1 Language Models

1.1 Ngram model

- Unigram model (n = 1) does not model word order (just a "bag of words").
- Can be defined as bags of phrases of length n.
- OR a model that can predict a current word from the $(n-1)$ previous context words, e.g. in a pentagram model $P(?? \mid$ "Please turn off your hand")
- Longer n-gram models are exponentially more costly to construct. Let $|V|$ = size of vocab in a language. In unigram, store probabilities for all $|V|$ words. In bigram, need to store all $|V| \times |V|$ ordered length 2 phrases.

1.1.1 Markov Assumption

- Presumption that the future behaviour of a dynamical system only depends on its recent history.
- A **kth-order Markov model**, the next state only depends on the k most recent states.
- Hence, an N-gram model = (N - 1)-order Markov model.

1.1.2 Add 1 smoothing

- Problem with zero probability.
- Add 1 count to all entries in the LM, including those that are not seen.
- Not used in practice, but most basic to understand.
- Add-one or add-k smoothing represents a uniform belief in all events (i.e. possible ngrams) as it assigns a simple $\frac{1}{n}$ probability to all events.
- Applied to:
  - **small** no of observations, **large** vocab space: **poor** choice (shifting the probability mass quite drastically, given small no observation but large no of possible outcomes (vocab space). Extreme example: English alphabet, only observe "a", then P(a) = 1.0, but with add-one smoothing, P(a) = $\frac{2}{27} = 0.074$)
  - **large** no of observations, **small** vocab space: **good** choice (only shift a small probability mass while maintaining the important guarantee of non-zero probability for all possible events.)

2 Boolean Retrieval

2.1 Term-document Incidence

	Harriet	Othello	Macbeth
Antony	0	0	1
Brutus	1	0	0
Caesar	1	1	1

- A vector of 1s and 0s for each term, whether they appear in the document.
- Query processing: take the vectors and perform bitwise AND (complement for NOT)
- Disadvantage: Big, but sparse matrix.
- Better to just store the position of the 1's.

2.2 Inverted Index

```
Brutus    |8|  => 1, 2, 4, 11, 31, 45, 173, 174
Caesar    |8|  => 1, 2, 4, 5, 6, 16, 57, 134
Calpurnia |4|  => 2, 31, 54, 101
```

- For each term t, we store a list of docIDs that contain t
- Requires variable-size postings list. (on disk, continuous run of postings. In memory, linked list or variable length arrays)
- docID inside each term is sorted.
- Construction process:
  1. **Tokenise** into a sequence of (modified token, document ID) pairs.
  2. **Sort** by terms, and then by docID
  3. Multiple term entries in a single document are merged, split into **dictionary** and **postings**, and store **document frequency**.
- Query processing AND:
  - Locate both terms in the dictionary, and merge the two postings.
  - This is $O(n+m)$ if docID are sorted.
  - Optimisation: process in order of increasing frequency

2.3 Boolean Queries: Exact match

- Operator precedence: NOT, AND, OR

3 Postings lists and choosing terms

3.1 Skip pointers

- Done at indexing time.
- Tradeoff:
  - **More skips** -> shorter skip spans -> more likely to skip, bot lots of comparisons.
  - **Fewer skips** -> few pointer comparison, but long skip spans -> few successful skips.
- **Simple heuristic**: for postings of length L, use $\sqrt{L}$ evenly-placed skip pointers.
  - Ignores the distribution of query terms (e.g. 1, 2, 3, 4, 5, 6, 10, 20, 40, 50 has skip pointers 1, 4, 10, 50)
  - Easy if index is relatively static, harder if L keeps on changing because of updates.
  - Used to help , but with modern hardware it may not unless memory-based (because IO cost of loading a bigger postings list can outweigh the gains from quicker in-memory merging)

3.2 Phrase Queries

3.2.1 Biword indexes

- Index every consecutive pair of terms as a phrase.
- 2-word phrase query processing is now immediate.
- Longer phrase processed as booleqn query on biwords. But can have false positives (without the docs, cannot make sure that the biwords appear in that order exactly)
- **Extended Biwords**
  - Parse the text and perform part-of-speech-tagging.
  - Bucket the terms into Nouns (N) and articles/prepositions (X)
  - Call any string of terms of the form `NX*N` an extended biword.
  - Each extended biword is now a term in the dictionary
  - E.g. "catcher in the rye" (NXXN) => "catcher rye"
  - Disadvantages: False positives, Index blowup (bigger dictionary)

3.2.2 Positional indexes

```
<term, no of docs containing term;
doc1: position1, position2, ...;
doc2: position1, position2, ...;
...>
```

- In the postings, store, for each **term** the position(s) that the term appear
- For phrase queries, use a merge algorithm recursively at the document level.
- Query processing:
  - **Extract** the inverted index entries for each distinct term
  - **Merge** their `doc:position` lists to enumerate all positions with the given phrase, e.g. searching for "to be", must look for documents that has "to" in which the position of "be" is exactly afterwards
  - Same strategy for proximity queries.
- Expands postings storage substantially. But standard because of the power and usefulness.
- Rules of thumb: A positional index is 2-4x larger as non-posiitonal index, and around 35-50% of the original text (for English-like languages)
- Can be combined with biwords (e.g. for oft-queried phrases, inefficient to keep on merging positional postings lists)

3.3 Token and terms

3.3.1 Extracting text

Granularity: what unit? file/email/group of files?

3.3.2 Tokenization

- Token = instance of a sequence of characters grouped together as a useful semantic unit
- Issues:
  - Handling apostrophe, hyphens, spaces in proper names, numbers, dates.
  - Language issues: e.g. in French, l'ensemble and un ensemble, the non-segmented German noun compounds, Chinese and Japanese have no spaces between words
  - LTR, RTL langauges.
  - **Stop words**: exclude words from stop list: most common words from the dictionary (little semantic content), but trend is away from doing this (good compression and query optimization can mitigate the size)

3.3.3 Normalization

- Removing dots from abbreviations, hyphens, diacritics.
- Normalization of date forms, (Japanese) kana vs kanji
- **Case folding**
  - Reduce all letters to lower case
  - Disadvantage: C.A.T. vs cat, MIT vs mit (German)
- Alternative to equivalence classing is to do asymmetric expansion (e.g. window => window, windows => Windows, windows, window; Windows => Windows)
- **Thesauri**: handle synonyms and homonyms (car = automobile, color = colour)

3.3.4 Lemmatization

- Reduce inflectional/variant forms to base form with the proper way (linguistically)
- A most modest benefit for retrieval
- For English, mixed results, but definitely useful for Spanish, German, Finnish (30% perf gain for Finnish)

3.3.5 Stemming

- Reduce terms to their roots with crude affix chopping (e.g. automate, automatic, automation => automat)
- **Porter's algorithm**: most common for stemming English
- Other stemmers, e.g. Lovins stemmer (single-pass, longest suffix removal, about 250 rules)

4 Dictionaries and tolerant retrieval

4.1 Hash Table

- Pros: Lookup is faster than for a tree: $O(1)$
- Cons:
  - No easy way to find minor variants
  - No prefix search
  - If vocab keeps growing, need to occasionally perform expensive operation of rehasing **everything**

4.2 Tree

- Requires standard ordering of characters and strings: lexicographical ordering
- Pros: solves the prefix problem (`hyp*`)
- Cons:
  - Slower: $O(\log M)$ and requires a **balanced** tree
  - Rebalancing binary trees is expensive (B-trees mitigate the rebalancing problem)

4.2.1 Binary tree

Simplest, whereby each branch divides by two, and only the leaves store the words.

4.2.2 B-tree

Every internal node has a number of children has a number of children in the interval $[a,b]$ where a,b are appropriate natural numbers.

4.3 Wildcard queries

- With B-tree, simple for `mon*`
- `*mon`: maintain an additional B-tree for the terms reversed.
- `A*B`: intersect `A*` and `*B`
- However, still have to look up the postings for each enumerated term (expensive)

4.4 Permuterm Index

- For "hello", index under: "hello$", "ello$h", "llo$he", "lo$hel", "o$hell", "$hello"
- Queries:
  - `X` lookup on `X$`
  - `*X` lookup on `X$*`
  - `X*Y` lookup on `Y$X*`
  - `X*` lookup on `$X*`
  - `*X*` lookup on `X*`
  - `X*Y*Z` lookup on `$X*` and `Z$*`, AND them and find which of them contains `Y`
- Problem: lexicon size blows up, proportional to average word length

4.5 Bigram (k-gram) indexes

- Enumerate all k-grams (sequence of k chars) occuring in any term (per letter)
- Maintain a second inverted index from bigrams to dictionary terms that match each bigram
- E.g. `mon*` = `$m` AND `mo` AND `on` (usually not useful to add `n$`)
- Prone to false positive, e.g. in this case moon, thus must post filter.
- Fast, space efficient (comapred to permuterm)

4.6 Spelling Correction

4.6.1 Isolated word correction

- Given query, enumerate all character sequences within a preset (weighted) edit distance (e.g. 2), intersect this set with list of correct words.
- Alternatively:
  - Look up all possible corrections in our inverted index and return all docs (slow)
  - We can run with a single most likely correction
- However, computing edit distance to every dictionary term is expensive and slow. Alternative: use ngram overlap

How to define closest?

1. **Edit distance (Levenshtein distance)**
   - Given two strings, the minimum no of ops to convert one to the other.
   - Ops: insert, delete, replace (optionally transposition)
   - **Wagner–Fischer algorithm**:
     Create a matrix whereby $E(i,j) = min\{E(i,j-1)+1, E(i-1,j)+1, E(i-1,j-1)+m\}$ where $m = 1$ if $P_i \neq T_j$, 0 otherwise
2. **Weighted edit distance**
   - As above, but the weight of an operation depends on the characters involved, e.g. keyboard errors, m is more likely to be mistyped as n than as q
   - Requires a weighted matrix as input
3. **Ngram overlap**
   - Enumerate all ngrams in the query strings as well as in the lexicon
   - Use the ngram index (from wildcard search) to retrieve all lexicon terms matching any of the query ngrams
   - Threshold by no of matching ngrams (weight by e.g. keyboard layout, assume initial letter correct, etc.)
   - **Jaccard coefficient** = $\frac{X \cap Y}{X \cup Y}$
     - 1 when X and Y have the same elements, 0 when they are disjoint
     - Always assigns a number between 0 and 1.
     - Use a threshold to decide if you have a match, e.g. above 0.8 is a match
     - To calculate the union, can use $n(X)+n(Y)-n(X \cap Y)$

4.6.2 Context-Sensitive

- Need surrounding context
- Retrieve dictionary terms close (in weighted edit distance) to each query term
- Try all possible resulting phrases with one word corrected at a time
- **Hit-based spelling correction**: suggest the alternative that has lots of hits
- Alternative: break phrase queries into conjunctions of biwords, look for ones that need only one term corrected, enumerate phrase matches and rank them.

4.7 Soundex

- Class of heuristics to expand a query into phonetic equivalents (language specific)
- Invented for the US cencus
- Turn every token to be indexed into a 4-character reduced form
- Build and search an index on the reduced form
- Algorithm: refer to lecture notes
- Not very useful for general IR, okay for high recall tasks, though biased to names of certain nationalities.