Information Retrieval and Question Answering

Jonathan May

November 4, 2022(Prepared for Fall 2022)

1 Information Retrieval

You can take entire courses on information retrieval (e.g. CSCI 572, Stanford's CS 276,), and it's not always considered the same thing as NLP, but it's the general problem which question answering (QA) is an instance of, and QA is squarely an NLP task.

IR is probably the NLP application you use most often and the most commercially successful NLP application – search (like web search) is an IR problem. The general idea is that you have a (very large) collection of **documents** that for our purposes you can think of as text strings, usually paragraph or more long. You then have a **query** which is a short phrase, natural language of some sort, that indicates the kind of document you're looking for. In general, the task of IR is to **rank** all the documents in order based on how much they satisfy the goals of the query. In practice, all but a small number of documents are ranked at ∞ and the rest are chosen, so you can think of that as a way of **classifying** each document as matching or not.

1.1 IR evaluation methods

Generally we have a set of held-out queries and documents along with annotations of documents that are relevant for each query. The document set is typically very large and the relevant document set for a query much smaller than the background data. It also is generally not perfect recall, since this would require checking every document in a collection (background assumed not to match).

1.1.1 Precision and Recall @K

Standard precision and recall seem valid but as there are multiple relevant documents per query, we must consider how many documents we are allowed to return. Precision@K = you get K docs per query. So if you have N queries and H total hits, then P@K = $\frac{H}{NK}$. Similarly R@K, and K here still refers to the number of docs per query. If more than K relevant exist you can't get 100%.

None of these take the position in the K you return into account.

1.1.2 Mean Average Precision (MAP)

Consider ranked list of documents per query. Take average of precision over several K. Example for K of 5, with 3 hits:

- $1/1/1/0/0 = \frac{1}{5}(1 + 1 + 1 + \frac{3}{4} + \frac{3}{5}) = .87$
- $0/0/1/1/1 = \frac{1}{5}(0 + 0 + \frac{1}{3} + \frac{1}{2} + \frac{3}{5}) = .2867$

That's the Average Precision. Take a mean across all queries and you get MAP = Mean Average Precision.

Other metrics that you can look up:

- nDCG (Normalized Discounted Cumulative Gain) – take how relevant a document is into account and penalize for being lower on the list
- MRR (Mean Reciprocal Rank) – For a query, if the first relevant hit is at position r in your list, score $\frac{1}{r}$. Average across all queries.
- User results (from clicks, eye-tracking, rarely from reported relevance results).

1.2 Vocabulary Models

I'm sure you're thinking 'ok, this is going to be old stuff and then we'll see the neural stuff that works much better.' In a way this is true, but if you're searching for something very specific, just making sure the words you search for are matched is better than the more 'semantic' neural methods and so (we think) some form of these exact match approaches are still employed in commercial search.

1.2.1 Not Covered

- SQL
- other logical forms (how to exclude something)

1.2.2 Inverted Index Lookup

The straightforward approach is 'if the words of the query are in the document, return the document.' This is often what I want from, say email. Like, I know there was an email about when this class was held in 2020, So I would search `csci 662 fall 2020 schedule` and hopefully I'd find the email that has those words in it.

Complications:

- What if the email says `cs662` (tokenization)?
- what if the email says `autumn` (synonym)?
- what if the email says `schedules` (normalization)?

- What if I have 25 million emails to look up? (scalability)?

To solve the first three you do some standard document preprocessing. To do the last you use an inverted index. For each term in the vocabulary you save off a list of documents in sorted order that have that word. Then you can easily do an intersection.

(example from Chris Manning's slides)

$t1 : 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64 \rightarrow 128$

$t2 : 1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 13 \rightarrow 21 \rightarrow 34$

Algorithm 1 Intersect

Require: sorted queues/min heaps v_1, v_2

Ensure: a contains the elements common to both v_1 and v_2

```
a ← {}
while v1 ≠ () and v2 ≠ () do
    if v1.head = v2.head then
        a ← a ∪ {v1.head}
        v1.pop()
        v2.pop()
    else if v1.head < v2.head then
        v1.pop()
    else
        v2.pop()
    end if
end while
```

1.2.3 tf-idf

If all your query words match the document then you have a match, but if only some of them match, then what? Should it just be most of them that match? No, all words are not equally relevant. Intuitively, content words matter more. So how do we quantify this?

Term Frequency is an important characteristic. If a word you search for occurs 10 times in document 1 and 1 time in document 2 it's probably more important in document 1. But is it 10 times more important? Probably not, so the rule of thumb is, for term (word) t occurring f times in document d, the term frequency rate $tf_{t,d}$ is:

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

Why add 1? Because the log of 1 is 0, which would be awkward. Why set it to 0 if t doesn't exist? Because the log of 0 is undefined, which is even more awkward. Practicalities!

Term Scarcity is also important! What a contradiction! But it's true – if your query is 'the yankees', it's much more important to match 'yankees' than 'the.' Let N be the number of documents in your collection. Let d be the number of documents term t appears in. Then the document frequency rate idf_t is:

$$df_t = \log_{10}(\frac{N}{d})$$

This only really matters for queries with more than one term, of course. But we weight a term t by $tf.idf_{t,d} = tf_{t,d} \times df_t$ and can represent the score for a query-document pair as the sum of the $tf.idf_{t,d}$ for all t in both query and document.

1.2.4 BM25

(I was going to write something here about BM25 which is a fancier version of tf-idf that doesn't punish documents for being short but it's not critical knowledge.)

1.3 Dense Vector Models

Of course, these methods only work if you have an exact match of some terms. We tried to avoid mismatches with tokenization, normalization, stemming, but we still might not guess the exact words. So, you guessed it, we can try using sequence representations, where it's assumed that two sequences that are similar will have a small cosine distance (note we are relying on the semantics to be pretty much only synonymy.

1.3.1 Transformer Approach

Since we do have lots (hundreds of thousands of examples) of training data, we could just use BERT, right? We would encode the query, a `[SEP]` token, then the document text, and learn a binary classifier to predict whether the document matches the query.

Would this work? Yes and no. That approach will give good performance on lots of IR corpora, but in order to use it you have to pair each query with each document. At web scale that's impossible.

1.3.2 Bi-encoder Approach

A lighterweight approach uses two (hence bi-) encoders, one to encode all the documents and one to encode the query. At training time you optimize to maximize dot product/minimize cosine distance of the query and its matching documents; you can have two separate encoders (perhaps BERT-based), one that gets optimized jointly, and lots of other variants are possible too. At inference time you take your entire corpus and encode it, then for each query you get the nearest neighbors; there are some really fast approaches for this; I like the Faiss library.

2 Question Answering

Question Answering (QA) is a special kind of IR where the query is posed in the form of a question, and rather than a document, we typically want a fact (that often comes from a document).

2.1 Open Domain QA

If there's a big document collection you can treat the problem like IR, at least at first. Simply(?) rephrase the question as a query (e.g. 'What is the capital of New York' → 'New York capital'), then retrieve matching documents, then proceed as if you've been given the passage, as described below.

Or we can make things a little easier and just provide the document. That is often what is done; see list of data sets below.

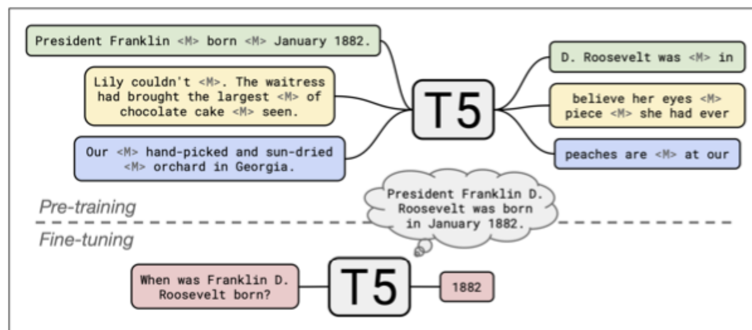
2.2 Extractive Factoid QA

Now that you have a document, we need to whittle it down to find which sentence contains the answer. Here a single BERT model might actually work – encode the question and each sentence, and binary classify. The scale is not so big.

Finally we need to find the actual passage. An IE or other span-based model simply needs to find the beginning and the end from the selected sentence. Assuming nothing has gone wrong, you now have your answer!

2.3 Abstractive QA

What if the exact answer isn't in the text but a reasonable person could figure it out? What if we just want to train a big model with a lot of text and ask it questions? GPT-3 can do that, as can T5, which is an encoder-decoder model trained using the paradigm in the figure below.



2.4 Classic Approach

QA is really old and the classic approach is a pipeline of multiple systems. Even now those pipelines can often outperform slick neural models. Here's a brief list of the steps:

- Figure out the answer type (Is 'Which US state capital has the largest population?' asking for a number, a state, or a city?)
- Figure out the focus (what words in the question should be replaced by the answer?)
- Based on the answer type, extract entities of the correct type from the retrieved document

- Rerank all the candidate answers

Something like this was used by Watson to beat experts like Ken Jennings in 2011. There is current work on doing well in "quiz bowl" environments where context information and the question itself is progressively revealed and answering early yields more points.

2.5 Datasets

- SQuAD (Stanford Question Answering Dataset, 2016) – Annotators took passages from wikipedia, then made up questions whose answers are in the passages. SQuAD 2.0 (2018) also includes unanswerable questions – 150k total questions. If you include the passages it's extractive QA, if you throw them away it's open-domain with wikipedia as the document corpus.
- HotpotQA (2018) – take several documents and make a question that you need to read and reason from all of them to make a decision. Requires multi-hop reasoning.
- TriviaQA, Natural Questions (2017, 2019) were written by trivia people/google users, and also include contexts with the answer, but weren't written by unconsciously biased annotators.
- TyDi QA (typological diverse) (2020) is 204k QA pairs from 11 different languages.