Vector space model
- matching word more times more credit
- some words are more important than other words

Improved Vector Placement
- Term frequency vector
  - Sim(q,d) = q.d (dot product)
  - inverse document frequency
    - count the number of documents that don't contain a particular word
  - document frequency
    - count of documents that contain a particular word

IDF (inverse document frequency) (common words have low IDF and rare words have high IDF)
- log[(M+1)/k]
  - k = total number of docs containing W (doc frequency)
  - M = total number of docs in collection

How effective is VSM with TF-IDF weighting
- fixing one problem tends to lead to another problem

—————-
Ranking function with TF-IDF Weighting
- f(q,d) = summation x_i y_i to N from 1
  - = summation all matched query words in document d
  - c(w,q) c(w,d) (log(M+1)/df(w))

TF Transformation: c(w,d) -> TF(w,d)
- linear y = x
- 0/1 bit vector
- y = log(1+x)
  - controls the inference of high weight
- y = log(1 + log(1 + x))
- BM25 transformation (best function)
  - y = ((k+1)x)/(x+k)
    - upper bounded by k + 1
    - varying k can simulate different transformation functions
    - setting k = 0 turns it into a bit vector 0/1 transformation
    - setting k to very large number makes the transformation look like a linear transformation
    - upper bound controls the inference from the upper bound
- Sub linear TF Transformation is need to
  - capture the intuition of diminishing return from higher TF
  - avoid dominance by one single term over all others
- Ranking function with BM25
——
Doc Length Normalization
- penalize a long doc with a length normalizer
- pivoted length normalizer: average doc length as pivot
  - normalizer = 1 if |d| = average doc length

Document length normalizer
- if length of document is longer than the pivot then there is a penalty
- normalizer = 1 - b + b (ldl/avdl) b element of [0 ,1]
- b controls the normalization
  - b = 0 no normalization
  - b > 0 the value would be higher for longer documents and reward for short documents. Nothing if equal
  - adjusting B we can control the degree of normalization

State of the Art VSM Ranking Functions
- pivoted length normalization VSM
  - summation of all matched query words in the documents c(w,q)
  - TF Transformation on the top. Double ln ln
  - document length normalizer on the bottom (using the pivot)
  - inverse document frequency normalizer
- BM25
  - inverse document frequency normalizer
  - summation of all matched query words in the documents c(w,q)
  - middle normalization
    - BM25 TF normalization
    - then a document length normalization multiplied by the BM 25 normalizer on the bottom

Further improvement
- improved instantiation of dimension?
  - stemmed words, stop word removal, phrases, latent semantic indexing, character n-grams
  - bag of words sufficient
  - language-specific and domain specific tokenization is important to ensure “normalization of terms”
- improved instantiation of similarity function?
  - cosine of angle between two vectors?
  - euclidean?
  - dot product seems still the best (sufficiently general especially with appropriate term weighting)

BM25
- further improvement?
  - combine the frequency counts of terms in all fields and then apply BM25
  - address the problem of over penalization of long documents by BM25 by adding a small constant to TF

Summary
- Relevance(q,d) = similarity(q,d)
- query and documents represent as vectors
- Heuristic design of ranking function.
- Major term weighting heuristics
  - TF, IDF, and document length normalization
- BM25 and Pivoted normalization seem to be most effective
  - BM25 derived using probabilistic modeling

———-

Typical TR System Architecture
- three parts
  - indexer
  - scorer
  - feedback
    - online or offline
- tokenization
  - normalize lexical units = words with similar meanings should be mapped to the same indexing term
  - stemming: mapping all inflectional forms of words to the same root form
  - some languages pose challenges in word segmentation
- indexing
  - indexing = convert documents to data structures that enable fast search
  - inverted index is the dominating indexing method for supporting basic search algorithms
  - other indices may be needed for feedback
- inverted index
  - dictionary and postings
    - dictionary - contains the count of documents and the total frequency
    - postings - contains the document id and the frequency
      - can also store the positions of the words
- inverted index for fast search
  - single term query?
  - multi-term boolean query?
  - multi-term keyword query?

Empirical distribution of words

Zipf's law
- rank * frequency ~~ constant
- F(w) = C / r (w) ^ alpha

Data Structures for Inverted Index
- Dictionary: modest size
  - fast random access
  - preferred to be in memory
  - Hash table, B-tree, trie
- Postings: huge
  - sequential access is expected
  - can stay on disk
  - may contain docID, term frequency, term pos, etc
  - compression is desirable

Inverted Index Construction
- memory based methods: not usable for large collections
- sort-based methods:
  - collect local (termID, docID, freq) tuples
  - sort local tuples (to make "runs")
  - pair-wise merge runs

  - output inverted file

Sort-based Inversion
- obtain all the information containing the
  - document identification
  - term identification
  - count
- merge sort so that the entries are sorted based on term identifications

Inverted Index Compression (Encodings)
- leverage skewed distribution of values and use variable length encoding
- TF compression
  - small numbers tend to occur far more frequently than large numbers
  - fewer bits for small (high frequency) integers at the cost of more bits for large integers
- Doc id compression
  - "d-gape" (store difference): d1, d2-d1, d3-d2,….
  - feasible due to sequential access

Integer Compression Methods
- binary encoding
- unary encoding
- gamma encoding

Uncompress Inverted Index
- decoding of encoded integers
  - unary decoding
  - gamma decoding
- decode doc ids encoded using d-gap

———
How to Score Documents Quickly?
- general form of scoring function

f(q,d) = f_a(h (g(t_1,d,q), …), f_d(d), f_q(q))
- f_a
  - combines all the functions
- h
  - inside of function
    - functions that compute the weights of contribution of matched query term in d
    - inside functions weight aggregation
- f_d, f_q
  - adjustment/scoring factors of document and query

General algorithm for ranking documents

- $f\_d(d)$ and $f\_q(q)$ are precomputed
- maintain a score accumulator for each d to compute h
- for each query term t_i
  - fetch the inverted list {(d_1,f_1),…,(d_n,f_n)}
  - for each entry (d_j,f_j), compute g(t_i,d_i,q), and update score accumulator for doc d_i to incrementally compute h
- adjust the score to compute the function f_a then sort

Ranking based on TF Sum

- f(d,q) = g(t_1,d,q) + …. where g(t_i,d,q) = c(t_i,d)

Further Improving Efficiency

- Caching
- Keep only the most promising accumulators
- Scaling up to the Web-scale?