CS3245

Information Retrieval

AY2022/23 Semester 2

Notes by Jonathan Tay

Last updated on April 15, 2023

Contents

Part I

Boolean Retrieval

1 Language Models

A **language model** is a grammarless, computational model created from collections of text.

They are used to assign scores (e.g. probabilities) to a sequence of words.

unigram model

Create a **frequency table** of all tokens (words) that appear in the collection.

Unigram models have insufficient context to model the order of words in a sentence.

***n*-gram model**

By remembering sequences of n tokens we can predict the n-th token given only the previous $n-1$ tokens as context (**Markov assumption**).

A unigram model is a 1-gram model, bigram model is a 2-gram model, etc.

However, n-gram models require exponentially more space as n increases.

The **count** of an input is the *sum* of the counts of all tokens in the input, while the **probability** of an input is the *product* of the probabilities of all tokens in the input.

However, if a token does not appear in the collection, its probability is 0, resulting in a probability of 0 for the entire input, which is undesirable.

1-smoothing is a technique to avoid this problem. It adds a count of 1 to every token in the collection, even if it does not appear in the input.

2 Index Compression

Index compression decreases the disk space required, increases the speed of postings lists transfer from from disk to memory, and increases the amount of data that can be stored in memory.

Heaps' law

$M = kT^b$, where

M is the vocabulary size (number of distinct terms in the dictionary), T is the number of tokens in the collection, and $30 \leq k \leq 100$ and $b \approx 0.5$.

Heaps' law is used to predict the size of the dictionary given the size of the collection.

Next, we define the **collection frequency (cf)** of a term to be the number of times that it appears in the collection.

This is different from (but positively correlated with) the **document frequency (df)** of a term, which is the number of documents that contain the term.

Zipf's law

$\mathrm{cf}_i = \frac{K}{i}$, where

cf_i is the collection frequency of the i-th most frequent term, and K is a normalizing constant (typically $K = \mathrm{cf}_1$).

Zipf's law is used to predict the collection frequency of a term given its rank.

In general, there are a few very common terms and very many rare terms.

2.1 Dictionary Compression

Every search begins with the dictionary, so keeping its memory footprint small is important.

Each entry in an uncompressed dictionary is typically of the form <term, df, postings pointer>.

However, this means every term has a fixed-width, which not only limits which the maximum length of a term that can be stored, but also wastes space for short terms.

dictionary-as-a-string

Every term in the dictionary is concatenated into a single string.

Each entry in the dictionary is then of the form <term pointer, df, postings pointer>, where each term pointer points to the start of each term.

blocking

Let k be the number of terms in a block.

For each term in a block, prefix it with its length (number of characters).

Then, only the first term in each block retains a term pointer, while the remaining terms are retrieved by adding the length of all the previous terms to the term pointer of the first term in the block.

e.g.:

```
...7systile9syzygetic8syzygial6syzygy...
...^ptr
```

front-coding

Sorted words typically have a long common prefix, so we can take advantage of this by storing the common prefix once and then only storing the suffix for each word.

An asterisk $*$ marks the end of the prefix and the start of the first suffix.

The remaining $k-1$ suffixes in the block are each prefixed with their length, followed by a diamond $\diamond$, then the suffix itself.

e.g.

```
8automata8automate9automatic10automation
8automat∗a1⋄e2⋄ic3⋄ion
```

2.2 Postings File Compression

The postings file contains document IDs which can grow to large integers.

By Zipf's law a small number of terms have very high cfs, which implies high dfs, and as such the gaps should be small.

gap encoding

After the first document ID, subsequent values are the difference between the current document ID and the previous document ID.

However, it's still inefficient to use the same number of bits to store a small gap versus a large gap.

Therefore, we want to minimize the number of bits used to store the gaps.

variable byte encoding

The smallest unit of data is still the byte, of which the first bit will be used as a **continuation bit**.

If the continuation bit is 0, then the next byte is part of the same integer. Otherwise, the byte is the last (least significant) byte of the integer.

Therefore, each byte can only store 7 bits of data, so we need to use multiple bytes to store large integers.

The postings list will be every single byte from the encoded gaps, concatenated together.

e.g.

```
5 => 10000101
214577 => 00001101 00001100 10110001
```

Part II

Ranked Retrieval

In Boolean retrieval, documents are either included or excluded from the result based on whether they contain the query terms.

While accurate, this can yield too many results without any consideration for the relevance of the documents.

3 Vector Space Model

In ranked retrieval, documents are scored from $[0, 1]$ based on how well the document matches the query, and sorted thereafter to yield only the most relevant.

scoring: Jaccard coefficient

Let A be the set of terms in the query and B the set of terms in the document.

The Jaccard coefficient is defined as:

$$\begin{aligned}
&\mathrm{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|} && \text{if } A \cap B \neq \varnothing \\
&\mathrm{Jaccard}(A, B) = 0 && \text{if } A \cap B = \varnothing \\
&\mathrm{Jaccard}(A, A) = 1
\end{aligned}$$

However, the Jaccard coefficient consider neither the **term frequency (tf)** nor the **document frequency (df)**.

A document which contains the query terms more times should be scored higher than a document which contains the query terms fewer times, but the Jaccard coefficient assumes $tf = 1$.

Furthermore, rare terms are more informative than frequent terms, so we introduce the **inverse document frequency (idf)** weighting scheme.

tf-idf weighting

Let $tf_{t,d}$ be the number of times term t appears in document d.

Also, let df_t be the number of documents which contain t, and N the number of documents in the collection.

Then, $idf_t = \log_{10}\left(\frac{N}{df_t}\right)$.

Then, define the log-frequency weight $w_{t,d}$ as:

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

Putting the two together, the tf-idf weight ($tf.idf_{t,d}$)
$= w_{t,d} \times idf_t$
$= \left(1 + \log_{10} tf_{t,d}\right) \times \log_{10}(N \div df_t)$.

As such, the tf-idf weight increases with both the number of occurrences of a term in a document *and* its rarity in the collection.

scoring: tf-idf

Finally, for each term t in both the query q and document d, we define the term score as:

$$\text{Score}(q,d) = \sum_{t \in (q \cap d)} tf.idf_{t,d}$$

vector space model

Documents and queries can be represented in a **tf-idf matrix**, where each document is a column vector of tf-idf weights for each term.

These $|V|$-dimensional column vectors are sparse.

Therefore, we want to rank documents $\overrightarrow{d_i}$ in increasing order of their **cosine similarity** to the query vector $\overrightarrow{q}$.

In other words, the smaller the angle between $\overrightarrow{d_i}$ and $\overrightarrow{q}$, the more similar they are.

We do not rank by Euclidean distance as the distance between $\overrightarrow{d_i}$ and $\overrightarrow{q}$ is large even if their term distributions are similar.

scoring: cosine similarity

$$\cos(\overrightarrow{d}, \overrightarrow{q}) = \frac{\overrightarrow{q} \cdot \overrightarrow{d}}{|\overrightarrow{q}||\overrightarrow{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

The square root terms performs **length normalization** so that weights are comparable across different vectors even if they have different lengths.

Queries and documents may have different weighting schemes, in the form $ddd.qqq$.

lnc.ltc

document: logarithmic *tf*, no *idf*, with cosine normalization.

query: logarithmic *tf*, with *idf* and cosine normalization.

We avoid using *idf* for documents as insertion of new documents would require recomputation of the *idf* for all terms in that document, and normalization for all existing documents, which is inefficient.

Part III

Calculation Reference

4 Boolean Retrieval

```
# Given a sequence of tokens A and B
# and a query Q, construct 2 LMs from A and B:
lm_a = dict(term -> frequency)
lm_b = dict(term -> frequency)

# Counting model:
def count(lm, query_term):
    if query_term not in lm:
        return 0
    return lm[query_term]
    # for multiple terms, take the sum

# Probability model:
def probability(lm, query_term):
    if query_term not in lm:
        return 0
    return lm[query_term] / sum(lm.values())
    # for multiple terms, take the product

# Add-1 smoothing:
def add_one(lm, collection):
    for term in collection:
        if term in lm:
            lm[term] += 1
        else:
            lm[term] = 1
    # terms outside of the collection are given
    # a count of 0

# Bigrams:
def bigram_probability(lm, "foo bar baz"):
    # unigram style
    return probability(lm, "foo bar") *
        probability(lm, "bar baz")
    # true bigram style
    return probability(lm, "foo") *
        probability(lm, "bar" | "foo") *
        probability(lm, "baz" | "bar")
    # | denotes conditional probabilities
```

```
# Indexing:
def index(collection):
    # generate sequence of pairs (term, doc_id)
    pairs = []
    for document in collection:
        for term in document:
            pairs.append((term, document.id))
    # sort the pairs by term then doc_id
    # alphabetical order is used for sorting
    pairs.sort()
    # create the index
    index = dict(term -> df, postings(term))
    postings(term) = list(doc.id if term in doc)
    return index

# Query processing:
def and_query(foo, bar): # foo AND bar
    posts_foo, df_foo = index[foo]
    posts_bar, df_bar = index[bar]
    # merge the postings lists
    ptr_foo, ptr_bar = &posts_foo[0], &posts_bar[0]
    result = []
    while not past the end of either postings:
        if *ptr_foo == *ptr_bar:
            result.append(*ptr_foo)
            ptr_foo++
            ptr_bar++
        elif *ptr_foo < *ptr_bar:
            ptr_foo++
        else:
            ptr_bar++
    # perf: evaluate terms in increasing df
    # perf: use skip pointers

# Query pre-processing:
def preprocess(query):
    query.remove_stop_words()
    query.remove_punctuation() # normalization
    query = query.lower() # case folding
    query.stem() or query.lemmatize()
    query.add_positional_indices()

def extended_biwords(query):
    for every phrase starting with a noun and
        ending with a noun:
        yield phrase
```

```
def add_skip_pointers(postings):
   interval = floor(sqrt(len(postings)))
   for i in 1..interval:
      from = postings[interval * (i - 1)]
      to = postings[interval * i]
      from.skip = &to

# Query positional index:
def phrase_appears?(query, index):
   for document with all terms in query:
      # let query be the phrase "foo bar"
      # get the position lists
      pos_foo, tf_foo = index["foo"][document.id]
      pos_bar, tf_bar = index["bar"][document.id]
      while in both postings list:
         if "foo" is the n-th word and
            "bar" is the (n+1)-th word:
            return true
         else:
            advance the pointer into the position
            list with the smaller doc.id
   return false

   # Permuterm index:
   def permuterms(term):
      # add the end token ‘$‘
      term += ’$’
      rotations = [term]
      while not term.starts_with(’$’):
         move first character to the end
         terms.append(term)
      return rotations

   assert permuterm("motion") == [
      "motion$", "otion$m", "tion$mo",
      "ion$mot", "on$moti", "n$motio", "$motion"
   ]

   # Wildcard query:
   def wildcard_permuterm(term):
      assert ’*’ in term # wildcard
      term += ’$’ # end token
      while not term.ends_with(’*’):
         move first character to the end
      return term

assert wildcard_permuterm("mo*on") == "on$mo*"
# will match with any term with a permuterm
# starting with the same prefix "on$mo"

# K-gram index:
def k_grams(term, k):
"""
Instead of indexing every possible permuterm,
we index only the first k-grams of every
permuterm, preventing dictionary size from
exploding.
"""
   for permuterm in permuterms(term):
      yield permuterm[:k]

assert k_grams("motion", 2) == [
   "mo", "ot", "ti", "io", "on", "n$", "$m"
]

def in_wildcard_k_gram_index?(term, index):
"""
May have false positives.
"""
   for k_gram in k_grams(term, 2):
      if k_gram not in index or
         term not in index[k_gram]:
         return false
   return true


# Edit distance:
def edit_distance(A, B):
"""
Returns the number of insertions, deletions,
or substitutions to transform A into B.
"""
   # add empty strings (will affect length)
   A = ’_’ + A
   B = ’_’ + B
   M = Matrix(cols = len(A), rows = len(B))
   # dynamic programming
   for every column i and row j:
      up = M[i][j - 1] + 1
      left = M[i - 1][j] + 1
      up_left = M[i - 1][j - 1] +
         1 if A[i] != B[j] else 0
```

```python
            # discard out of bounds
            M[i][j] = min(up, left, up_left)
    bottom_rightmost = M[-1][-1]
    return bottom_rightmost # edit distance

"""
  _ A V T
_ 0 1 2 3
A 1 0 1 2
P 2 1 1 2
T 3 2 2 1
"""
assert edit_distance("AVT", "APT") == 1

# N-gram overlap:
def n_gram_overlap(query, index, n):
"""
Returns the number of n-grams that appear
in both the query and the index.
"""
    query_n_grams = set(n_grams(query, n))
    index_n_grams = set(n_grams(index, n))
    return len(query_n_grams & index_n_grams)

"""
index:
an: anna, banana, bane
ba: banana, bane, cuba
na: anna, banana, native
"""
assert n_gram_overlap("anna", index, 2) == 2
assert n_gram_overlap("banana", index, 2) == 3
assert n_gram_overlap("bane", index, 2) == 2
assert n_gram_overlap("cuba", index, 2) == 1

# Jaccard coefficient:
def jaccard(A, B, n):
"""
Accounts for the length of the queries A and B
by normalizing.
"""
    A_n_grams = set(n_grams(A, n))
    B_n_grams = set(n_grams(B, n))
    intersection = A_n_grams & B_n_grams
    union = A_n_grams | B_n_grams
    return len(intersection) / len(union)

assert jaccard("bana", "anna", 2) == 2 / 3
assert jaccard("bana", "bane", 2) == 2 / 4
assert jaccard("bana", "banana", 2) == 3 / 3

# Soundex:
def soundex(term):
"""
Reduce the number of terms that sound the
same to a 4 character code.
"""
# 1. keep the first letter
# 2. replace vowels with 0s
# 3. replace consonants with their number:
#    b, f, p, v -> 1
#    c, g, j, k, q, s, x, z -> 2
#    d, t -> 3
#    l -> 4
#    m, n -> 5
#    r -> 6
# 4. de-dupe consecutive repeated numbers
# 5. remove 0s
# 6. pad with 0s or truncate to 4 characters
    pass

"""
Onomatopoeia -> On0m0t0p0000 -> O50503010000
-> O50503010 -> O5531 -> O553
Spencer -> Sp0nc0r -> S105206 -> S105206
-> S1526 -> S152
"""
assert soundex("onomatopoeia") == "O553"
assert soundex("SPENCER") == "S152"

# Block sort-based indexing:
def bsbi(collection, block_size):
    # create all the blocks
    while not all documents are processed:
        if block is full:
            sort block
            create postings lists in block
            write block to disk
            clear block
            continue
        read next document
        for term in document:
```

```
        get or create term_id for term
            in dictionary
        add (term_id, doc_id) to block
    # create the last block
    if block is not empty:
        sort block
        create postings lists in block
        write block to disk
        clear block
    # merge the blocks
    for every pair of blocks a, b:
        merge(a, b)
    # undo the term -> term_id mapping
    for term_id in dictionary:
        replace term_id with term
    return (dictionary, postings_lists)

"""
document 1: "b", "a"
document 2: "a", "c", "b"
mapping: "b" -> 1, "a" -> 2, "c" -> 3
block 1: [(1, 1), (2, 1), (2, 2)]
block 2: [(3, 2), (1, 2)]
block 1: {1: [1], 2: [1, 2]}
block 2: {3: [2], 1: [2]}
merged: {1: [1, 2], 2: [1, 2], 3: [2]}
"""

# Single-pass in-memory indexing:
def spimi(collection):
    while not all documents are processed:
        read next document
        for term in document:
            add (term, doc_id) to postings list
            if memory limit is reached:
                sort keys of postings list
                write postings list to disk
                clear postings list
    for every pair of blocks a, b:
        merge(a, b)

"""
document 1: "b", "a"
document 2: "a", "c", "b"
memory limit: 3 pairs
block 1: {"b": [1], "a": [1, 2]}
block 2: {"c": [2], "b": [2]}
merged: {"b": [1, 2], "a": [1, 2], "c": [2]}
"""

# MapReduce
def map_reduce(collection):
    # list(k, v)
    mapped = [(term, 1) for term in document
        for document in collection]
    # k, list(v) -> output
    reduced = [(term, freq(term)) for term
        in mapped]
    return reduced

"""
document 1: "a", "b", "a"
document 2: "a", "c", "b"
mapped: [("a", 1), ("b", 1), ("a", 1),
    ("a", 1), ("c", 1), ("b", 1)]
intermediate: [("a", [1, 1, 1]), ("b", [1, 1]),
    ("c", [1])]
reduced: [("a", 3), ("b", 2), ("c", 1)]
"""

# Linear merging:
def linear_merge(A, B, C..., Z):
"""
1. merge B into A
2. merge C into A
...
n. merge Z into A
"""
    pass

# Logarithmic merging:
def logarithmic_merge(blocks):
    while any pair of blocks i, j have the
        same size:
        merge(i, j)
"""
X(n), where n denotes the size of the block X:
A(1) + B(1) -> A(2)
A(2) + C(1) -> A(2), C(1)
A(2), C(1) + D(1) -> A(2) + C(2) -> A(4)
...
"""
```

```
# Index compression: dictionary as a string:
def dict_as_string(dict):
"""
Stores fixed-size pointers to terms in the
dictionary instead of storing the variable-
length terms themselves.
"""
   s = ""
   for row in dict:
      s += row.term
      row.term_ptr = &s[start of term]
   return s

"""
dict(term, ...):
operand, ...
oeprate, ...
operation, ...
operator, ...
"""
s = dict_as_string(dict)
assert s == "operandoeprateoperationoperator"
assert dict[0].term_ptr == &s[0]
assert dict[1].term_ptr == &s[7]
assert dict[2].term_ptr == &s[15]
assert dict[3].term_ptr == &s[24]

# Index compression: blocking:
def blocking(dict, k):
"""
Stores only every k-th pointer to a term,
using length of the term as a proxy for
removed pointers.
"""
   s = ""
   for row, index in enumerate(dict):
      s += str(len(row.term)) + row.term
      if index is a multiple of k:
         row.term_ptr = &s[start of block]
   return s

assert blocking(dict, 2) ==
   "7operand7operate9operation8operator"
assert dict[0].term_ptr == &s[0]
assert dict[1].term_ptr is None
assert dict[2].term_ptr == &s[17]
assert dict[3].term_ptr is None

# Index compression: front coding:
def front_coding(dict, k):
"""
Eliminates common prefixes between terms.
"""
   s = ""
   blocks = chunks(dict, k)
   for block in blocks:
      prefix = (longest common prefix
         for all terms in block)
      s += str(len(prefix))
      s += '*'
      # add the first suffix
      suffix = block[0][len(prefix):]
      s += suffix
      # add the remaining suffixes
      for term remaining in block:
         suffix = term[len(prefix):]
         s += str(len(suffix))
         s += <> # diamond symbol
         s += suffix
      # add the pointer
      block[0].term_ptr = &s[start of block]
   return s

assert front_coding(dict, 2) ==
   "7opera*nd2<>te9operat*ion2<>or"
assert dict[0].term_ptr == &s[0]
assert dict[1].term_ptr is None
assert dict[2].term_ptr == &s[15]
assert dict[3].term_ptr is None

# Posting list compression: gap encoding
def gap_encoding(postings_list):
"""
Stores the difference between the current
and the previous document ID instead of
storing the document ID itself.
"""
   ids = []
   prev = 0
   for doc_id in postings_list:
      ids.append(doc_id - prev)
      prev = doc_id
```

```
    return ids

assert gap_encoding(
    [10, 25, 152, 153, 281, 16666])
    == [10, 15, 127, 1, 128, 16385]

# Posting list compression: variable byte
# encoding:
def variable_byte_encoding(number):
"""
Stores the number using a variable number
of bytes instead of a fixed length.
The most significant bit of each byte is
a continuation bit that indicates whether
the next byte is part of the number.
"""
    bin = binary repr. of number
    if len(bin) is not a multiple of 7:
        left-pad (prepend) bin with 0s
    for each 7-bit chunk in bin:
        if chunk is the last (rightmost) chunk:
            prepend 1 to chunk
        else:
            prepend 0 to chunk
    return concatenation of all chunks

assert variable_byte_encoding(1) == 0b10000001
assert variable_byte_encoding(127) == 0b11111111
assert variable_byte_encoding(128) ==
    0b 00000001 10000000
```