CHAPTER

11 Information Retrieval and Retrieval-Augmented Generation

On two occasions I have been asked,—"Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" ... I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.
Babbage (1864)

People need to know things. So pretty much as soon as there were computers we were asking them questions. By 1961 there was a system to answer questions about American baseball statistics like "How many games did the Yankees play in July?" (Green et al., 1961). Even fictional computers in the 1970s like Deep Thought, invented by Douglas Adams in *The Hitchhiker's Guide to the Galaxy*, answered "the Ultimate Question Of Life, The Universe, and Everything".[1] And because so much knowledge is encoded in text, systems were answering questions at human-level performance even before LLMs: IBM's Watson system won the TV game-show *Jeopardy!* in 2011, surpassing humans at answering questions like:

WILLIAM WILKINSON'S "AN ACCOUNT OF THE PRINCIPALITIES OF WALLACHIA AND MOLDOVIA" INSPIRED THIS AUTHOR'S MOST FAMOUS NOVEL[2]

It follows naturally, then, that an important function of large language models is to fill **human information needs** by answering people's questions. And since a lot of information is online, answering questions is closely related to web information retrieval, the task performed by search engines. Indeed, the distinction is becoming ever more fuzzy, as modern search engines are integrated with large language models.

factoid questions

Consider some simple information needs, for example **factoid questions** that can be answered with facts expressed in short texts like the following:

(11.1) Where is the Louvre Museum located?

(11.2) Where does the energy in a nuclear explosion come from?

(11.3) How to get a script l in latex?

To get an LLM to answer these questions, we can just prompt it! For example a pretrained LLM that has been instruction-tuned on question answering (Chapter 9) could directly answer the following question

Where is the Louvre Museum located?

by performing conditional generation given this prefix, and take the response as the answer. This works because large language models have processed a lot of facts in their pretraining data, including the location of the Louvre, and have encoded this information in their parameters. Factual knowledge of this type seems to be stored

[1] The answer was 42, but unfortunately the question was never revealed.

[2] The answer, of course, is 'Who is Bram Stoker', and the novel was *Dracula*.

in the connections in the very large feedforward layers of transformer models (Geva et al., 2021; Meng et al., 2022).

Simply prompting an LLM can be a useful approach to answer many factoid questions. But the fact that knowledge is stored in the feedforward weights of the LLM leads to a number of problems with prompting as a method for correctly answering factual questions.

hallucinate

The first and main problem is that LLMs often give the wrong answer to factual questions! Large language models **hallucinate**. A hallucination is a response that is not faithful to the facts of the world. That is, when asked questions, large language models sometimes make up answers that sound reasonable. For example, Dahl et al. (2024) found that when asked questions about the legal domain (like about particular legal cases), large language models hallucinated from 69% to 88% of the time! LLMs sometimes give incorrect factual responses even when the correct facts are stored in the parameters; this seems to be caused by the feedforward layers failing to recall the knowledge stored in their parameters (Jiang et al., 2024).

calibrated

And it's not always possible to tell when language models are hallucinating, partly because LLMs aren't well-**calibrated**. In a **calibrated** system, the confidence of a system in the correctness of its answer is highly correlated with the probability of an answer being correct. So if a calibrated system is wrong, at least it might hedge its answer or tell us to go check another source. But since language models are not well-calibrated, they often give a very wrong answer with complete certainty (Zhou et al., 2024).

A second problem with answering questions with simple prompting methods is that prompting a large language model to answer from its pretrained parameters doesn't allow us to ask questions about proprietary data. We would like to use language models to answer factual questions about proprietary data like personal email. Or for the healthcare application we might want to apply a language model to medical records. Or a company may have internal documents that contain answers for customer service or internal use. Or legal firms need to ask questions about legal discovery from proprietary documents. None of this data (hopefully) was in the large web-based corpora that large language models are pretrained on.

A final issue with using large language models to answer knowledge questions is that they are static; they were pretrained once, at a particular time. This means that LLMs cannot answer questions about rapidly changing information (like questions about something that happened last week) since they won't have up-to-date information from after their release data.

RAG
information retrieval

One solution to all these problems with simple prompting for answering factual questions is to give a language model external sources of knowledge, for example proprietary texts like medical or legal records, personal emails, or corporate documents, and to use those documents in answering questions. This method is called **retrieval-augmented generation** or **RAG**, and that is the method we will focus on in this chapter. In RAG we use **information retrieval** (**IR**) techniques to retrieve documents that are likely to have information that might help answer the question. Then we use a large language model to **generate** an answer given these documents.

Basing our answers on retrieved documents can solve some of the problems with using simple prompting to answer questions. First, it helps ensure that the answer is grounded in facts from some curated dataset. And the system can give the user the answer accompanied by the context of the passage or document the answer came from. This information can help users have confidence in the accuracy of the answer (or help them spot when it is wrong!). And these retrieval techniques can be used on any proprietary data we want, such as legal or medical data for those applications.

We'll begin by introducing information retrieval, the task of choosing the most relevant document from a document set given a user's query expressing their information need. We'll see the classic method based on cosines of sparse tf-idf vectors, modern neural 'dense' retrievers based on instead representing queries and documents neurally with BERT or other language models. We then introduce retriever-based question answering and the retrieval-augmented generation paradigm.

Finally, we'll discuss various datasets with questions and answers that can be used for finetuning LLMs in instruction tuning and for use as benchmarks for evaluation.

11.1 Information Retrieval

information retrieval
IR

Information retrieval or **IR** is the name of the field encompassing the retrieval of all manner of media based on user information needs. The resulting IR system is often called a **search engine**. Our goal in this section is to give a sufficient overview of IR to see its application to question answering. Readers with more interest specifically in information retrieval should see the Historical Notes section at the end of the chapter and textbooks like Manning et al. (2008).

ad hoc retrieval
document
collection
term
query

The IR task we consider is called **ad hoc retrieval**, in which a user poses a **query** to a retrieval system, which then returns an ordered set of **documents** from some **collection**. A **document** refers to whatever unit of text the system indexes and retrieves (web pages, scientific papers, news articles, or even shorter passages like paragraphs). A **collection** refers to a set of documents being used to satisfy user requests. A **term** refers to a word in a collection, but it may also include phrases. Finally, a **query** represents a user's information need expressed as a set of terms. The high-level architecture of an ad hoc retrieval engine is shown in Fig. 11.1.

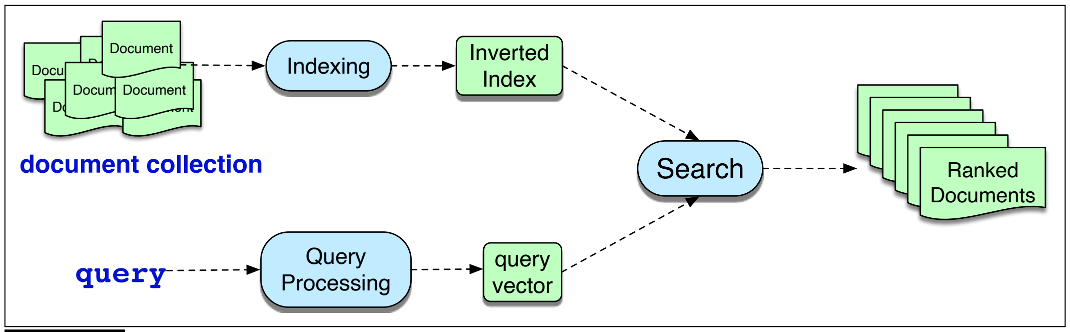


Figure 11.1 The architecture of an ad hoc IR system.

The basic IR architecture uses the vector space model we introduced in Chapter 5, in which we map queries and document to vectors based on unigram word counts, and use the cosine similarity between the vectors to rank potential documents (Salton, 1971). This is thus an example of the **bag-of-words** model introduced in Appendix K, since words are considered independently of their positions.

11.1.1 Term weighting and document scoring

Let's look at the details of how the match between a document and query is scored.

term weight

BM25

We don't use raw word counts in IR, instead computing a **term weight** for each document word. Two term weighting schemes are common: the **tf-idf** weighting introduced in Chapter 5, and a slightly more powerful variant called **BM25**.

We'll reintroduce tf-idf here so readers don't need to look back at Chapter 5. Tf-idf (the '-' here is a hyphen, not a minus sign) is the product of two terms, the term frequency **tf** and the inverse document frequency **idf**.

The term frequency tells us how frequent the word is; words that occur more often in a document are likely to be informative about the document's contents. We usually use the $\log_{10}$ of the word frequency, rather than the raw count. The intuition is that a word appearing 100 times in a document doesn't make that word 100 times more likely to be relevant to the meaning of the document. We also need to do something special with counts of 0, since we can't take the log of 0.[3]

$$\text{tf}_{t,d} = \begin{cases} 1 + \log_{10} \text{count}(t,d) & \text{if } \text{count}(t,d) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{11.4}$$

If we use log weighting, terms which occur 0 times in a document would have $\text{tf} = 0$, 1 times in a document $\text{tf} = 1 + \log_{10}(1) = 1 + 0 = 1$, 10 times in a document $\text{tf} = 1 + \log_{10}(10) = 2$, 100 times $\text{tf} = 1 + \log_{10}(100) = 3$, 1000 times $\text{tf} = 4$, and so on.

The **document frequency** df_t of a term t is the number of documents it occurs in. Terms that occur in only a few documents are useful for discriminating those documents from the rest of the collection; terms that occur across the entire collection aren't as helpful. The **inverse document frequency** or **idf** term weight (Sparck Jones, 1972) is defined as:

$$\text{idf}_t = \log_{10} \frac{N}{\text{df}_t} \tag{11.5}$$

where N is the total number of documents in the collection, and df_t is the number of documents in which term t occurs. The fewer documents in which a term occurs, the higher this weight; the lowest weight of 0 is assigned to terms that occur in every document.

Here are some idf values for some words in the corpus of Shakespeare plays, ranging from extremely informative words that occur in only one play like *Romeo*, to those that occur in a few like *salad* or *Falstaff*, to those that are very common like *fool* or so common as to be completely non-discriminative since they occur in all 37 plays like *good* or *sweet*.[4]

Word	**df**	**idf**
Romeo	1	1.57
salad	2	1.27
Falstaff	4	0.967
forest	12	0.489
battle	21	0.246
wit	34	0.037
fool	36	0.012
good	37	0
sweet	37	0

[3] We can also use this alternative formulation, which we have used in earlier editions: $\text{tf}_{t,d} = \log_{10}(\text{count}(t,d) + 1)$

[4] *Sweet* was one of Shakespeare's favorite adjectives, a fact probably related to the increased use of sugar in European recipes around the turn of the 16th century (Jurafsky, 2014, p. 175).

The **tf-idf** value for word t in document d is then the product of term frequency $\text{tf}_{t,d}$ and IDF:

$$\text{tf-idf}(t,d) = \text{tf}_{t,d} \cdot \text{idf}_t \tag{11.6}$$

11.1.2 Document Scoring

We score document d by the cosine of its vector $\mathbf{d}$ with the query vector $\mathbf{q}$:

$$\text{score}(q,d) = \cos(\mathbf{q},\mathbf{d}) = \frac{\mathbf{q}\cdot\mathbf{d}}{|\mathbf{q}||\mathbf{d}|} \tag{11.7}$$

Another way to think of the cosine computation is as the dot product of unit vectors; we first normalize both the query and document vector to unit vectors, by dividing by their lengths, and then take the dot product:

$$\text{score}(q,d) = \cos(\mathbf{q},\mathbf{d}) = \frac{\mathbf{q}}{|\mathbf{q}|}\cdot\frac{\mathbf{d}}{|\mathbf{d}|} \tag{11.8}$$

We can spell out Eq. 11.8, using the tf-idf values and spelling out the dot product as a sum of products:

$$\text{score}(q,d) = \sum_{t\in\mathbf{q}} \frac{\text{tf-idf}(t,q)}{\sqrt{\sum_{q_i\in q}\text{tf-idf}^2(q_i,q)}} \cdot \frac{\text{tf-idf}(t,d)}{\sqrt{\sum_{d_i\in d}\text{tf-idf}^2(d_i,d)}} \tag{11.9}$$

Now let's use Eq. 11.9 to walk through an example of a tiny query against a collection of 4 nano documents, computing tf-idf values and seeing the rank of the documents. We'll assume all words in the following query and documents are downcased and punctuation is removed:

Query:	sweet love
Doc 1:	Sweet sweet nurse! Love?
Doc 2:	Sweet sorrow
Doc 3:	How sweet is love?
Doc 4:	Nurse!

Fig. 11.2 shows the computation of the tf-idf cosine between the query and Document 1, and the query and Document 2. The cosine is the normalized dot product of tf-idf values, so for the normalization we must need to compute the document vector lengths $|q|$, $|d_1|$, and $|d_2|$ for the query and the first two documents using Eq. 11.4, Eq. 11.5, Eq. 11.6, and Eq. 11.9 (computations for Documents 3 and 4 are also needed but are left as an exercise for the reader). The dot product between the vectors is the sum over dimensions of the product, for each dimension, of the values of the two tf-idf vectors for that dimension. This product is only non-zero where both the query and document have non-zero values, so for this example, in which only *sweet* and *love* have non-zero values in the query, the dot product will be the sum of the products of those elements of each vector.

Document 1 has a higher cosine with the query (0.747) than Document 2 has with the query (0.0779), and so the tf-idf cosine model would rank Document 1 above Document 2. This ranking is intuitive given the vector space model, since Document 1 has both terms including two instances of *sweet*, while Document 2 is missing one of the terms. We leave the computation for Documents 3 and 4 as an exercise for the reader.

			Query			
word	**cnt**	**tf**	**df**	**idf**	**tf-idf**	**n'lized** = tf-idf/$\lvert q\rvert$
sweet	1	1	3	0.125	0.125	0.383
nurse	0	0	2	0.301	0	0
love	1	1	2	0.301	0.301	0.924
how	0	0	1	0.602	0	0
sorrow	0	0	1	0.602	0	0
is	0	0	1	0.602	0	0
$\lvert q\rvert = \sqrt{.125^2 + .301^2} = .326$						

		Document 1						Document 2		
word	**cnt**	**tf**	**tf-idf**	**n'lized**	$\times$ **q**	**cnt**	**tf**	**tf-idf**	**n'lized**	$\times$**q**
sweet	2	1.301	0.163	0.357	**0.137**	1	1.000	0.125	0.203	**0.0779**
nurse	1	1.000	0.301	0.661	0	0	0	0	0	0
love	1	1.000	0.301	0.661	**0.610**	0	0	0	0	**0**
how	0	0	0	0	0	0	0	0	0	0
sorrow	0	0	0	0	0	1	1.000	0.602	0.979	0
is	0	0	0	0	0	0	0	0	0	0
$\lvert d_1\rvert = \sqrt{.163^2 + .301^2 + .301^2} = .456$						$\lvert d_2\rvert = \sqrt{.125^2 + .602^2} = .615$				
		Cosine: $\sum$ of column:			**0.747**			Cosine: $\sum$ of column:		**0.0779**

Figure 11.2 Computation of tf-idf cosine score between the query and nano-documents 1 (0.747) and 2 (0.0779), using Eq. 11.4, Eq. 11.5, Eq. 11.6 and Eq. 11.9.

In practice, there are many variants and approximations to Eq. 11.9. For example, we might choose to simplify processing by removing some terms. To see this, let's start by expanding the formula for tf-idf in Eq. 11.9 to explicitly mention the tf and idf terms from Eq. 11.6:

$$\text{score}(q,d) \;=\; \sum_{t\in\mathbf{q}} \frac{\text{tf}_{t,q}\cdot\text{idf}_t}{\sqrt{\sum_{q_i\in q}\text{tf-idf}^2(q_i,q)}} \cdot \frac{\text{tf}_{t,d}\cdot\text{idf}_t}{\sqrt{\sum_{d_i\in d}\text{tf-idf}^2(d_i,d)}} \tag{11.10}$$

In one common variant of tf-idf cosine, for example, we drop the idf term for the document. Eliminating the second copy of the idf term (since the identical term is already computed for the query) turns out to sometimes result in better performance:

$$\text{score}(q,d) \;=\; \sum_{t\in\mathbf{q}} \frac{\text{tf}_{t,q}\cdot\text{idf}_t}{\sqrt{\sum_{q_i\in q}\text{tf-idf}^2(q_i,q)}} \cdot \frac{\text{tf}_{t,d}\cdot\text{idf}_t}{\sqrt{\sum_{d_i\in d}\text{tf-idf}^2(d_i,d)}} \tag{11.11}$$

Other variants of tf-idf eliminate various other terms.

BM25 A slightly more complex variant in the tf-idf family is the **BM25** weighting scheme (sometimes called Okapi BM25 after the Okapi IR system in which it was introduced (Robertson et al., 1995)). BM25 adds two parameters: k, a knob that adjust the balance between term frequency and IDF, and b, which controls the importance of document length normalization. The BM25 score of a document d given a query q is:

$$\sum_{t\in q} \overbrace{\log\left(\frac{N}{df_t}\right)}^{\text{IDF}} \overbrace{\frac{tf_{t,d}}{k\left(1-b+b\left(\frac{|d|}{|d_{\text{avg}}|}\right)\right)+tf_{t,d}}}^{\text{weighted tf}} \tag{11.12}$$

where $|d_{\text{avg}}|$ is the length of the average document. When k is 0, BM25 reverts to no use of term frequency, just a binary selection of terms in the query (plus idf). A large k results in raw term frequency (plus idf). b ranges from 1 (scaling by document length) to 0 (no length scaling). Manning et al. (2008) suggest reasonable values are k = [1.2,2] and b = 0.75. Kamphuis et al. (2020) is a useful summary of the many minor variants of BM25.

Stop words In the past it was common to remove high-frequency words from both the query and document before representing them. The list of such high-frequency

stop list

words to be removed is called a **stop list**. The intuition is that high-frequency terms (often function words like *the*, *a*, *to*) carry little semantic weight and may not help with retrieval, and can also help shrink the inverted index files we describe below. The downside of using a stop list is that it makes it difficult to search for phrases that contain words in the stop list. For example, common stop lists would reduce the phrase *to be or not to be* to the phrase *not*. In modern IR systems, the use of stop lists is much less common, partly due to improved efficiency and partly because much of their function is already handled by IDF weighting, which downweights function words that occur in every document. Nonetheless, stop word removal is occasionally useful in various NLP tasks so is worth keeping in mind.

11.1.3 Inverted Index

In order to compute scores, we need to efficiently find documents that contain words in the query. (Any document that contains none of the query terms will have a score of 0 and can be ignored.) The basic search problem in IR is thus to find all documents $d \in C$ that contain a term $q \in Q$.

inverted index

The data structure for this task is the **inverted index**, which we use for making this search efficient, and also conveniently storing useful information like the document frequency and the count of each term in each document.

An inverted index, given a query term, gives a list of documents that contain the

postings

term. It consists of two parts, a **dictionary** and the **postings**. The dictionary is a list of terms (designed to be efficiently accessed), each pointing to a **postings list** for the term. A postings list is the list of document IDs associated with each term, which can also contain information like the term frequency or even the exact positions of terms in the document. The dictionary can also store the document frequency for each term. For example, a simple inverted index for our 4 sample documents above, with each word containing its document frequency in {}, and a pointer to a postings list that contains document IDs and term counts in [], might look like the following:

how {1}	→	3 [1]
is {1}	→	3 [1]
love {2}	→	1 [1] → 3 [1]
nurse {2}	→	1 [1] → 4 [1]
sorry {1}	→	2 [1]
sweet {3}	→	1 [2] → 2 [1] → 3 [1]

Given a list of terms in query, we can very efficiently get lists of all candidate documents, together with the information necessary to compute the tf-idf scores we need.

There are alternatives to the inverted index. For the question-answering domain of finding Wikipedia pages to match a user query, Chen et al. (2017) show that indexing based on bigrams works better than unigrams, and use efficient hashing algorithms rather than the inverted index to make the search efficient.

11.1.4 Evaluation of Information-Retrieval Systems

We measure the performance of ranked retrieval systems using the same **precision** and **recall** metrics we have been using. We make the assumption that each document returned by the IR system is either **relevant** to our purposes or **not relevant**. Precision is the fraction of the returned documents that are relevant, and recall is the fraction of all relevant documents that are returned. More formally, let's assume a system returns T ranked documents in response to an information request, a subset R of these are relevant, a disjoint subset, N, are the remaining irrelevant documents, and U documents in the collection as a whole are relevant to this request. Precision and recall are then defined as:

$$Precision = \frac{|R|}{|T|} \qquad Recall = \frac{|R|}{|U|} \tag{11.13}$$

Unfortunately, these metrics don't adequately measure the performance of a system that *ranks* the documents it returns. If we are comparing the performance of two ranked retrieval systems, we need a metric that prefers the one that ranks the relevant documents higher. We need to adapt precision and recall to capture how well a system does at putting relevant documents higher in the ranking.

Rank	Judgment	$\textbf{Precision}_{Rank}$	$\textbf{Recall}_{Rank}$
1	R	1.0	.11
2	N	.50	.11
3	R	.66	.22
4	N	.50	.22
5	R	.60	.33
6	R	.66	.44
7	N	.57	.44
8	R	.63	.55
9	N	.55	.55
10	N	.50	.55
11	R	.55	.66
12	N	.50	.66
13	N	.46	.66
14	N	.43	.66
15	R	.47	.77
16	N	.44	.77
17	N	.44	.77
18	R	.44	.88
19	N	.42	.88
20	N	.40	.88
21	N	.38	.88
22	N	.36	.88
23	N	.35	.88
24	N	.33	.88
25	R	.36	1.0

Figure 11.3 Rank-specific precision and recall values calculated as we proceed down through a set of ranked documents (assuming the collection has 9 relevant documents).

Let's turn to an example. Assume the table in Fig. 11.3 gives rank-specific precision and recall values calculated as we proceed down through a set of ranked documents for a particular query; the precisions are the fraction of relevant documents seen at a given rank, and recalls the fraction of relevant documents found at the same

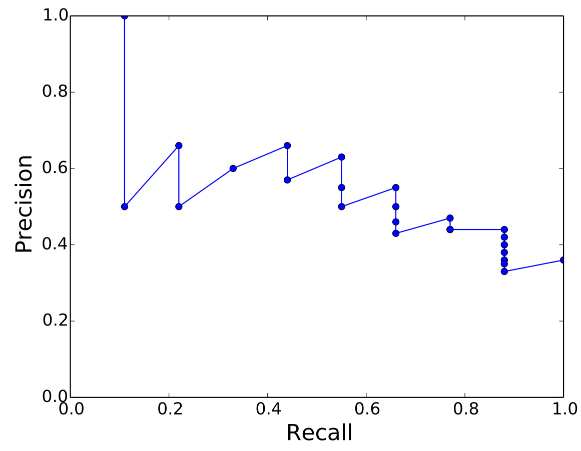


Figure 11.4 The precision recall curve for the data in table 11.3.

rank. The recall measures in this example are based on this query having 9 relevant documents in the collection as a whole.

Note that recall is non-decreasing; when a relevant document is encountered, recall increases, and when a non-relevant document is found it remains unchanged. Precision, on the other hand, jumps up and down, increasing when relevant documents are found, and decreasing otherwise. The most common way to visualize precision and recall is to plot precision against recall in a **precision-recall curve**, like the one shown in Fig. 11.4 for the data in table 11.3.

precision-recall curve

Fig. 11.4 shows the values for a single query. But we'll need to combine values for all the queries, and in a way that lets us compare one system to another. One way of doing this is to plot averaged precision values at 11 fixed levels of recall (0 to 100, in steps of 10). Since we're not likely to have datapoints at these exact levels, we use **interpolated precision** values for the 11 recall values from the data points we do have. We can accomplish this by choosing the maximum precision value achieved at any level of recall at or above the one we're calculating. In other words,

interpolated precision

$$\text{IntPrecision}(r) = \max_{i>=r} \text{Precision}(i) \tag{11.14}$$

This interpolation scheme not only lets us average performance over a set of queries, but also helps smooth over the irregular precision values in the original data. It is designed to give systems the benefit of the doubt by assigning the maximum precision value achieved at higher levels of recall from the one being measured. Fig. 11.5 and Fig. 11.6 show the resulting interpolated data points from our example.

Given curves such as that in Fig. 11.6 we can compare two systems or approaches by comparing their curves. Clearly, curves that are higher in precision across all recall values are preferred. However, these curves can also provide insight into the overall behavior of a system. Systems that are higher in precision toward the left may favor precision over recall, while systems that are more geared towards recall will be higher at higher levels of recall (to the right).

mean average precision

A second way to evaluate ranked retrieval is **mean average precision** (MAP), which provides a single metric that can be used to compare competing systems or approaches. In this approach, we again descend through the ranked list of items, but now we note the precision **only** at those points where a relevant item has been encountered (for example at ranks 1, 3, 5, 6 but not 2 or 4 in Fig. 11.3). For a single

Interpolated Precision	Recall
1.0	0.0
1.0	.10
.66	.20
.66	.30
.66	.40
.63	.50
.55	.60
.47	.70
.44	.80
.36	.90
.36	1.0

Figure 11.5 Interpolated data points from Fig. 11.3.

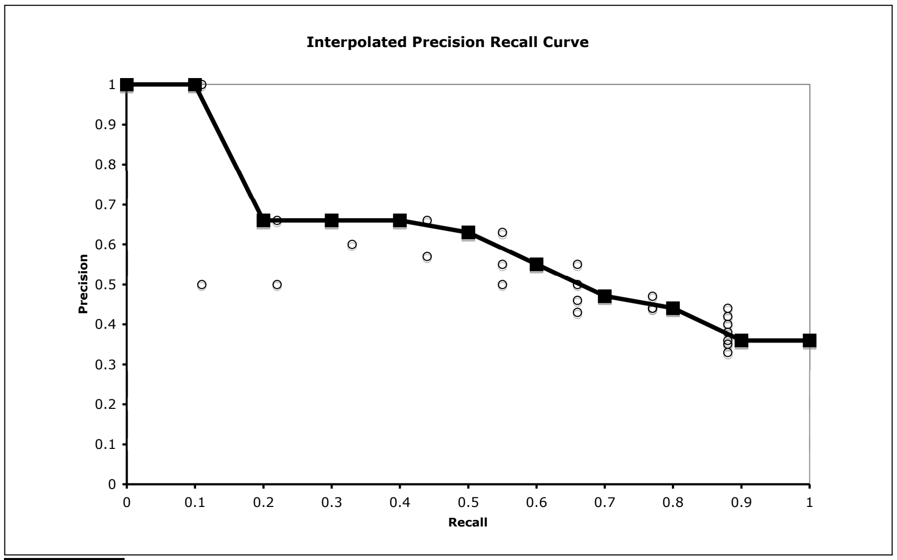


Figure 11.6 An 11 point interpolated precision-recall curve. Precision at each of the 11 standard recall levels is interpolated for each query from the maximum at any higher level of recall. The original measured precision recall points are also shown.

query, we average these individual precision measurements over the return set (up to some fixed cutoff). More formally, if we assume that R_r is the set of relevant documents at or above r, then the **average precision** (**AP**) for a single query is

$$\text{AP} = \frac{1}{|R_r|} \sum_{d \in R_r} \text{Precision}_r(d) \tag{11.15}$$

where $Precision_r(d)$ is the precision measured at the rank at which document d was found. For an ensemble of queries Q, we then average over these averages, to get our final MAP measure:

$$\text{MAP} = \frac{1}{|Q|} \sum_{q \in Q} \text{AP}(q) \tag{11.16}$$

The MAP for the single query (hence = AP) in Fig. 11.3 is 0.6.

11.2 Information Retrieval with Dense Vectors

The classic tf-idf or BM25 algorithms for IR have long been known to have a conceptual flaw: they work only if there is exact overlap of words between the query and document. In other words, the user posing a query (or asking a question) needs to guess exactly what words the writer of the answer might have used, an issue called the **vocabulary mismatch problem** (Furnas et al., 1987).

The solution to this problem is to use an approach that can handle synonymy: instead of (sparse) word-count vectors, using (dense) embeddings. This idea was first proposed for retrieval in the last century under the name of Latent Semantic Indexing approach (Deerwester et al., 1990), but is implemented in modern times via encoders like BERT.

The most powerful approach is to present both the query and the document to a single encoder, allowing the transformer self-attention to see all the tokens of both the query and the document, and thus building a representation that is sensitive to the meanings of both query and document. Then a linear layer can be put on top of the [CLS] token to predict a similarity score for the query/document tuple:

$$\begin{aligned} \mathbf{z} &= \text{BERT}(\text{q};\texttt{[SEP]};\text{d})\texttt{[CLS]} \\ \text{score}(q,d) &= \text{softmax}(\mathbf{U}(\mathbf{z})) \end{aligned} \tag{11.17}$$

This architecture is shown in Fig. 11.7a. Usually the retrieval step is not done on an entire document. Instead documents are broken up into smaller passages, such as non-overlapping fixed-length chunks of say 100 tokens, and the retriever encodes and retrieves these passages rather than entire documents. The query and document have to be made to fit in the BERT 512-token window, for example by truncating the query to 64 tokens and truncating the document if necessary so that it, the query, [CLS], and [SEP] fit in 512 tokens. The BERT system together with the linear layer $\mathbf{U}$ can then be fine-tuned for the relevance task by gathering a tuning dataset of relevant and non-relevant passages.

The problem with the full BERT architecture in Fig. 11.7a is the expense in computation and time. With this architecture, every time we get a query, we have to pass every single document in our entire collection through a BERT encoder jointly with the new query! This enormous use of resources is impractical for real cases.

At the other end of the computational spectrum is a much more efficient architecture, the **bi-encoder**. In this architecture we can encode the documents in the collection only one time by using two separate encoder models, one to encode the query and one to encode the document. We encode each document, and store all the encoded document vectors in advance. When a query comes in, we encode just this query and then use the dot product between the query vector and the precomputed document vectors as the score for each candidate document (Fig. 11.7b). For example, if we used BERT, we would have two encoders BERT_Q and BERT_D and we could represent the query and document as the `[CLS]` token of the respective encoders (Karpukhin et al., 2020):

$$\begin{aligned} \mathbf{z}_q &= \text{BERT}_Q(\text{q})\texttt{[CLS]} \\ \mathbf{z}_d &= \text{BERT}_D(\text{d})\texttt{[CLS]} \\ \text{score}(q,d) &= \mathbf{z}_q \cdot \mathbf{z}_d \end{aligned} \tag{11.18}$$

The bi-encoder is much cheaper than a full query/document encoder, but is also less accurate, since its relevance decision can't take full advantage of all the possi-

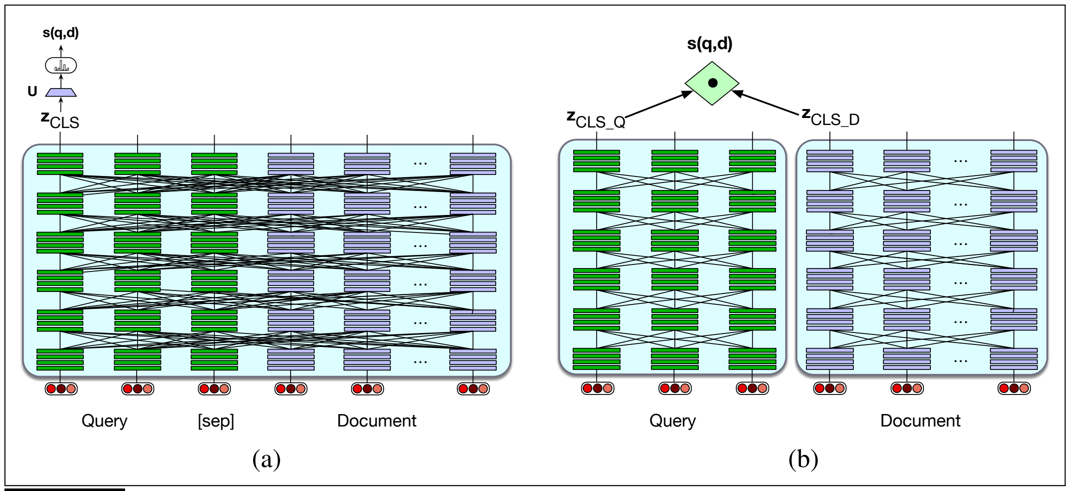


Figure 11.7 Two ways to do dense retrieval, illustrated by using lines between layers to schematically represent self-attention: (a) Use a single encoder to jointly encode query and document and finetune to produce a relevance score with a linear layer over the CLS token. This is too compute-expensive to use except in rescoring (b) Use separate encoders for query and document, and use the dot product between CLS token outputs for the query and document as the score. This is less compute-expensive, but not as accurate.

ble meaning interactions between all the tokens in the query and the tokens in the document.

There are numerous approaches that lie in between the full encoder and the bi-encoder. One intermediate alternative is to use cheaper methods (like BM25) as the first pass relevance ranking for each document, take the top N ranked documents, and use expensive methods like the full BERT scoring to rerank only the top N documents rather than the whole set.

ColBERT Another intermediate approach is the **ColBERT** approach of Khattab and Zaharia (2020) and Khattab et al. (2021), shown in Fig. 11.8. This method separately encodes the query and document, but rather than encoding the entire query or document into one vector, it separately encodes each of them into contextual representations for each token. These BERT representations of each document word can be pre-stored for efficiency. The relevance score between a query q and a document d is a sum of maximum similarity (MaxSim) operators between tokens in q and tokens in d. Essentially, for each token in q, ColBERT finds the most contextually similar token in d, and then sums up these similarities. A relevant document will have tokens that are contextually very similar to the query.

More formally, a question q is tokenized as $[q_1,\dots,q_n]$, prepended with a `[CLS]` and a special `[Q]` token, truncated to N=32 tokens (or padded with `[MASK]` tokens if it is shorter), and passed through BERT to get output vectors $\mathbf{q} = [\mathbf{q}_1,\dots,\mathbf{q}_N]$. The passage d with tokens $[d_1,\dots,d_m]$, is processed similarly, including a `[CLS]` and special `[D]` token. A linear layer is applied on top of $\mathbf{d}$ and $\mathbf{q}$ to control the output dimension, so as to keep the vectors small for storage efficiency, and vectors are rescaled to unit length, producing the final vector sequences $\mathbf{E}_q$ (length N) and $\mathbf{E}_d$ (length m). The ColBERT scoring mechanism is:

$$\text{score}(q,d) = \sum_{i=1}^{N} \max_{j=1}^{m} \mathbf{E}_{q_i} \cdot \mathbf{E}_{d_j} \tag{11.19}$$

While the interaction mechanism has no tunable parameters, the ColBERT ar-

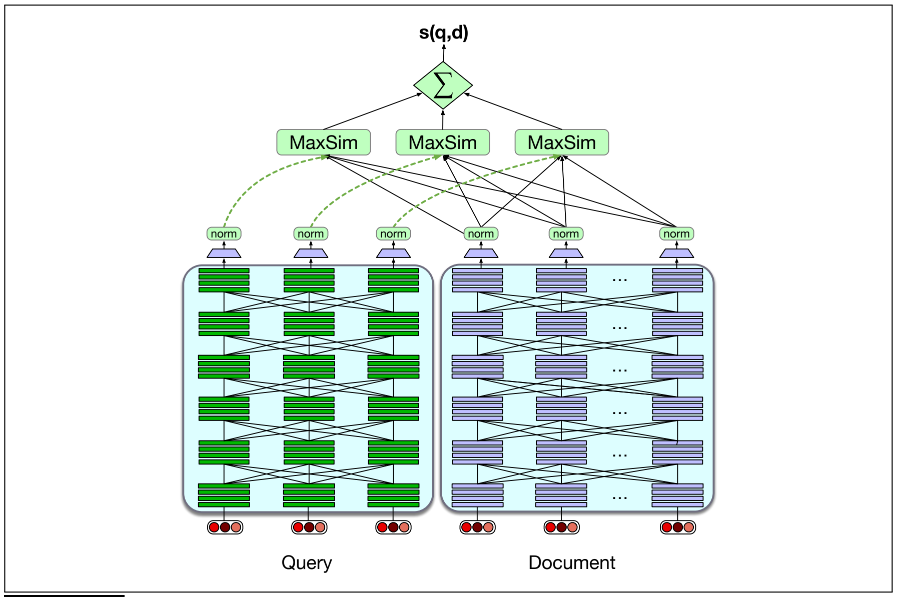


Figure 11.8 A sketch of the ColBERT algorithm at inference time. The query and document are first passed through separate BERT encoders. Similarity between query and document is computed by summing a soft alignment between the contextual representations of tokens in the query and the document. Training is end-to-end. (Various details aren't depicted; for example the query is prepended by a `[CLS]` and `[Q:]` tokens, and the document by `[CLS]` and `[D:]` tokens). Figure adapted from Khattab and Zaharia (2020).

chitecture still needs to be trained end-to-end to fine-tune the BERT encoders and train the linear layers (and the special `[Q]` and `[D]` embeddings) from scratch. It is trained on triples $\langle q, d^+, d^- \rangle$ of query q, positive document d^+ and negative document d^- to produce a score for each document using Eq. 11.19, optimizing model parameters using a cross-entropy loss.

All the supervised algorithms (like ColBERT or the full-interaction version of the BERT algorithm applied for reranking) need training data in the form of queries together with relevant and irrelevant passages or documents (positive and negative examples). There are various semi-supervised ways to get labels; some datasets (like MS MARCO Ranking, Section 11.4) contain gold positive examples. Negative examples can be sampled randomly from the top-1000 results from some existing IR system. If datasets don't have labeled positive examples, iterative methods like **relevance-guided supervision** can be used (Khattab et al., 2021) which rely on the fact that many datasets contain short answer strings. In this method, an existing IR system is used to harvest examples that do contain short answer strings (the top few are taken as positives) or don't contain short answer strings (the top few are taken as negatives), these are used to train a new retriever, and then the process is iterated.

Efficiency is an important issue, since every possible document must be ranked for its similarity to the query. For sparse word-count vectors, the inverted index allows this very efficiently. For dense vector algorithms finding the set of dense document vectors that have the highest dot product with a dense query vector is an instance of the problem of **nearest neighbor search**. Modern systems therefore make use of approximate nearest neighbor vector search algorithms like **Faiss** (Johnson et al., 2017).

Faiss

11.3 Answering Questions with RAG

Here we introduce an important paradigm for using LLMs to answer knowledge-based questions, based on first finding supportive text segments from the web or another other large collection of documents, and then generating an answer based on the documents. The method of generating based on retrieved documents is called **retrieval-augmented generation** or **RAG**, and the two components are sometimes called, for historical reasons, the retriever and the reader (Chen et al., 2017). Fig. 11.9 sketches out this standard model for answering questions.

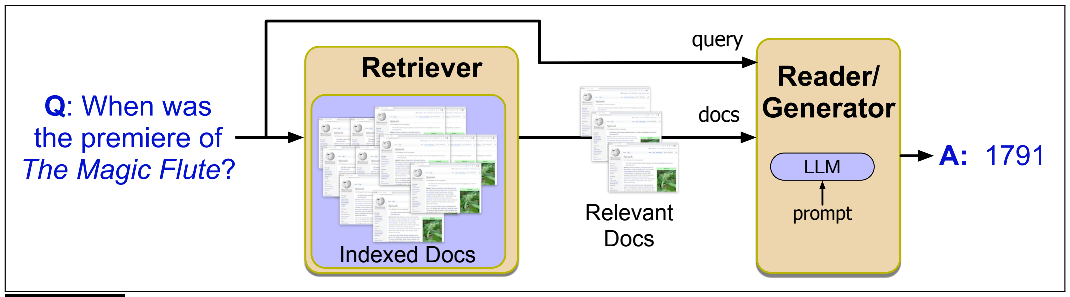


Figure 11.9 Retrieval-based question answering has two stages: **retrieval**, which returns relevant documents from the collection, and **reading**, in which an LLM **generates** answers given the documents as a prompt.

In the first stage of the 2-stage retrieve and read model in Fig. 11.9 we **retrieve** relevant passages from a text collection, for example using the dense retrievers of the previous section. In the second **reader** stage, we generate the answer via **retrieval-augmented generation**. In this method, we take a large pretrained language model, give it the set of retrieved passages and other text as its prompt, and autoregressively generate a new answer token by token.

11.3.1 Retrieval-Augmented Generation

retrieval-augmented generation
RAG

The standard reader algorithm is to generate from a large language model, conditioned on the retrieved passages. This method is known as **retrieval-augmented generation**, or **RAG**.

Recall that in simple conditional generation, we can cast the task of question answering as word prediction by giving a language model a question and a token like `A:` suggesting that an answer should come next:

```
Q: Who wrote the book ‘‘The Origin of Species"?  A:
```

Then we generate autoregressively conditioned on this text.

More formally, recall that simple autoregressive language modeling computes the probability of a string from the previous tokens:

$$p(x_1,\ldots,x_n) = \prod_{i=1}^{n} p(x_i|x_{<i})$$

And simple conditional generation for question answering adds a prompt like `Q:` , followed by a query q , and `A:`, all concatenated:

$$p(x_1,\ldots,x_n) = \prod_{i=1}^{n} p(\texttt{[Q:]} \,;\, q \,;\, \texttt{[A:]} \,;\, x_{<i})$$

The advantage of using a large language model is the enormous amount of knowledge encoded in its parameters from the text it was pretrained on. But as we mentioned at the start of the chapter, while this kind of simple prompted generation can work fine for many simple factoid questions, it is not a general solution for QA, because it leads to hallucination, is unable to show users textual evidence to support the answer, and is unable to answer questions from proprietary data.

The idea of retrieval-augmented generation is to address these problems by conditioning on the retrieved passages as part of the prefix, perhaps with some prompt text like "Based on these texts, answer this question:". Let's suppose we have a query q, and call the set of retrieved passages based on it $\text{R}(q)$. For example, we could have a prompt like:

Schematic of a RAG Prompt

```
retrieved passage 1

retrieved passage 2

...

retrieved passage n

Based on these texts, answer this question:  Q: Who wrote
the book ‘‘The Origin of Species"?  A:
```

Or more formally,

$$p(x_1,\ldots,x_n) \;=\; \prod_{i=1}^{n} p(x_i|\text{R}(q)\,;\,\text{prompt}\,;\,\texttt{[Q:]}\,;\,q\,;\texttt{[A:]}\,;x_{<i})$$

As with the span-based extraction reader, successfully applying the retrieval-augmented generation algorithm for QA requires a successful retriever, and often a two-stage retrieval algorithm is used in which the retrieval is reranked. Some complex questions may require **multi-hop** architectures, in which a query is used to retrieve documents, which are then appended to the original query for a second stage of retrieval. Details of prompt engineering also have to be worked out, like deciding whether to demarcate passages, for example with [SEP] tokens, and so on. Combinations of private data and public data involving an externally hosted large language model may lead to privacy concerns that need to be worked out (Arora et al., 2023). Much research in this area also focuses on ways to more tightly integrate the retrieval and reader stages.

multi-hop

11.4 Question Answering Datasets

There are scores of question answering datasets, used both for instruction tuning and for evaluation of the question answering abilities of language models.

We can distinguish the datasets along many dimensions, summarized nicely in Rogers et al. (2023). One is the original purpose of the questions in the data, whether they were natural **information-seeking** questions, or whether they were questions designed for **probing**: evaluating or testing systems or humans.

Natural Questions On the natural side there are datasets like **Natural Questions** (Kwiatkowski et al., 2019), a set of anonymized English queries to the Google search engine and their answers. The answers are created by annotators based on Wikipedia information, and include a paragraph-length long answer and a short span answer. For example the question "When are hops added to the brewing process?" has the short answer *the boiling process* and a long answer which is an entire paragraph from the Wikipedia page on *Brewing*.

MS MARCO A similar natural question set is the **MS MARCO** (Microsoft Machine Reading Comprehension) collection of datasets, including 1 million real anonymized English questions from Microsoft Bing query logs together with a human generated answer and 9 million passages (Bajaj et al., 2016), that can be used both to test retrieval ranking and question answering.

Although many datasets focus on English, natural information-seeking question datasets exist in other languages. The DuReader dataset is a Chinese QA resource based on search engine queries and community QA (He et al., 2018). **TyDi QA** **TyDi QA** dataset contains 204K question-answer pairs from 11 typologically diverse languages, including Arabic, Bengali, Kiswahili, Russian, and Thai (Clark et al., 2020). In the TYDI QA task, a system is given a question and the passages from a Wikipedia article and must (a) select the passage containing the answer (or NULL if no passage contains the answer), and (b) mark the minimal answer span (or NULL).

MMLU On the probing side are datasets like **MMLU** (Massive Multitask Language Understanding), a commonly-used dataset of 15908 knowledge and reasoning questions in 57 areas including medicine, mathematics, computer science, law, and others. MMLU questions are sourced from various exams for humans, such as the US Graduate Record Exam, Medical Licensing Examination, and Advanced Placement exams. So the questions don't represent people's information needs, but rather are designed to test human knowledge for academic or licensing purposes. Fig. 11.10 shows some examples, with the correct answers in bold.

Some of the question datasets described above augment each question with passage(s) from which the answer can be extracted. These datasets were mainly created **reading comprehension** for an earlier QA task called **reading comprehension** in which a model is given a question and a document and is required to extract the answer from the given document. We sometimes call the task of question answering given one or more **open book** documents (for example via RAG), the **open book** QA task, while the task of an- **closed book** swering directly from the LM with no retrieval component at all is the **closed book** QA task.[5] Thus datasets like Natural Questions can be treated as open book if the solver uses each question's attached document, or closed book if the documents are not used, while datasets like MMLU are solely closed book.

Another dimension of variation is the format of the answer: multiple-choice versus freeform. And of course there are variations in prompting, like whether the model is just the question (zero-shot) or also given demonstrations of answers to similar questions (few-shot). MMLU offers both zero-shot and few-shot prompt options.

[5] This repurposes the word for types of exams in which students are allowed to 'open their books' or not.

MMLU examples

College Computer Science
Any set of Boolean operators that is sufficient to represent all Boolean expressions is said to be complete. Which of the following is NOT complete?
(A) AND, NOT
(B) NOT, OR
(C) AND, OR
(D) NAND

College Physics
The primary source of the Sun's energy is a series of thermonuclear reactions in which the energy produced is c^2 times the mass difference between
(A) two hydrogen atoms and one helium atom
(B) four hydrogen atoms and one helium atom
(C) six hydrogen atoms and two helium atoms
(D) three helium atoms and one carbon atom

International Law
Which of the following is a treaty-based human rights mechanism?
(A) The UN Human Rights Committee
(B) The UN Human Rights Council
(C) The UN Universal Periodic Review
(D) The UN special mandates

Prehistory
Unlike most other early civilizations, Minoan culture shows little evidence of
(A) trade.
(B) warfare.
(C) the development of a common religion.
(D) conspicuous consumption by elites.

Figure 11.10 Example problems from MMLU

11.5 Evaluating Question Answering

Three techniques are commonly employed to evaluate question-answering systems, with the choice depending on the type of question and QA situation. For **multiple choice** questions like in MMLU, we report exact match:

> **Exact match**: The % of predicted answers that match the gold answer exactly.

For questions with **free text** answers, like Natural Questions, we commonly evaluated with token **F_1 score** to roughly measure the partial string overlap between the answer and the reference answer:

> **F_1 score**: The average token overlap between predicted and gold answers. Treat the prediction and gold as a bag of tokens, and compute F_1 for each question, then return the average F_1 over all questions.

mean reciprocal rank

MRR

Finally, in some situations QA systems give multiple **ranked** answers. In such cases we evaluated using **mean reciprocal rank**, or **MRR** (Voorhees, 1999). MRR is designed for systems that return a short **ranked** list of answers or passages for each test set question, which we can compare against the (human-labeled) correct answer. First, each test set question is scored with the reciprocal of the **rank** of the first correct answer. For example if the system returned five answers to a question but the first three are wrong (so the highest-ranked correct answer is ranked fourth), the reciprocal rank for that question is $\frac{1}{4}$. The score for questions that return no correct answer is 0. The MRR of a system is the average of the scores for each question in the test set. In some versions of MRR, questions with a score of zero are ignored in this calculation. More formally, for a system returning ranked answers to each question in a test set Q, (or in the alternate version, let Q be the subset of test set questions that have non-zero scores). MRR is then defined as

$$\text{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{11.20}$$

11.6 Summary

This chapter introduced the tasks of **question answering** and **information retrieval**.

- **Question answering** (**QA**) is the task of answering a user's questions.
- We focus in this chapter on the task of retrieval-based question answering, in which the user's questions are intended to be answered by the material in some set of documents (which might be the web).
- **Information Retrieval** (**IR**) is the task of returning documents to a user based on their information need as expressed in a **query**. In ranked retrieval, the documents are returned in ranked order.
- The match between a query and a document can be done by first representing each of them with a sparse vector that represents the frequencies of words, weighted by **tf-idf** or **BM25**. Then the similarity can be measured by cosine.
- Documents or queries can instead be represented by dense vectors, by encoding the question and document with an encoder-only model like BERT, and in that case computing similarity in embedding space.
- The **inverted index** is a storage mechanism that makes it very efficient to find documents that have a particular word.
- Ranked retrieval is generally evaluated by **mean average precision** or **interpolated precision**.
- Question answering systems generally use the **retriever/reader** architecture. In the **retriever** stage, an IR system is given a query and returns a set of documents.
- The **reader** stage is implemented by **retrieval-augmented generation**, in which a large language model is prompted with the query and a set of documents and then conditionally generates a novel answer.
- QA can be evaluated by exact match with a known answer if only a single answer is given, with token F_1 score for free text answers, or with mean reciprocal rank if a ranked set of answers is given.

Historical Notes

Question answering was one of the earliest NLP tasks. By 1961 the BASEBALL system (Green et al., 1961) answered questions about baseball games like "Where did the Red Sox play on July 7" by querying a structured database of game information. The database was stored as a kind of attribute-value matrix with values for attributes of each game:

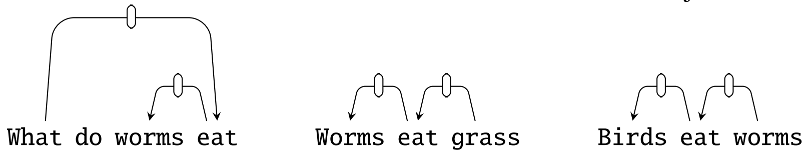
```
 Month = July
    Place = Boston
        Day  = 7
        Game Serial No.  = 96
        (Team = Red Sox, Score = 5)
        (Team = Yankees, Score = 3)
```

Each question was constituency-parsed using the algorithm of Zellig Harris's TDAP project at the University of Pennsylvania, essentially a cascade of finite-state transducers (see the historical discussion in Joshi and Hopely 1999 and Karttunen 1999). Then in a content analysis phase each word or phrase was associated with a program that computed parts of its meaning. Thus the phrase 'Where' had code to assign the semantics `Place = ?`, with the result that the question "Where did the Red Sox play on July 7" was assigned the meaning

```
Place = ?
Team = Red Sox
Month = July
Day = 7
```

The question is then matched against the database to return the answer.

The Protosynthex system of Simmons et al. (1964), given a question, formed a query from the content words in the question, and then retrieved candidate answer sentences in the document, ranked by their frequency-weighted term overlap with the question. The query and each retrieved sentence were then parsed with dependency parsers, and the sentence whose structure best matches the question structure selected. Thus the question *What do worms eat?* would match *worms eat grass*: both have the subject *worms* as a dependent of *eat*, in the version of dependency grammar used at the time, while *birds eat worms* has *birds* as the subject:



Simmons (1965) summarizes other early QA systems.

By the 1970s, systems used predicate calculus as the meaning representation language. The **LUNAR** system (Woods et al. 1972, Woods 1978) was designed to be a natural language interface to a database of chemical facts about lunar geology. It could answer questions like *Do any samples have greater than 13 percent aluminum* by parsing them into a logical form

LUNAR

(TEST (FOR SOME X16 / (SEQ SAMPLES) : T ; (CONTAIN' X16
(NPR* X17 / (QUOTE AL203)) (GREATERTHAN 13 PCT))))

By the 1990s question answering shifted to machine learning. Zelle and Mooney (1996) proposed to treat question answering as a semantic parsing task, by creating the Prolog-based GEOQUERY dataset of questions about US geography. This model was extended by Zettlemoyer and Collins (2005) and 2007. By a decade later, neural models were applied to semantic parsing (Dong and Lapata 2016, Jia and Liang 2016), and then to knowledge-based question answering by mapping text to SQL (Iyer et al., 2017).

[TBD: History of IR.]

Meanwhile, a paradigm for answering questions that drew more on information-retrieval was influenced by the rise of the web in the 1990s. The U.S. government-sponsored TREC (Text REtrieval Conference) evaluations, run annually since 1992, provide a testbed for evaluating information-retrieval tasks and techniques (Voorhees and Harman, 2005). TREC added an influential QA track in 1999, which led to a wide variety of factoid and non-factoid question answering systems competing in annual evaluations.

At that same time, Hirschman et al. (1999) introduced the idea of using children's reading comprehension tests to evaluate machine text comprehension algorithms. They acquired a corpus of 120 passages with 5 questions each designed for 3rd-6th grade children, built an answer extraction system, and measured how well the answers given by their system corresponded to the answer key from the test's publisher. Their algorithm focused on word overlap as a feature; later algorithms added named entity features and more complex similarity between the question and the answer span (Riloff and Thelen 2000, Ng et al. 2000).

The DeepQA component of the Watson Jeopardy! system was a large and sophisticated feature-based system developed just before neural systems became common. It is described in a series of papers in volume 56 of the IBM Journal of Research and Development, e.g., Ferrucci (2012).

Early neural reading comprehension systems drew on the insight common to early systems that answer finding should focus on question-passage similarity. Many of the architectural outlines of these neural systems were laid out in Hermann et al. (2015), Chen et al. (2017), and Seo et al. (2017). These systems focused on datasets like Rajpurkar et al. (2016) and Rajpurkar et al. (2018) and their successors, usually using separate IR algorithms as input to neural reading comprehension systems. The paradigm of using dense retrieval with a span-based reader, often with a single end-to-end architecture, is exemplified by systems like Lee et al. (2019) or Karpukhin et al. (2020). An important research area with dense retrieval for open-domain QA is training data: using self-supervised methods to avoid having to label positive and negative passages (Sachan et al., 2023).

Early work on large language models showed that they stored sufficient knowledge in the pretraining process to answer questions (Petroni et al., 2019; Raffel et al., 2020; Radford et al., 2019; Roberts et al., 2020), at first not competitively with special-purpose question answerers, but quickly surpassing them. Retrieval-augmented generation algorithms were first introduced as a way to improve language modeling word prediction (Khandelwal et al., 2019), but were quickly applied to question answering (Izacard et al., 2022; Ram et al., 2023; Shi et al., 2023).

Exercises

Arora, S., P. Lewis, A. Fan, J. Kahn, and C. Ré. 2023. Reasoning over public and private data in retrieval-based systems. *TACL*, 11:902–921.

Babbage, C. 1864. *Passages from the Life of a Philosopher*. Longman.

Bajaj, P., D. Campos, N. Craswell, L. Deng, J. G. ando Xiaodong Liu, R. Majumder, A. McNamara, B. Mitra, T. Nguye, M. Rosenberg, X. Song, A. Stoica, S. Tiwary, and T. Wang. 2016. MS MARCO: A human generated MAchine Reading COmprehension dataset. *NeurIPS*.

Chen, D., A. Fisch, J. Weston, and A. Bordes. 2017. Reading Wikipedia to answer open-domain questions. *ACL*.

Clark, J. H., E. Choi, M. Collins, D. Garrette, T. Kwiatkowski, V. Nikolaev, and J. Palomaki. 2020. TyDi QA: A benchmark for information-seeking question answering in typologically diverse languages. *TACL*, 8:454–470.

Dahl, M., V. Magesh, M. Suzgun, and D. E. Ho. 2024. Large legal fictions: Profiling legal hallucinations in large language models. *Journal of Legal Analysis*, 16:64–93.

Deerwester, S. C., S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. 1990. Indexing by latent semantics analysis. *JASIS*, 41(6):391–407.

Dong, L. and M. Lapata. 2016. Language to logical form with neural attention. *ACL*.

Ferrucci, D. A. 2012. Introduction to “This is Watson”. *IBM Journal of Research and Development*, 56(3/4):1:1–1:15.

Furnas, G. W., T. K. Landauer, L. M. Gomez, and S. T. Dumais. 1987. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971.

Geva, M., R. Schuster, J. Berant, and O. Levy. 2021. Transformer feed-forward layers are key-value memories. *EMNLP*.

Green, B. F., A. K. Wolf, C. Chomsky, and K. Laughery. 1961. Baseball: An automatic question answerer. *Proceedings of the Western Joint Computer Conference 19*.

He, W., K. Liu, J. Liu, Y. Lyu, S. Zhao, X. Xiao, Y. Liu, Y. Wang, H. Wu, Q. She, X. Liu, T. Wu, and H. Wang. 2018. DuReader: a Chinese machine reading comprehension dataset from real-world applications. *Workshop on Machine Reading for Question Answering*.

Hermann, K. M., T. Kocisky, E. Grefenstette, L. Espeholt, W. Kay, M. Suleyman, and P. Blunsom. 2015. Teaching machines to read and comprehend. *NeurIPS*.

Hirschman, L., M. Light, E. Breck, and J. D. Burger. 1999. Deep Read: A reading comprehension system. *ACL*.

Iyer, S., I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. *ACL*.

Izacard, G., P. Lewis, M. Lomeli, L. Hosseini, F. Petroni, T. Schick, J. Dwivedi-Yu, A. Joulin, S. Riedel, and E. Grave. 2022. Few-shot learning with retrieval augmented language models. ArXiv preprint.

Jia, R. and P. Liang. 2016. Data recombination for neural semantic parsing. *ACL*.

Jiang, C., B. Qi, X. Hong, D. Fu, Y. Cheng, F. Meng, M. Yu, B. Zhou, and J. Zhou. 2024. On large language models’ hallucination with regard to known facts. *NAACL HLT*.

Johnson, J., M. Douze, and H. Jégou. 2017. Billion-scale similarity search with GPUs. ArXiv preprint arXiv:1702.08734.

Joshi, A. K. and P. Hopely. 1999. A parser from antiquity. In A. Kornai, ed., *Extended Finite State Models of Language*, 6–15. Cambridge University Press.

Jurafsky, D. 2014. *The Language of Food*. W. W. Norton, New York.

Kamphuis, C., A. P. de Vries, L. Boytsov, and J. Lin. 2020. Which BM25 do you mean? a large-scale reproducibility study of scoring variants. *European Conference on Information Retrieval*.

Karpukhin, V., B. Oğuz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih. 2020. Dense passage retrieval for open-domain question answering. *EMNLP*.

Karttunen, L. 1999. Comments on Joshi. In A. Kornai, ed., *Extended Finite State Models of Language*, 16–18. Cambridge University Press.

Khandelwal, U., O. Levy, D. Jurafsky, L. Zettlemoyer, and M. Lewis. 2019. Generalization through memorization: Nearest neighbor language models. *ICLR*.

Khattab, O., C. Potts, and M. Zaharia. 2021. Relevance-guided supervision for OpenQA with ColBERT. *TACL*, 9:929–944.

Khattab, O. and M. Zaharia. 2020. ColBERT: Efficient and effective passage search via contextualized late interaction over BERT. *SIGIR*.

Kwiatkowski, T., J. Palomaki, O. Redfield, M. Collins, A. Parikh, C. Alberti, D. Epstein, I. Polosukhin, J. Devlin, K. Lee, K. Toutanova, L. Jones, M. Kelcey, M.-W. Chang, A. M. Dai, J. Uszkoreit, Q. Le, and S. Petrov. 2019. Natural questions: A benchmark for question answering research. *TACL*, 7:452–466.

Lee, K., M.-W. Chang, and K. Toutanova. 2019. Latent retrieval for weakly supervised open domain question answering. *ACL*.

Manning, C. D., P. Raghavan, and H. Schütze. 2008. *Introduction to Information Retrieval*. Cambridge.

Meng, K., D. Bau, A. Andonian, and Y. Belinkov. 2022. Locating and editing factual associations in GPT. *NeurIPS*, volume 36.

Ng, H. T., L. H. Teo, and J. L. P. Kwan. 2000. A machine learning approach to answering questions for reading comprehension tests. *EMNLP*.

Petroni, F., T. Rocktäschel, S. Riedel, P. Lewis, A. Bakhtin, Y. Wu, and A. Miller. 2019. Language models as knowledge bases? *EMNLP*.

Radford, A., J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. 2019. Language models are unsupervised multitask learners. OpenAI tech report.

Raffel, C., N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR*, 21(140):1–67.

Rajpurkar, P., R. Jia, and P. Liang. 2018. Know what you don’t know: Unanswerable questions for SQuAD. *ACL*.

Rajpurkar, P., J. Zhang, K. Lopyrev, and P. Liang. 2016. SQuAD: 100,000+ questions for machine comprehension of text. *EMNLP*.

Ram, O., Y. Levine, I. Dalmedigos, D. Muhlgay, A. Shashua, K. Leyton-Brown, and Y. Shoham. 2023. In-context retrieval-augmented language models. ArXiv preprint.

Riloff, E. and M. Thelen. 2000. A rule-based question answering system for reading comprehension tests. *ANLP/NAACL workshop on reading comprehension tests*.

Roberts, A., C. Raffel, and N. Shazeer. 2020. How much knowledge can you pack into the parameters of a language model? *EMNLP*.

Robertson, S., S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford. 1995. Okapi at TREC-3. *Overview of the Third Text REtrieval Conference (TREC-3)*.

Rogers, A., M. Gardner, and I. Augenstein. 2023. QA dataset explosion: A taxonomy of NLP resources for question answering and reading comprehension. *ACM Computing Surveys*, 55(10):1–45.

Sachan, D. S., M. Lewis, D. Yogatama, L. Zettlemoyer, J. Pineau, and M. Zaheer. 2023. Questions are all you need to train a dense passage retriever. *TACL*, 11:600–616.

Salton, G. 1971. *The SMART Retrieval System: Experiments in Automatic Document Processing*. Prentice Hall.

Seo, M., A. Kembhavi, A. Farhadi, and H. Hajishirzi. 2017. Bidirectional attention flow for machine comprehension. *ICLR*.

Shi, W., S. Min, M. Yasunaga, M. Seo, R. James, M. Lewis, L. Zettlemoyer, and W.-t. Yih. 2023. REPLUG: Retrieval-augmented black-box language models. ArXiv preprint.

Simmons, R. F. 1965. Answering English questions by computer: A survey. *CACM*, 8(1):53–70.

Simmons, R. F., S. Klein, and K. McConlogue. 1964. Indexing and dependency logic for answering English questions. *American Documentation*, 15(3):196–204.

Sparck Jones, K. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21.

Voorhees, E. M. 1999. TREC-8 question answering track report. *Proceedings of the 8th Text Retrieval Conference*.

Voorhees, E. M. and D. K. Harman. 2005. *TREC: Experiment and Evaluation in Information Retrieval*. MIT Press.

Woods, W. A. 1978. Semantics and quantification in natural language question answering. In M. Yovits, ed., *Advances in Computers*, 2–64. Academic.

Woods, W. A., R. M. Kaplan, and B. L. Nash-Webber. 1972. The lunar sciences natural language information system: Final report. Technical Report 2378, BBN.

Zelle, J. M. and R. J. Mooney. 1996. Learning to parse database queries using inductive logic programming. *AAAI*.

Zettlemoyer, L. and M. Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *Uncertainty in Artificial Intelligence, UAI'05*.

Zettlemoyer, L. and M. Collins. 2007. Online learning of relaxed CCG grammars for parsing to logical form. *EMNLP/CoNLL*.

Zhou, K., J. Hwang, X. Ren, and M. Sap. 2024. Relying on the unreliable: The impact of language models' reluctance to express uncertainty. *ACL*.