CS3245 Cheatsheet AY21/22 Sem 2

by Richard Willie

Language Model

The Unigram Model

- Language as an unorderd collection of tokens.
- Each of the n tokens contributes one count (or $1/n$ probability) to the model.
- Outputs as a count (or probability) of an input based on its individual tokens.

The N-gram Model

- Remembers sequences of n tokens:
  - Unigram is just a special case of $n = 1$.
  - Bigrams are N-gram with $n = 2$.
  - Trigrams where $n = 3$.
- Complexity of N-gram:
  - Let $|V|$ be the size of the vocabulary used in a language.
    - For Unigram, we need to store counts/probability for all $|V|$ words.
    - For Bigrams, we need to store counts/probability for all $|V| \cdot |V| = |V|^2$ ordered length of 2 phrases.
  - In general, the space complexity of N-gram is $|V|^N$, i.e. it gets expensive very quickly as N increases.

Markov Assumption

- The presumption that the future behavior of a dynamical system only depends on its recent history. In particular, in a k^{th}-order Markov model, the next state only depends on the k most recent states.
- An N-gram model is a $(N-1)$-order Markov model.
- Let N be the size of an n-gram. The probability of a given word w_n appearing as the next sequence is:

$$\Pr(w_n) = \Pr(w_n | w_{n-N+1:n-1}) = \frac{C(w_{n-N+1:n-1} w_n)}{C(w_{n-N+1:n-1})}$$

Smoothing

- To keep the language model from assigning zero probability to unseen events, we'll have to shave off a bit of probability mass from some more frequent events and give it to the events we've never seen. This modification is called **smoothing** or **discounting**.
- **Add-k smoothing**:
  - Adds k count to all entries in the LM, including those that are not seen.
  - The unigram probability after add-k smoothing:

$$\Pr(w_i) = \frac{C(w_i) + k}{N + kV}$$

  - The n-gram probability after add-k smoothing:

$$\Pr(w_n | w_{n-1}) = \frac{C(w_{n-1} w_n) + k}{C(w_{n-1}) + kV}$$

- **Add-one smoothing** (also called Laplace smoothing) is just a special case of $k = 1$.
- Applied to
  - **small** number of observations and **large** vocabulary space: **poor** choice because we're shifting the probability mass quite drastically. For example, English alphabet, only observe "a", then Pr(a) = 1, but with add-one smoothing, Pr(a) = 2/27 = 0.074.
  - **large** number of observations and **small** vocabulary space: **good** choice because we're only shifting a small probability mass while achieving the guarantee of non-zero probability for all events.

Boolean Retrieval

- Suppose the collection is 40 plays by Shakespeare.
- The information needed is which plays of Shakespeare contain the words Brutus and Caesar but not Calpurnia?
- The Boolean query: `Brutus AND Caesar AND (NOT Calpurnia)`.

Naive Approach

- For each play, run `CTRL+F` for Brutus, Caesar, and Calpurnia separately.
- If there is at least one match for Brutus, at least one match for Caesar, but none for Calpurnia, add this play to the result.
- Disadvantage: too slow!

Term-Document Incidence Matrix

- The index:

	Julius Caesar	Hamlet	Othello	Macbeth
Brutus	1	1	0	0
Caesar	1	1	1	1
Calpurnia	1	0	0	0

- Query processing: Take the rows for Brutus, Caesar, and NOT Calpurnia and bitwise-AND them.

	Julius Caesar	Hamlet	Othello	Macbeth
Brutus	1	1	0	0
Caesar	1	1	1	1
NOT Calpurnia	0	1	1	1
AND	0	1	0	0

- Disadvantage: big and sparse matrix.

Inverted Index

```
Brutus => 1, 2, 4, 11, 31, 45, 173, 174
Caesar => 1, 2, 4, 5, 6, 16, 57, 132, ...
Calpurnia => 2, 31, 54, 101
```

- For each term t, we store a list of docIDs that contain t.
- Requires variable-size postings list.
  - In memory, use linked lists or variable-length arrays.
  - On disk, store a continuous run of postings.
- Index construction:
  1. Generate a sequence of (term, docID) pairs.
  2. Sort the sequence by terms and then docIDs.

3. Multiple term entries in a single document are merged, split into dictionary and postings, and store document frequencies.

- Query processing:
  - `AND`
    - Example query: `Brutus AND Caesar`
      1. Locate Brutus in the dictionary and retrieve its postings.
      2. Locate Caesar in the dictionary and retrieve its postings.
      3. Find the intersection of the two postings.
    - Time complexity: $\Omega(\min(m,n))$ and $O(m+n)$
    - $0 \leqslant$ Length of intersection $\leqslant \min(m,n)$
  - `OR`
    - Example query: `Brutus OR Caesar`
      1. Locate Brutus in the dictionary and retrieve its postings.
      2. Locate Caesar in the dictionary and retrieve its postings.
      3. Find the union of the two postings.
    - Time complexity: $\Theta(m+n)$
    - $\max(m,n) \leqslant$ Length of union $\leqslant m+n$
  - `NOT`
    - Example query: `NOT Brutus`
      1. Locate Brutus in the dictionary and retrieve its postings.
      2. Keep all entries that appear in the full list (contains N docID(s)) but not in the postings.
    - Time complexity: $\Theta(N)$
- Query optimization: process in order of frequency.

Postings List and Choosing Terms

Skip Pointers

- Done at indexing time.
- Tradeoff:
  - More skips $\rightarrow$ shorter skip spans $\rightarrow$ more likely to skip, but lots of comparisons.
  - Fewer skips $\rightarrow$ few pointer comparisons, but long skip spans $\rightarrow$ few successful skips.
- Simple heuristic: for postings of lengh L, use $\sqrt{L}$ evenly-spaced skip pointers.

Phrase Queries

- Want to be able to answer queries such as "stanford university" – as a phrase.
- For example, "I went to Stanford University" is a match, but "I went to university at Stanford" is not.
- **Biword Indexes**:
  - Index every consecutive pairs of terms in the text.
  - For example, "I went to Stanford University" has 4 biwrods: "I went", "went to", "to Stanford", "Stanford University".
  - Process the two-word phrase queries by looking up the biwords directly.
  - Longer phrase queries:
    - Processed as Boolean queries on biwords.
    - For example, "stanford university palo alto" is processed as "stanford university" AND "university palo" AND "palo alto".
    - Disadvantage: can have false positives.
  - Extended biwords:
    - Index all extended biwords, in the form NX*N, where N = Noun, X = Articles/Prepositions.
    - For example, "catcher in the rye" has 1 extended biword: "catcher rye".
    - Process phrase queries by extracting and looking up the extended biwords.
  - Issues:
    - False positives.
    - Index blowup due to bigger dictionary (infeasible for more than biwords).
- **Positional Indexes**:

```
<term document frequency;
doc1: position1, position2, ...;
doc2, position2, position2, ...;
...>
```

  - In the postings, store, for each term the position(s) in which tokens of its 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.
  - Rules of thumb: A positional index is 2-4x larger than a non-positional index, and 35-50% the volume of original text.
- **Combining Biword and Positional Indexes**:
  - Merging is slow in positional indexes.
  - Possible enhancement: index popular bi-word based on the query log.
    - For example. "Michael Jackson", "Britney Spears".
    - Retrieve the postings without merging (at the cost of some additional storage).

Choosing Terms

- **Tokenization**:
  - A token is an instance of sequence of characters grouped together as a useful semantic unit.
  - Issues:
    - Handling apostrophe, hyphens, spaces in proper names, numbres, dates.
    - Language issues:
      - French: *l'ensemble* is one token or two? How about *l'ensemble* and *un ensemble*?
      - German noun compounds are not segmented.
      - Chinese and Japanese have no spaces between words.
      - Japanese is written in multiple systems, e.g. Hiragana, Karakana, Kanji.
      - Arabic is written right to left.
  - **Stop word removal**: exclude common words, e.g. *the*, *a*, *and*, *to*, *be*.
  - **Normalization**:

    - Removing dots from abbreviations.
    - Deleting hyphens to form a term.
    - Removing accents and diacritics.
    - **Case-folding**: reduce all letters to lower case.
  - **Lemmatization**:
    - Reduce inflectional/variant forms to base form properly (linguistically).
    - For English, mixed results, but definitely useful for Spanish, German, Finnish.
  - **Stemming**:
    - Reduce terms to their roots with crude affix chopping (e.g. automate, automatic, automation → automat).
    - **Porter's algorithm**: most common algorithm for stemming English.

Dictionaries and Tolerant Retrieval

Dictionary Data Structures

- **Hash table**:
  - Pros: Lookup is $O(1)$.
  - Cons:
    - No easy way to find minor invariants (e.g. judgment or judgement).
    - No prefix search.
    - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing everything.
    - Overall, not very tolerant!
- **Tree**:
  - Requires a standard ordering of characters, and we have one: lexicograhical ordering.
  - Pros:
    - Solves the prefix problem.
  - Cons:
    - Slower: $O(\log M)$ and this requires a balanced tree.
    - Rebalancinng binary trees is expensive (B-trees mitigate the rebalancing problem).

Wildcard Queries

- Maintain a B-tree for terms.
- For `A*`, find docs with words beginning with "mon".
- For `*B`, maintain an additional B-tree for terms reversed.
- For `A*B`, intersect `A*` and `*B`.
- Disadvantage: still have to lookup the postings for each enumerated term (expensive).

Permuterm Index

- For the term "hello", add an end marker $ and index all rotations: "hello$", "ello$h", "llo$he", "lo$hel", "o$hell", "$hello".
- For wildcard 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 one of them contains `Y`

Bigram (k-gram) Index

1. Enumerate all k-grams (sequence of k chars) occurring in any term.
2. Maintain a second inverted index from bigrams to dictionary terms that match each bigram.

```
m => mace, madden, ...
mo => among, amortize, ...
on => among, axon, ...
```

3. For example, the query `mon*` can be run as `$m AND mo AND on`.
4. Possible matches: `month`, `moon`, ...
5. Disadvantage: false positives, e.g. `moon` contains all 3 bigrams `$m`, `mo`, `on`. Must post-filter these terms against query.
6. Advantage: fast, space efficient compared to permuterm.

Spelling Correction

- **Isolated Word Correction**:
  - Fundamental premise – there is a lexicon from which the correct spellings come.
  - Given a lexicon and a character sequence Q, return the words in the lexicon *closest* to Q, e.g. dof → dog, dock, cat, ...?
  - How do we define *closest*?
    - **Edit Distance (Levenshtein Distance)**:
      - Given two strings, the minimum number of operations to convert one to the other.
      - Operations: insert, delete, replace, (optionally transposition).
      - Computing edit distance: $E(i,j) = \min(E(i,j-1)+1, E(i-1,j)+1, E(i-1,j-1)+m)$, where $m=1$ if $P_i \neq T_j$, 0 otherwise.
    - **Ngram Overlap**:
      - Enumerate all ngrams in the query as well as in the lexicon.
      - Count the overlaps between a pair of terms.
      - Threshold to decide if you have a match, e.g. count $\geqslant 2$.
      - Disadvantage: favors longer terms by nature.
      - Alternative: **Jaccard Coefficient**, $\mathrm{JC}(X,Y) = \frac{X \cap Y}{X \cup Y}$.
- **Context-Sensitive Correction**:
  - Need surrounding context.
  - Retrieve dictionary terms close to each query term.
  - Enumerate all possible resulting phrases with one word corrected at a time.
    - flew *from* Narita
    - *fled* form Narita
    - flew form *Arita*
  - Decide which ones to present using heuristics, e.g. **hit-based spelling correction**.

Soundex

- A class of heuristics to expand a query into phonetic equivalentd (language specific).
- Turn every token (in both query and documents) into a 4-character reduced form.
- Build and search and index on the reduced forms.
- Algorithm:
  1. Retain the first letter of the word.
  2. Change all occurrences of the following letters to '0' (zero): 'A', 'E', 'I', 'O', 'U', 'H', 'W', 'Y'.
  3. Change all letters to digits as follows:
     - B, F, P, V $\rightarrow$ 1
     - C, G, J, K, Q, S, X, Z $\rightarrow$ 2
     - D, T $\rightarrow$ 3
     - L $\rightarrow$ 4
     - M, N $\rightarrow$ 5
     - R $\rightarrow$ 6
  4. Repeatedly remove one out of each pair of consecutive identical digits.
  5. Remove all zeros from the resulting string.
  6. Pad the resulting string with trailing zeros and return the first four positions, which will be of the form <uppercase letter><digit><digit><digit>.
- Not very useful for general IR, okay for high recall tasks, though biased to names of certain nationalities.

Index Construction

Blocked Sort-Based Indexing (BSBI)

```
BSBIndexConstruction()
    n <- 1
    while (all documents have not been processed)
    do n <- n + 1
        block <- ParseNextBlock()
        BSBI-Invert(block)
        WriteBlockToDisk(block, fn)
    MergeBlocks(f1, ..., fn; fmerged)
```

Single-Pass In-Memory Indexing (SPIMI)

```
SPIMI-Invert(token_stream)
    output_file = NewFile()
    dictionary = NewHash()
    while (free memory available)
    do token <- next(token_stream)
        if term(token) not in dictionary
            then postings_list =
                AddToDictionary(dictionary, term(token))
            then postings_list =
                GetPostingsList(dictionary, term(token))
        if full(postings_list)
            then postings_list =
                DoublePostingsList(dictionary, term(token))
        AddToPostingsList(postings_list, docID(token))
    sorted_terms <- SortTerms(dictionary)
    WriteBlockToDisk(sorted_terms, dictionary, output_file)
    return output_file
```

- Merging of blocks is analogous to BSBI.
- Faster than BSBI:
  - No sorting of pairs.
  - Only sorting of dictionary terms.

Distributed Indexing

- Architecture:
  - Maintain a *master* machine directing the indexing job.
  - Breakup indexing into sets of parallel tasks.
  - Master machine assigns each task to an idle worker machine from a pool.
  - Use two sets of parallel tasks: parser and inverter.
  - Parser:
    - Master assigns a split ot an idle parser machine.
    - Parser reas a docuyment at a time and emits (term, docID) pairs.
    - Parser writes pairs into j partitions, each partition is for a range of terms' first letters.
  - Inverter:
    - Inverter collects all (term, docID) pairs (= postings) for one term-partition.
    - Sort and writes to postings lists.
- The algorithm describe above is an instance of **MapReduce**, a robust and simple framework for distributed computing.
- Schema for map and reduce functions:
  - map: input $\rightarrow$ list(k, v)
  - reduce: (k, list(v)) $\rightarrow$ output
- Schema for index construction:
  - map: web collection $\rightarrow$ list(term, docID)
  - reduce: (<term 1, list(docID)>, <term 2, list(docID)>, ...) $\rightarrow$ (postings list 1, postings list 2, ...)

Dynamic Indexing

- In practice, static collections are rare.
- Simplest, naive approach: re-index every time (impractical).
- 2nd simplest approach:
  - Keep two indexes: one big main index I, and one small (in memory) auxiliary index Z.
  - Mechanism:
    - Add: new docs goes to the auxiliary index.
    - Delete: maintain a list of deleted docs.
    - Update: delete + add.
    - Search: search both, merge results and omit deleted docs.
  - Need to perform merge when auxiliary index is too large:
    - **Linear merge**:
      - Algorithm: Once Z is full, write out Z and merge with I.

- Cost of merging: $O(nk^2) = O(T^2)$, where T is the number of (term, docID) pairs which require k merges, e.g. $k = T/n$.

* **Logarithmic merge**:
  - Idea: maintain a series of indexes
  - Z_0: in memory, with the same capacity as I_0 $(= n)$.
  - I_0, I_1, ...: on disk, each twice as large as the previous one.
  - Example:

	I_0	I_1	I_2
0	0	0	0
n	1	0	0
$2n$	0	1	0
$3n$	1	1	0
$4n$	0	0	1

  - Cost of merging: each merging is touched $O(\log T)$ times, so complexity is $O(T \log T)$.
  - Disadvantage: query processing is slower, merging from $O(\log T)$ indexes (as compared to 2).

Index Compression

Heaps' Law

$$M = kT^b$$

- M is the size of the vocabulary, T is the number of tokens in the collection.
- Typical values: $30 \leqslant k \leqslant 100$ and $b \approx 0.5$.
- An empirical finding, i.e. not based on scientific law.
- In a log-log plot of vocabulary size M vs. T, Heaps' Law predicts a line with slope about 1/2.

Zipf's Law

$$\text{cf}_i = \frac{K}{i}$$

- cf = collection frequency, the number of occurrences of a term in the collection, not the same as document frequency (df).
- cf_i is the cf of the i-th most frequent term.
- K is a normalizing constant, $\text{cf}_1 = K/1 = K$.

Dictionary Compression

- **Fixed-width entries**:
  - Most of the bytes in the term column are wasted.
  - Cannot handle long terms.
- **Dictionary-as-a-String**:
  - Store dictionary as a long string of characters.
  - Pointer to the next word shows end of current word.
- **Blocking**:
  - Store pointers to every kth term string.
  - Need to store term lengths (1 extra byte).
- **Front Coding**:
  - Sorted words commonly have long common prefix – store differences only.

Postings File Compression

- **Gap Encoding**:
  - Stores the gaps of the docIDs.
- **Variable Byte Encoding**:
  - Observation: it is wasteful to use a fixed number of bits to store every number.
  - Key challenge: encode every integer (gap) with about as little space as needed for that integer.
  - Algorithm:
    1. Begin with one byte to store a gap G and dedicate 1 bit in it to be a continuation bit c (0 = not ending, 1 = ending).
    2. If $G \leqslant 127$, binary-encode it in the 7 available buts and set $c = 1$.
    3. Else encode G's lower-order 7 bits and then use additional bits to encode the higher order bits using the same algorithm.
    4. At the end set the continuation bit of the last byte to 1 ($c = 1$) – and for other bytes $c = 0$.

Vector Space Model

Term frequency: tf

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

Inverse document frequency: idf

$$\text{idf}_t = \log_{10}(N/\text{df}_t)$$

tf-idf weighting

- The tf-idf weight of a term is the product of its tf weight and its idf weight.

$$w_{t,d} = (1 + \log_{10}(\text{tf}_{t,d})) \cdot \log_{10}(N/\text{df}_t)$$

- Variants:

Term frequency		Document frequency		Normalization	
n (natural)	$\text{tf}_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(\text{tf}_{t,d})$	t (idf)	$\log \frac{N}{\text{df}_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2 + w_2^2 + ... + w_M^2}}$
a (augmented)	$0.5 + \frac{0.5 \times \text{tf}_{t,d}}{\max_t(\text{tf}_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N - \text{df}_t}{\text{df}_t}\}$	u (pivoted unique)	$1/u$ (Section 6.4.4)
b (boolean)	$\begin{cases} 1 & \text{if } \text{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1/\mathit{CharLength}^{\alpha}$, $\alpha < 1$
L (log ave)	$\frac{1 + \log(\text{tf}_{t,d})}{1 + \log(\text{ave}_{t \in d}(\text{tf}_{t,d}))}$				

Evaluation

Unranked Retrieval Evaluation

- **Precision**: fraction of retrieved docs that are relevant.
- **Recall**: fraction of relevant docs that are retrieved.
- In a good system, precision decreases as either the number of docs retrieved or recall increases.
- **A combined measure: F**
  - Combined measure that assesses the precision/recall tradeoff is F measure.

$$F = \frac{1}{\alpha \frac{1}{P} + (1 - \alpha) \frac{1}{R}} = \frac{(\beta^2 + 1)PR}{\beta^2 P + R}$$

  - People usually use F_1 measure:

$$F_1 = \frac{2PR}{P + R}$$

Evaluating Ranked Results

- Relevant documents should be ranked higher than non-relevant documents.
- A precision-recall curve can be drawn by computing precision at different recall levels, i.e. every time a relevant document is retrieved.
- Example:
  - For a collection of 5 docs 1, 2, 3, 4, 5, and a query, 3 docs 1, 2, 3 are relevant. A system returns 5 docs in the order of 3, 4, 5, 1, 2.
  - The data points in the form of (R, P) are:
    - (0.33, 1) when doc 3 is retrieved
    - (0.66, 0.5) when doc 1 is retrieved
    - (1, 0.6) when doc 2 is retrieved
- **Interpolated precision**:
  - Sometimes precision does increase with recall locally. This should be accounted for since the precision is not as bad as it seems at the low point.
  - We take the max of precisions to the right of the value.
  - Example:
    - Raw data points: (0.33, 1), (0.66, 0.5), (1, 0.6)
    - Interpolated data points: (0.33, 1), (0.66, 0.6), (1, 0.6)

Test Collections for Evaluation

- **Kappa measure**:

$$\text{Kappa}(K) = \frac{P(A) - P(E)}{1 - P(E)}$$

  - Agreement measure among judges.
  - $P(A)$ – proportion of time judges agree
  - $P(E)$ – what agreement would be by chance
  - $P(E) = P(\text{non-relevant})^2 + P(\text{relevant})^2$
- **A/B Testing**:
  - Purpose: Test a single innovation, i.e. change.
  - Prerequisite: You have a large search engine up and running.
  - Idea:
    - Have most users use old system, bust divert a small proportion of traffic (e.g., 1%) to the new system with the innovation.
    - Evaluate with an "automatic" **Overall Evaluation Criterion (OEC)** like clickthrough on first result.

Relevance Feedback and Query Refinement

Relevance Feedback

- User provides explicit feedback: Standard RF
- Implicit feedback: Clickstream mining
- No feedback:
  - Pseudo RF
  - Blind Feedback
- **Query Refinement**:
  - q_0 = initial query
  - D_r = a (small) set of known relevant doc vectors
  - D_nr = a (small) set of known irrelevant doc vectors
  - q_m = modified query
  - Centroid = the center of mass of a set of documents

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

  - Rocchio:

$$\vec{q}_m = \alpha \vec{q}_0 + \beta \frac{1}{|D_\text{r}|} \sum_{\vec{d}_j \in D_\text{r}} \vec{d}_j - \gamma \frac{1}{|D_\text{nr}|} \sum_{\vec{d}_j \in D_\text{nr}} \vec{d}_j$$

  - α, β, γ = weights (hand-chosen or set empirically)
  - Term weight in the query vector can go negative.
- When does RF work? When two assumptions hold:
  - User's initial query at least partially works.
  - (Non)-relevant documents are similar or term distribution in non-relevant documents are sufficiently distinct from relevant documents.
- **Pseudo Relevance Feedback**:
  - Blind feedback automates the "manual" part of true RF, by assuming the top K is actually relevant.
  - Algorithm:
    1. Retrieve a ranked list of hits for the user's query.
    2. Assume that the top k documents are relevant.
    3. Do relevance feedback.
  - Works well on average.
  - But can go horribly wrong for some queries.
  - Several iterations can cause query drift.

Query Expansion

- For each query term, expand it with the related words of t from a thesaurus.
  - The thesaurus can be manually compiled or automatically generated.
- Examples:
  - feline → feline cat
  - interest rate → interest rate fascinate evaluate
- Generally increases recall, but may decrease precision when terms are ambiguous.
- Manually compiled thesaurus: MeSH, WordNet.
- Automatic Thesaurus Generation:
  - You can "harvest", "peel", "eat" and "prepare" apples an pears, so apples and pears must be similar.
  - Generate a thesaurus by analyzing documents.
  - Assumption: distributional similarity
    - Two words are similar if they co-occur/share the same grammatical relations with similar words.
    - Co-occurrences are more robust; grammatical relations are more accurate.

XML Retrieval

Structured Retrieval

- Premise: queries are structured or unstructured; documents are structured.
- **XLM Document**:
  - Internal nodes encode document structure or metadata.
  - An element can have one or more attributes and sub elements.
  - Leaf nodes consist of text.
  - Possible queries which match (with part) of this document: Macbeth scene/title#castle
- Common problems:
  - What is the unit of retrieval? e.g. the whole document or just an element?
  - How to rank the items in the result list?
  - How to evaluate the retrieval performance?

Vector Space Model for XML IR

- Aim: A dimension in vector space encode = a word + its position within the XML tree.
- How:
  1. Take each text node (leaf) and break it into multiple nodes, one for each word, e.g. split Bill Gates into Bill and Gates.
  2. Extract the lexicalized subtrees, which are subtrees that contain at least one vocabulary term.
- Compromise:
  - Index all paths that end in a single vocabulary term, i.e. all XML-content term pairs.
  - Each pair is a structural term and denote it by $< c, t >$: a pair of XML-content c and vocabulary term t.
- **Context resemblance**:
  - A simple measure of the similarity of a structural term c_q in a query and a structural term c_d in a document.

$$CR(c_q, c_d) = \begin{cases} \frac{1+|c_q|}{1+|c_d|}, & \text{if } c_q \text{ matches } c_d \\ 0, & \text{if } c_q \text{ does not match } c_d \end{cases}$$

  - c_q matches c_d if and only if we can transform c_q into c_d by inserting additional nodes.
- **Document Similarity Measure**:
  - The final score for a document is computed as a variant of the cosine measure, which we call **SimNoMerge**.

$$\text{SimNoMerge}(q, d) = \sum_{c_k \in B} \sum_{c_l in B} CR(c_k, c_l) \sum_{t \in V} \text{weight}(q, t_k, c_k) = \frac{\text{weight}(d, t, c_l)}{\sqrt{\sum_{c \in B, t \in V} \text{weight}^2(d, t, c)}}$$

XML IR Evaluation

- **Component Coverage**:
  - Evaluates whether the element retrieved is "structurally" correct, i.e. neither too low not too high in the tree.
  - Four cases:
    - Exact coverage (E)
    - Too small (S)
    - Too large (L)
    - No coverage (N)
- **Topical Relevance**:
  - Four levels:
    - Highly relevant (3)
    - Fairly relevant (2)
    - Marginally relevant (1)
    - Nonrelevant (0)
- The relevance-coverage combinations are quantized as follows:

$$Q(\text{rel}, \text{cov}) = \begin{cases} 1.00, & \text{if (rel, cov)} = \text{3E} \\ 0.75, & \text{if (rel, cov)} \in \text{2E, 3L, 3S} \\ 0.50, & \text{if (rel, cov)} \in \text{1E, 2L, 2S} \\ 0.25, & \text{if (rel, cov)} \in \text{1S, 1L} \\ 0.00, & \text{if (rel, cov)} = \text{0N} \end{cases}$$

- The number of relevant components in a retrieved st A can be computed as:

$$\#(\text{relevant items retrieved}) = \sum_{c \in A} Q(\text{rel}(c), \text{cov}(c))$$

- Example: if the system returns 5 item which are assessed as 3E, 3E, 0N, 1E, 1S, the precision is (1 + 1 + 0 + 0.5 + 0.25) / 5 = 0.55.