphology
- Derivational morphology
- Two methods : lemmatization and stemming
- Why we need to care?
  - Significant performance improvement for more complex languages.
  - Retrieves more relevant documents.
  - Index size reduction.

# Lemmatization

- Reduce inflectional/variant forms to base form

# Lemmatization

- Reduce inflectional/variant forms to base form
- Example: *am, are, is* $\rightarrow$ *be*

# Lemmatization

- Reduce inflectional/variant forms to base form
- Example: *am, are, is* $\longrightarrow$ *be*
- Example: *car, cars, car's, cars'* $\longrightarrow$ *car*

# Lemmatization

- Reduce inflectional/variant forms to base form
- Example: *am, are, is* $\longrightarrow$ *be*
- Example: *car, cars, car's, cars'* $\longrightarrow$ *car*
- Example: *the boy's cars are different colors* $\longrightarrow$ *the boy car be different color*

# Lemmatization

- Reduce inflectional/variant forms to base form
- Example: *am, are, is* $\longrightarrow$ *be*
- Example: *car, cars, car's, cars'* $\longrightarrow$ *car*
- Example: *the boy's cars are different colors* $\longrightarrow$ *the boy car be different color*
- Lemmatization implies doing "proper" reduction to dictionary headword form (the lemma).

# Lemmatization

- Reduce inflectional/variant forms to base form
- Example: *am, are, is* $\longrightarrow$ *be*
- Example: *car, cars, car's, cars'* $\longrightarrow$ *car*
- Example: *the boy's cars are different colors* $\longrightarrow$ *the boy car be different color*
- Lemmatization implies doing "proper" reduction to dictionary headword form (the lemma).
- Inflectional morphology (*cutting* $\longrightarrow$ *cut*) vs. derivational morphology (*destruction* $\longrightarrow$ *destroy*)

# Stemming

- Definition of stemming: Crude heuristic process that chops off the ends of words in the hope of achieving what "principled" lemmatization attempts to do with a lot of linguistic knowledge.

# Stemming

- Definition of stemming: Crude heuristic process that chops off the ends of words in the hope of achieving what "principled" lemmatization attempts to do with a lot of linguistic knowledge.
- Language dependent

# Stemming

- Definition of stemming: Crude heuristic process that chops off the ends of words in the hope of achieving what "principled" lemmatization attempts to do with a lot of linguistic knowledge.
- Language dependent
- Often inflectional and derivational

# Stemming

- Definition of stemming: Crude heuristic process that chops off the ends of words in the hope of achieving what “principled” lemmatization attempts to do with a lot of linguistic knowledge.
- Language dependent
- Often inflectional and derivational
- Example for derivational: *automate, automatic, automation* all reduce to *automat*

# Stemming

- Definition of stemming: Crude heuristic process that chops off the ends of words in the hope of achieving what "principled" lemmatization attempts to do with a lot of linguistic knowledge.
- Language dependent
- Often inflectional and derivational
- Example for derivational: *automate, automatic, automation* all reduce to *automat*
- Common strategy : Indentify (manually or statistically) a set of suffixes and remove from the ends of words

# Porter algorithm

- Most common algorithm for stemming English

# Porter algorithm

- Most common algorithm for stemming English
- Results suggest that it is at least as good as other stemming options

# Porter algorithm

- Most common algorithm for stemming English
- Results suggest that it is at least as good as other stemming options
- 5 phases of reductions

# Porter algorithm

- Most common algorithm for stemming English
- Results suggest that it is at least as good as other stemming options
- 5 phases of reductions
- Phases are applied sequentially

# Porter algorithm

- Most common algorithm for stemming English
- Results suggest that it is at least as good as other stemming options
- 5 phases of reductions
- Phases are applied sequentially
- Each phase consists of a set of commands.

# Porter algorithm

- Most common algorithm for stemming English
- Results suggest that it is at least as good as other stemming options
- 5 phases of reductions
- Phases are applied sequentially
- Each phase consists of a set of commands.
  - Sample command: Delete final *ement* if what remains is longer than 1 character

# Porter algorithm

- Most common algorithm for stemming English
- Results suggest that it is at least as good as other stemming options
- 5 phases of reductions
- Phases are applied sequentially
- Each phase consists of a set of commands.
  - Sample command: Delete final *ement* if what remains is longer than 1 character
  - replacement $\longrightarrow$ replac

# Porter algorithm

- Most common algorithm for stemming English
- Results suggest that it is at least as good as other stemming options
- 5 phases of reductions
- Phases are applied sequentially
- Each phase consists of a set of commands.
  - Sample command: Delete final *ement* if what remains is longer than 1 character
  - replacement → replac
  - cement → cement

# Porter algorithm

- Most common algorithm for stemming English
- Results suggest that it is at least as good as other stemming options
- 5 phases of reductions
- Phases are applied sequentially
- Each phase consists of a set of commands.
  - Sample command: Delete final *ement* if what remains is longer than 1 character
  - replacement $\longrightarrow$ replac
  - cement $\longrightarrow$ cement
- Sample convention: Of the rules in a compound command, select the one that applies to the longest suffix.

# Porter stemmer: A few rules

| **Rule** | | | **Example** | | |
|---|---|---|---|---|---|
| SSES | $\longrightarrow$ | SS | caresses | $\longrightarrow$ | caress |
| IES | $\longrightarrow$ | I | ponies | $\longrightarrow$ | poni |
| SS | $\longrightarrow$ | SS | caress | $\longrightarrow$ | caress |
| S | $\longrightarrow$ | | cats | $\longrightarrow$ | cat |

# No stem, stem and lemmatization : A comparison

*Input text:* Such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation

# No stem, stem and lemmatization : A comparison

*Input text:* Such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation

*Porter stemmer:* such an analysi can reveal featur that ar not easili visibl from the variat in the individu gene and can lead to a pictur of express that is more biolog transpar and access to interpret

# No stem, stem and lemmatization : A comparison

*Input text:* Such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation

*Porter stemmer:* such an analysi can reveal featur that ar not easili visibl from the variat in the individu gene and can lead to a pictur of express that is more biolog transpar and access to interpret

*Lemmatized text :* Such an analysis can reveal feature that are not easily visible from the variation in the individual gene and can lead to a picture of expression that is more biologically transparent and accessible to interpretation

# Does stemming improve effectiveness?

- In general, stemming increases effectiveness for some queries, and decreases effectiveness for others.
- Some languages (Hungarian, Czech) enjoy 30% to 40% benefits.
- Queries where stemming is likely to help
  - tartan sweaters
    - (equivalence class: {sweater,sweaters})
  - sightseeing tour san francisco
    - (equivalence class: {tour,tours})
- Porter Stemmer maps all of {*operate, operating, operates, operation, operative, operatives, operational* } to *oper*.
- Queries where stemming hurts:
  - operational research
  - operating system
  - operative dentistry
- Stemming reduces the vocabulary size significantly.

# So far . . . and now

*Documents*

↓

*Tokenization*

↓

*Pre processing*

# So far . . . and now

*Documents*

$\downarrow$

*Tokenization*

$\downarrow$

*Pre processing*

$\downarrow$

*Indexing*

# Inverted index

- **For each term $t$, we store a list of all documents that contain $t$ along with the $tf$.**

Input : Documents

id1 : Web mining is useful.

id2 : Usage mining applications.

id3 : Web structure mining studies the hyperlink structure of web.

Output : Inverted Index

| Dictionary | | Postings |
|---|---|---|
| applications | : | $\langle id2, 1, [3] \rangle$ |
| hyperlink | : | $\langle id3, 1, [6] \rangle$ |
| mining | : | $\langle id1, 1, [2] \rangle$, $\langle id2, 1, [2] \rangle$, $\langle id3, 1, [3] \rangle$ |
| structure | : | $\langle id3, 2, [2, 7] \rangle$ |
| studies | : | $\langle id3, 1, [4] \rangle$ |
| usage | : | $\langle id2, 1, [1] \rangle$ |
| useful | : | $\langle id1, 1, [4] \rangle$ |
| web | : | $\langle id1, 1, [1] \rangle$, $\langle id3, 2, [1, 8] \rangle$ |

# Dictionary data structures for inverted index

- The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list.
  - Term vocabulary : the data
  - Document frequency : the no. of documents that contain the term

# Dictionary as array of fixed-width entries

- For each term, we need to store a couple of items:
  - document frequency
  - pointer to postings list
  - ...
- Assume for the time being that we can store this information in a fixed-length entry.
- Assume that we store these entries in an array.

# Dictionary as array of fixed-width entries

| term | document frequency | pointer to postings list |
|---|---|---|
| a | 656,265 | $\longrightarrow$ |
| aabir | 65 | $\longrightarrow$ |
| ... | ... | ... |
| zonathan | 221 | $\longrightarrow$ |

# Dictionary as array of fixed-width entries

| term | document frequency | pointer to postings list |
|---|---|---|
| a | 656,265 | $\longrightarrow$ |
| aabir | 65 | $\longrightarrow$ |
| ... | ... | ... |
| zonathan | 221 | $\longrightarrow$ |

How do we look up a query term $q_i$ in this array at query time? That is: which data structure do we use to locate the entry (row) in the array where $q_i$ is stored?

# Data structures for looking up term

- Two main classes of data structures: hashes and trees
- Some IR systems use hashes, some use trees.
- Criteria for when to use hashes vs. trees:
  - Is there a fixed number of terms or will it keep growing?
  - How many terms are we likely to have?

# Hashes

- Each vocabulary term is hashed into an integer.
- Try to avoid collisions
- At query time, do the following: hash query term, resolve collisions, locate entry in fixed-width array
- Pros: Lookup in a hash is faster than lookup in a tree.
  - Lookup time is constant.
- Cons
  - no prefix search (all terms starting with *automat*)
  - need to rehash everything periodically if vocabulary keeps growing

# Trees

- Trees solve the prefix problem (find all terms starting with *automat*).
- Simplest tree: binary search tree
- Search is slightly slower than in hashes: $O(\log M)$, where $M$ is the size of the vocabulary.
- $O(\log M)$ only holds for balanced trees.
- Rebalancing binary search trees is expensive.
- B-trees mitigate the rebalancing problem.
- B-tree definition: every internal node has a number of children in the interval $[a, b]$ where $a, b$ are appropriate positive integers, e.g., $[2, 4]$.
- Trie

# Index Construction

- How do we construct an index efficiently?
- What strategies can we use with limited main memory?

# Hardware basics

- Access to data in main memory faster than in disk.
- Disk seeks: No data is transferred from disk while the disk head is being positioned.
- Therefore: Transferring one large chunk of data from disk to memory is faster than transferring many small chunks.
- Disk I/O is blockbased: Reading and writing of entire blocks (as opposed to smaller chunks).

# Index Construction : Key Steps

- Documents are parsed to extract words and these are saved with the Document ID.
- After all documents have been parsed, the inverted file is sorted by terms.

# Scaling Index Construction

- In-memory index construction does not scale.
- How can we construct an index for very large collections?

# Sort Based Index Construction

- As we build the index, we parse docs one at a time.
- The final postings for any term are incomplete until the end.
- Store the term (or termID), docid pairs in main memory buffer.
- Sort (term/termID as primary key, docid as secondary key) and write to intermediate files in disk when the buffer is full.
- Merge all intermediate files into a sorted file.
- Finally build the inverted index in a linear scan.

# Problem with sort based indexing

- Assumption : we can keep dictionary into main memory.
- We need the dictionary (which grows dynamically) in order to implement a term to termID mapping.
- Actually, we could work with term, docID postings instead of termID, docID postings. But then intermediate files are large and we end up with a scalable but slow method.

# Single pass in-memory indexing

- Key idea 1: Generate separate dictionaries for each block no need to maintain term-termID mapping across blocks.
- Key idea 2: Accumulate postings in postings lists as they occur.
- With these two ideas we can generate a complete inverted index for each block.
- These separate indexes can then be merged into one big index.

# So far . . .

*Documents*

$\downarrow$

*Tokenization*

$\downarrow$

*Pre processing*

$\downarrow$

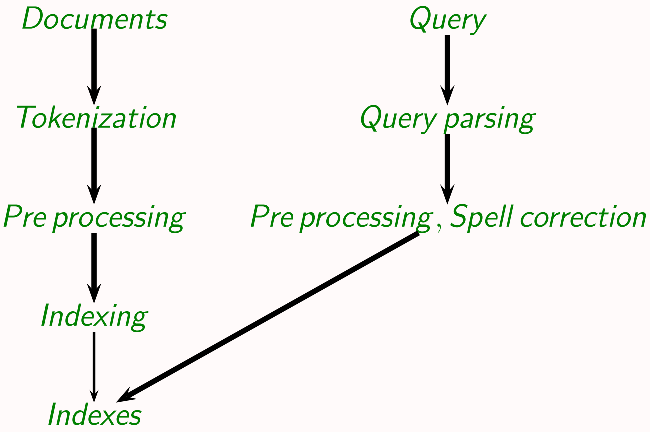
*Indexing*

$\downarrow$

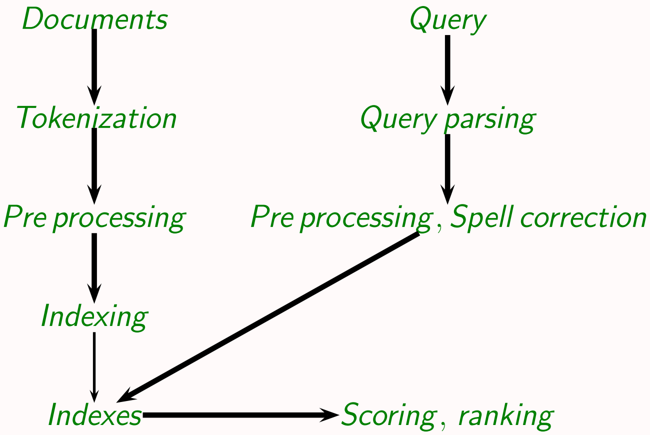
$\langle$*Indexes*$\rangle$

TEA BREAK

# So far . . . and now



# So far ... and now



# The retrieval problem

- Input
  - A set of documents : $d_1, d_2, \ldots d_n$
  - A query $q = q_1 \ q_2 \ \ldots \ q_n$
- Output : A set of documents relevant to the query $q$.
- The first idea
  - Just look if a document contain the query terms.
  - Not good because too much / or too few match.
  - Users hardly read beyond first few.
  - Solution : Ranked retrieval (return documents based on the matching score)

# Ranked retrieval

- Rather than a set of documents satisfying a query expression, in ranked retrieval models, the system returns an ordering over the (top) documents in the colletion with respect to a query
- Free text queries: Rather than a query language of operators and expressions, the users query is just one or more words in a human language

# Scoring as the basis of ranked retrieval

- We wish to return in order the documents most likely to be useful to the searcher
- How can we rank the documents in the collection with respect to a query?
- Assign a score to each document with respect to the query
- This score measures how well document and query match.

# Query-document matching scores

- We need a way of assigning a score to a query-document pair
- Let's start with a one-term query
  - If the query term does not occur in the document: score should be 0
  - The more frequent the query term in the document, the higher the score (should be)
  - We will look at a number of alternatives for this.
- For the rest of the talk : bag-of-words model assumption
  - Doesn't consider the ordering of words in a document
  - John is quicker than Mary and Mary is quicker than John have the same representation

# Term weighting : term frequency

- The term frequency $tf_{t,d}$ of a term $t$ in a document $d$ is defined as the no. of times $t$ occurs in document $d$.
- We want to use $tf$ as a measure of importance when matching query with a document.
- Raw term frequency is not a good idea.
  - A document with 5 occurrences of a term is more relevant than a document with 1 occurrence of the same term.
  - But not 5 times relevant.
- Relevance does not increase proportionally with term frequency.

# Scaling term frequency

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

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

- $\mathrm{tf}_{t,d} \rightarrow \mathrm{w}_{t,d}$:
  $0 \rightarrow 0$, $1 \rightarrow 1$, $2 \rightarrow 1.3$, $10 \rightarrow 2$, $1000 \rightarrow 4$, etc.
- Score for a document-query pair: sum over terms $t$ in both $q$ and $d$:
  **tf-matching-score(q,d)** $= \sum_{t \in q} (1 + \log \mathrm{tf}_{t,d})$
- The score is 0 if none of the query terms is present in the document.

# Desired weight for rare terms

- Rare terms are more informative than frequent terms.
- Consider a term in the query (capricious person) that is rare in the collection (e.g., **capricious**).
- A document containing this term is very likely to be relevant.
- $\rightarrow$ We want high weights for rare terms like **capricious**.

# Desired weight for frequent terms

- Frequent terms are less informative than rare terms.
- Consider a term in the query that is frequent in the collection (e.g., **good**, **increase**, **person**).
- A document containing this term is more likely to be relevant than a document that doesn't ...
- ... but words like **good**, **increase** and **person** are not sure indicators of relevance.
- $\rightarrow$ For frequent terms like **good**, **increase**, and **person**, we want positive weights ...
- ... but lower weights than for rare terms.

# Document frequency

- We want high weights for rare terms like **capricious**.
- We want low (positive) weights for frequent words like **good**, **increase**, and **person**.
- 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.

# idf weight

- $\mathrm{df}_t$ is the document frequency, the number of documents that $t$ occurs in.
- $\mathrm{df}_t$ is an inverse measure of the informativeness of term $t$.
- We define the idf weight of term $t$ as follows:

$$\mathrm{idf}_t = \log_{10} \frac{N}{\mathrm{df}_t}$$

  ($N$ is the number of documents in the collection.)
- $\mathrm{idf}_t$ is a measure of the informativeness of the term.
- $[\log N/\mathrm{df}_t]$ instead of $[N/\mathrm{df}_t]$ to "dampen" the effect of idf
- Note that we use the log transformation for both term frequency and document frequency.

# Examples for idf

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

| term | $\text{df}_t$ | $\text{idf}_t$ |
|---|---:|---:|
| calpurnia | 1 | 6 |
| animal | 100 | 4 |
| sunday | 1000 | 3 |
| fly | 10,000 | 2 |
| under | 100,000 | 1 |
| the | 1,000,000 | 0 |

# Effect of idf on ranking

- idf has no effect on ranking for one-term queries.
    - For example : iPhone
- idf affects the ranking of documents for queries with at least two terms.
- For example, in the query *capricious person*, idf weighting increases the relative weight of **capricious** and decreases the relative weight of **person**.

# Collection frequency vs. Document frequency

| word | 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
- Which word is a better search term (and should get a higher weight)?
- This example suggests that df (and idf) is better for weighting than cf (and "icf").

# tf-idf weighting

- The tf-idf weight of a term is the product of its tf weight and its idf weight.
- $w_{t,d} = (1 + \log \mathrm{tf}_{t,d}) \cdot \log \frac{N}{\mathrm{df}_t}$
- Best known weighting scheme in information retrieval

# Summary: tf-idf

- Assign a tf-idf weight for each term $t$ in each document $d$: $w_{t,d} = (1 + \log \mathrm{tf}_{t,d}) \cdot \log \frac{N}{\mathrm{df}_t}$
- 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)
  - Benefit is maximum when a term occurs many times within a small no. of documents
  - Lowest when the term occurs in virtually all documents.

# Scoring function

- $\mathbf{Score(q, d)} = \sum_{\mathbf{t \in q}} \mathbf{w_{t,q}} \times \mathbf{w_{t,d}}$

  For example, $w_{t,q}$ may be no. of time term $t$ occurs in $q$

# Vector Space Model : Documents as vectors

- Each document is now represented as a real-valued vector of tf-idf weights $\in \mathbb{R}^{|V|}$. $|V|$ is the no. of distinct terms in the collection.
- 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.

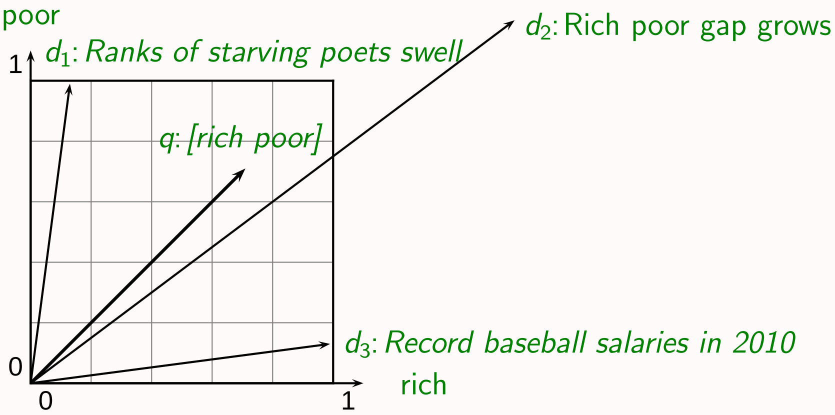
# Queries as vectors

- 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 $=$ similarity
- proximity $\approx$ inverse of distance

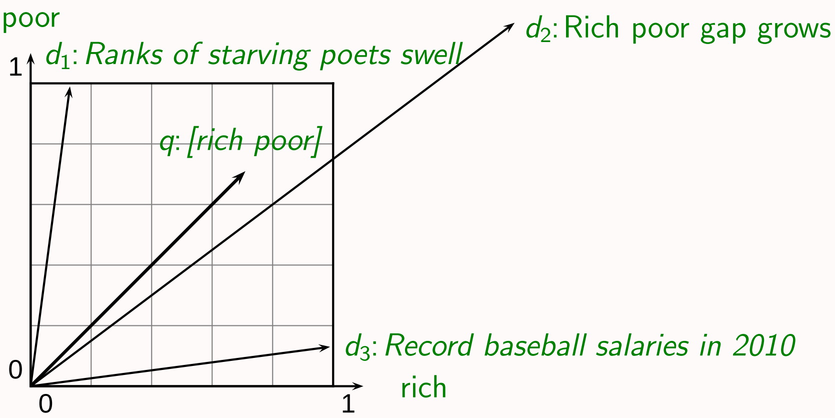
# How do we formalize vector space similarity?

- First cut: distance between two points?
- Euclidean distance?
- Euclidean distance is a bad idea
  - Because Euclidean distance is large for vectors of different lengths.

# Why distance is a bad idea?



# 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.

# Use angle instead of distance

- Rank documents according to angle with query
- An experiment
  - Take a document $d$ and append it to itself. Call this document $d'$. $d'$ is twice as long as $d$.
    - For example let $d =$ *Sachin is a cricketer* and $d' =$ *Sachin is a cricketer. Sachin is a cricketer*
  - "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.

# From angles to cosines

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

# Ranked retrieval in vector space model : summary

- 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 based on the similarity score
- Return the top $K$ (e.g., $K = 10$) to the user

# Computing the cosine score using inverted index

CosineScore($q$)

*float* $Scores[N] = 0$
*float* $Length[N]$
**for each** query term $t$
**do** calculate $\mathrm{w}_{t,q}$ and fetch postings list for $t$
    **for each** pair$(d, \mathrm{tf}_{t,d})$ in postings list
    **do** $Scores[d] += \mathrm{w}_{t,d} \times \mathrm{w}_{t,q}$
Read the array *Length*
**for each** $d$
**do** $Scores[d] = Scores[d]/Length[d]$
**return** Top $K$ components of $Scores[]$

# References


1. Introduction to Information Retrieval. 2008. By Manning et. al.
2. Managing Gigabytes: Compressing and Indexing Documents and Images. 1999. By Witten et.al.
3. Term Weighting Approaches to Automatic Text Retrieval. 1998. by Salton and Buckley

THANK YOU!