Speech and Language Processing. Daniel Jurafsky & James H. Martin. Copyright © 2023. All rights reserved. Draft of January 28, 2024.

CHAPTER

14 Question Answering and Information Retrieval

The quest for knowledge is deeply human, and so it is not surprising that practically as soon as there were computers we were asking them questions. By the early 1960s, systems used the two major paradigms of question answering—**retrieval-based** and **knowledge-based**—to answer questions about baseball statistics or scientific facts. Even imaginary computers got into the act. Deep Thought, the computer that Douglas Adams invented in *The Hitchhiker's Guide to the Galaxy*, managed to answer "the Ultimate Question Of Life, The Universe, and Everything".[1] In 2011, IBM's Watson question-answering system won the TV game-show *Jeopardy!*, 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]

Question answering systems are designed to fill human information needs that might arise in situations like talking to a virtual assistant or a chatbot, interacting with a search engine, or querying a database. Question answering systems often focus on a particular subset of these information needs: **factoid questions**, questions that can be answered with simple facts expressed in short texts, like the following:

(14.1) Where is the Louvre Museum located?

(14.2) What is the average age of the onset of autism?

One way to do question answering is just to directly ask a large language model. For example, we could use the techniques of Chapter 12, prompting a large pretrained causal language model with a string like

Q: Where is the Louvre Museum located? A:

have it do conditional generation given this prefix, and take the response as the answer. The idea is that huge pretrained language models have read a lot of facts in their pretraining data, presumably including the location of the Louvre, and have encoded this information in their parameters.

For some general factoid questions this can be a useful approach and is used in practice. But prompting a large language model is not yet a solution for question answering. The main problem is that large language models often give the wrong answer! 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 simply 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 had hallucination rates ranging from 69% to 88%.

hallucinate

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

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

Sometime there are ways to tell that language models are hallucinating, but often there aren't. One problem is that language model estimates of their confidence in their answers 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 the 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.

calibrated

A second problem is that simply prompting a large language model doesn't allow us to ask questions about proprietary data. A common use of question-answering is to query private data, like asking an assistant about our email or private documents, or asking a question about our own 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. Furthermore, the use of internal datasets, or even the web itself, can be especially useful for rapidly changing or dynamic information; by contrast, large language models are often only released at long increments of many months and so may not have up-to-date information.

For this reason the current dominant solution for question-answering is the two-stage **retriever/reader** model (Chen et al., 2017), and that is the method we will focus on in this chapter. In a retriever/reader model, we use **information retrieval** techniques to first retrieve documents that are likely to have information that might help answer the question. Then we either **extract** an answer from spans of text in the documents, or use large language models to **generate** an answer given these documents, sometimes called **retrieval-augmented generation**,

Basing our answers on retrieved documents can solve the above-mentioned problems with using simple prompting to answer questions. First, we can ensure that the answer is grounded in facts from some curated dataset. And we can give 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 we can use our retrieval techniques 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, as well as modern neural IR using dense retriever, in which we run documents through BERT or other language models to get neural representations, and use cosine between dense representations of the query and document.

We then introduce retriever-based question answering, via the **retriever/reader** model. This algorithm most commonly relies on the vast amount of text on the web, in which case it is sometimes called **open domain QA**, or on collections of proprietary data, or scientific papers like PubMed. We'll go through the two types of **readers**, span extractors and retrieval-augmented generation.

14.1 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

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. 14.1.

document collection term query

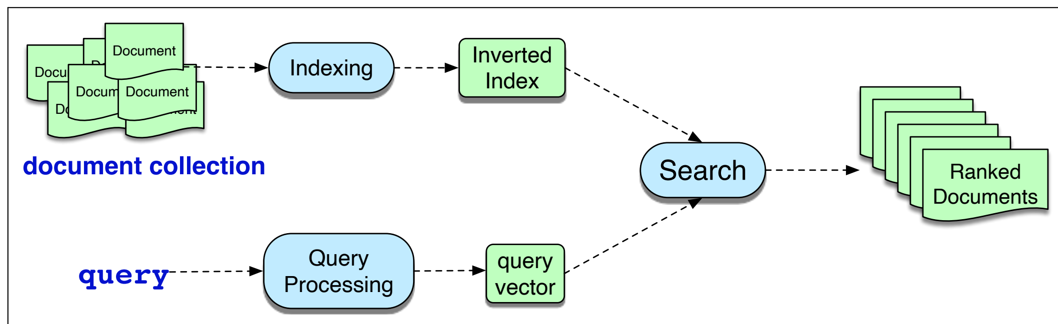


Figure 14.1 The architecture of an ad hoc IR system.

The basic IR architecture uses the vector space model we introduced in Chapter 6, 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 Chapter 4, since words are considered independently of their positions.

14.1.1 Term weighting and document scoring

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

term weight

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 6, and a slightly more powerful variant called **BM25**.

BM25

We'll reintroduce tf-idf here so readers don't need to look back at Chapter 6. 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]

$$\mathrm{tf}_{t,d} = \begin{cases} 1+\log_{10}\mathrm{count}(t,d) & \text{if } \mathrm{count}(t,d) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{14.3}$$

If we use log weighting, terms which occur 0 times in a document would have $\mathrm{tf}=0$,

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

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:

$$\text{idf}_t = \log_{10} \frac{N}{\text{df}_t} \tag{14.4}$$

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

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

$$\text{tf-idf}(t,d) = \text{tf}_{t,d} \cdot \text{idf}_t \tag{14.5}$$

14.1.2 Document Scoring

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

$$\text{score}(q,d) = \cos(\mathbf{q},\mathbf{d}) = \frac{\mathbf{q} \cdot \mathbf{d}}{|\mathbf{q}||\mathbf{d}|} \tag{14.6}$$

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:

$$\text{score}(q,d) = \cos(\mathbf{q},\mathbf{d}) = \frac{\mathbf{q}}{|\mathbf{q}|} \cdot \frac{\mathbf{d}}{|\mathbf{d}|} \tag{14.7}$$

[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).

We can spell out Eq. 14.7, using the tf-idf values and spelling out the dot product as a sum of products:

$$\text{score}(q,d) = \sum_{t \in \mathbf{q}} \frac{\text{tf-idf}(t,q)}{\sqrt{\sum_{q_i \in q} \text{tf-idf}^2(q_i,q)}} \cdot \frac{\text{tf-idf}(t,d)}{\sqrt{\sum_{d_i \in d} \text{tf-idf}^2(d_i,d)}} \tag{14.8}$$

Now let's use (14.8) 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. 14.2 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. 14.3, Eq. 14.4, Eq. 14.5, and Eq. 14.8 (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.

In practice, there are many variants and approximations to Eq. 14.8. 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. 14.8 to explicitly mention the tf and idf terms from (14.5):

$$\text{score}(q,d) = \sum_{t \in \mathbf{q}} \frac{\text{tf}_{t,q} \cdot \text{idf}_t}{\sqrt{\sum_{q_i \in q} \text{tf-idf}^2(q_i,q)}} \cdot \frac{\text{tf}_{t,d} \cdot \text{idf}_t}{\sqrt{\sum_{d_i \in d} \text{tf-idf}^2(d_i,d)}} \tag{14.9}$$

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:

$$\text{score}(q,d) = \sum_{t \in \mathbf{q}} \frac{\text{tf}_{t,q} \cdot \text{idf}_t}{\sqrt{\sum_{q_i \in q} \text{tf-idf}^2(q_i,q)}} \cdot \frac{\text{tf}_{t,d} \cdot \text{idf}_t}{\sqrt{\sum_{d_i \in d} \text{tf-idf}^2(d_i,d)}} \tag{14.10}$$

Other variants of tf-idf eliminate various other terms.

BM25

A slightly more complex variant in the tf-idf family is the **BM25** weighting

				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**	**× q**	**cnt**	**tf**	**tf-idf**	**n'lized**	**×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 14.2 Computation of tf-idf cosine score between the query and nano-documents 1 (0.747) and 2 (0.0779), using Eq. 14.3, Eq. 14.4, Eq. 14.5 and Eq. 14.8.

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:

$$\sum_{t \in q} \overbrace{\log\left(\frac{N}{df_t}\right)}^{\text{IDF}} \overbrace{\frac{tf_{t,d}}{k\left(1-b+b\left(\frac{|d|}{|d_{\text{avg}}|}\right)\right)+tf_{t,d}}}^{\text{weighted tf}} \tag{14.11}$$

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 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

stop list

words that occur in every document. Nonetheless, stop word removal is occasionally useful in various NLP tasks so is worth keeping in mind.

14.1.3 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.

postings

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

14.1.4 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:

$$Precision = \frac{|R|}{|T|} \quad Recall = \frac{|R|}{|U|} \tag{14.12}$$

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	Precision_{Rank}	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 14.3 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. 14.3 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 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. 14.4 for the data in table 14.3.

precision-recall curve

Fig. 14.4 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

$$\text{IntPrecision}(r) = \max_{i>=r} \text{Precision}(i) \tag{14.13}$$

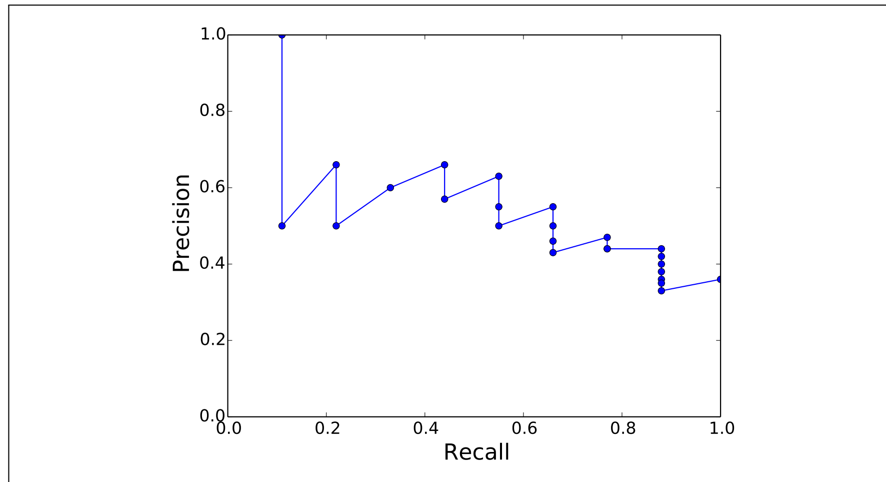


Figure 14.4 The precision recall curve for the data in table 14.3.

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. 14.5 and Fig. 14.6 show the resulting interpolated data points from our example.

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 14.5 Interpolated data points from Fig. 14.3.

Given curves such as that in Fig. 14.6 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. 14.3). For a single 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

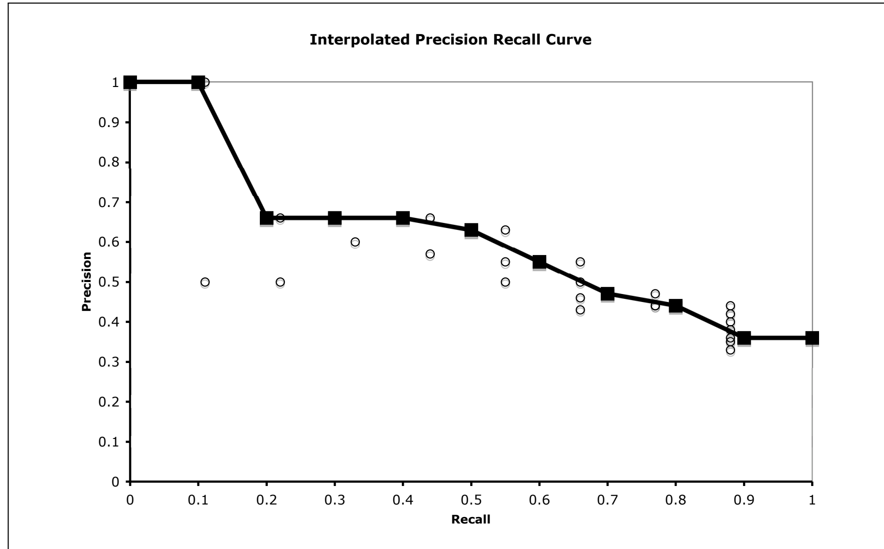


Figure 14.6 An 11 point interpolated precision-recall curve. Precision at each of the 11 standard recall levels is interpolated for each query from the maximum at any higher level of recall. The original measured precision recall points are also shown.

documents at or above r, then the **average precision** (**AP**) for a single query is

$$\text{AP} = \frac{1}{|R_r|} \sum_{d \in R_r} \text{Precision}_r(d) \tag{14.14}$$

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:

$$\text{MAP} = \frac{1}{|Q|} \sum_{q \in Q} \text{AP}(q) \tag{14.15}$$

The MAP for the single query (hence = AP) in Fig. 14.3 is 0.6.

14.2 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:

$$\mathbf{z} = \text{BERT}(\text{q}; \texttt{[SEP]}; \text{d})\texttt{[CLS]}$$
$$\text{score}(q,d) = \text{softmax}(\mathbf{U}(\mathbf{z})) \tag{14.16}$$

This architecture is shown in Fig. 14.7a. 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.

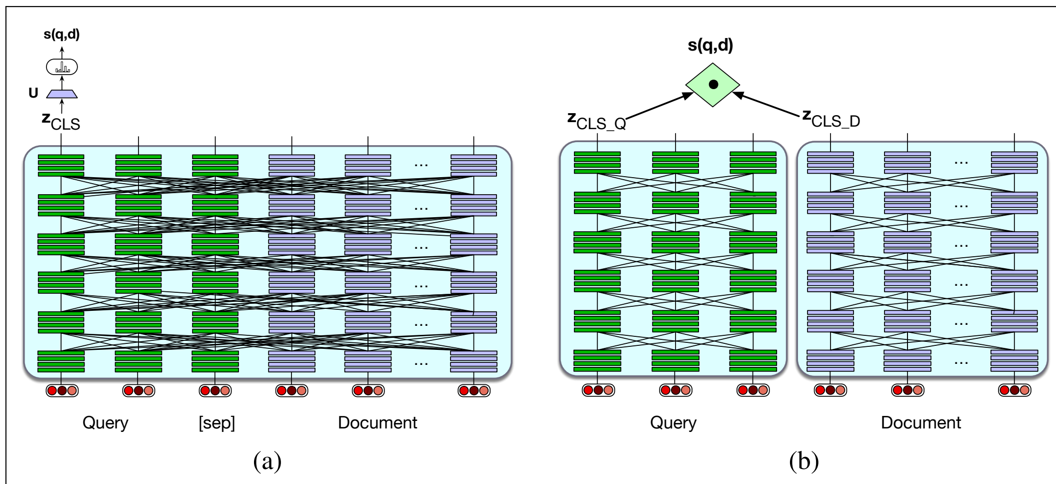


Figure 14.7 Two ways to do dense retrieval, illustrated by using lines between layers to schematically represent self-attention: (a) Use a single encoder to jointly encode query and document and finetune to produce a relevance score with a linear layer over the CLS token. This is too compute-expensive 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.

The problem with the full BERT architecture in Fig. 14.7a is the expense in computation and time. With this architecture, every time we get a query, we have to pass every single 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. 14.7b). 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):

$$\begin{aligned} \mathbf{z}_q &= \text{BERT}_Q(\text{q})\,\texttt{[CLS]} \\ \mathbf{z}_d &= \text{BERT}_D(\text{d})\,\texttt{[CLS]} \\ \text{score}(q,d) &= \mathbf{z}_q \cdot \mathbf{z}_d \end{aligned} \tag{14.17}$$

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 possible 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. 14.8. 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 $[q_1,\ldots,q_n]$, 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 $\mathbf{q} = [\mathbf{q}_1,\ldots,\mathbf{q}_N]$. The passage d with tokens $[d_1,\ldots,d_m]$, 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:

$$\text{score}(q,d) = \sum_{i=1}^{N} \max_{j=1}^{m} \mathbf{E}_{q_i} \cdot \mathbf{E}_{d_j} \tag{14.18}$$

While the interaction mechanism has no tunable parameters, the ColBERT architecture 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 (14.18), 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 14.3.1) 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

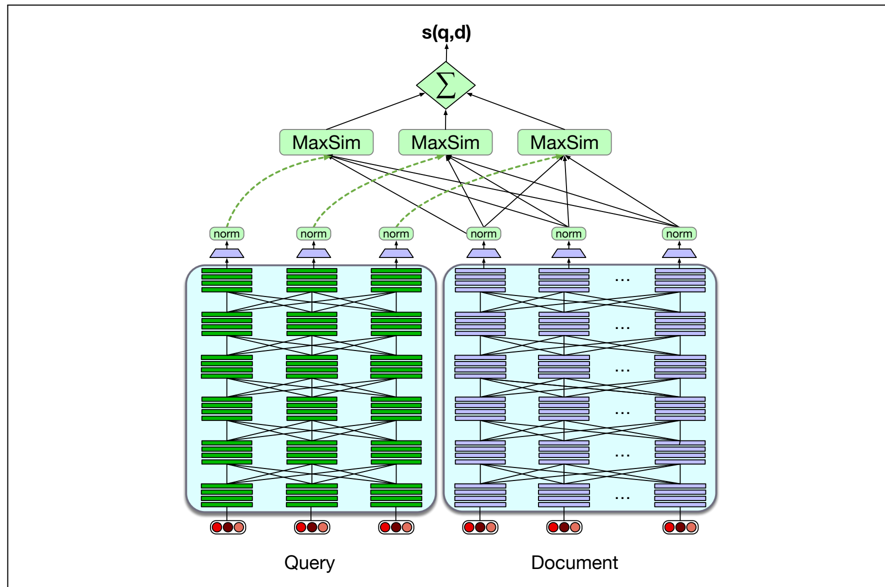


Figure 14.8 A sketch of the ColBERT algorithm at inference time. The query and document are first passed through separate BERT encoders. Similarity between query and document is computed by summing a soft alignment between the contextual representations of tokens in the query and the document. Training 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).

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

14.3 Using Neural IR for Question Answering

retrieval-based QA

The goal of **retrieval-based QA** (sometimes called **open domain QA**) is to answer a user's question by either finding short text segments from the web or some other large collection of documents, or by generating an answer based on them. Figure 14.9 shows some sample factoid questions with answers.

retrieve and read

The dominant paradigm for retrieval-based QA is sometimes called the **retrieve and read** model shown in Fig. 14.10. In the first stage of this 2-stage model we retrieve relevant passages from a text collection, for example using the dense retrievers of the previous section.

The second stage, called the **reader**, is commonly implemented as either an **ex-**

Question	Answer
Where is the Louvre Museum located?	in Paris, France
What are the names of Odin's ravens?	Huginn and Muninn
What kind of nuts are used in marzipan?	almonds
What instrument did Max Roach play?	drums
What's the official language of Algeria?	Arabic

Figure 14.9 Some factoid questions and their answers.

tractor or a **generator**. The first method is **span extraction**, using a neural **reading comprehension** algorithm that passes over each passage and is trained to find spans of text that answer the question. The second method is also known as **retrieval-augmented generation**: we take a large pretrained language model, give it some set of retrieved passages and other text as its prompt, and autoregressively generate a new answer token by token.

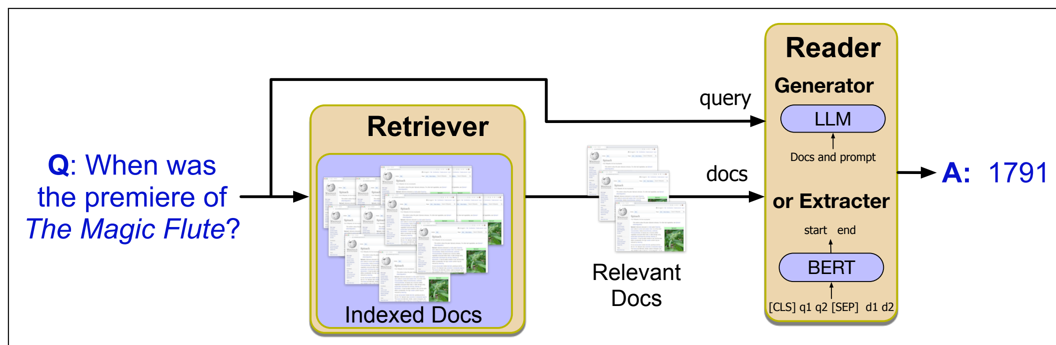


Figure 14.10 Retrieval-based question answering has two stages: **retrieval**, which returns relevant documents from the collection, and **reading**, in which a neural reading comprehension system **extracts** answer spans, or a large pretrained language model that **generates** answers autoregressively given the documents as a prompt.

In the next few sections we'll describe these two standard **reader** algorithms. But first, we'll introduce some commonly-used question answering datasets.

14.3.1 Retrieval-based QA: Datasets

Datasets for retrieval-based QA are most commonly created by first developing **reading comprehension datasets** containing tuples of *(passage, question, answer)*. Reading comprehension systems can use the datasets to train a reader that is given a passage and a question, and predicts a span in the passage as the answer. Including the passage from which the answer is to be extracted eliminates the need for reading comprehension systems to deal with IR.

SQuAD

For example the Stanford Question Answering Dataset (**SQuAD**) consists of passages from Wikipedia and associated questions whose answers are spans from the passage (Rajpurkar et al. 2016). Squad 2.0 in addition adds some questions that are designed to be unanswerable (Rajpurkar et al. 2018), with a total of just over 150,000 questions. Fig. 14.11 shows a (shortened) excerpt from a SQUAD 2.0 passage together with three questions and their gold answer spans.

SQuAD was built by having humans read a given Wikipedia passage, write questions about the passage, and choose a specific answer span.

HotpotQA

Other datasets are created by similar techniques but try to make the questions more complex. The **HotpotQA** dataset (Yang et al., 2018) was created by showing

Beyoncé Giselle Knowles-Carter (born September 4, 1981) is an American singer, songwriter, record producer and actress. Born and raised in Houston, Texas, she performed in various singing and dancing competitions as a child, and rose to fame in the late 1990s as lead singer of R&B girl-group Destiny's Child. Managed by her father, Mathew Knowles, the group became one of the world's best-selling girl groups of all time. Their hiatus saw the release of Beyoncé's debut album, Dangerously in Love (2003), which established her as a solo artist worldwide, earned five Grammy Awards and featured the Billboard Hot 100 number-one singles "Crazy in Love" and "Baby Boy".
Q: "In what city and state did Beyoncé grow up?" A: "Houston, Texas"
Q: "What areas did Beyoncé compete in when she was growing up?" A: "singing and dancing"
Q: "When did Beyoncé release Dangerously in Love?" A: "2003"

Figure 14.11 A (Wikipedia) passage from the SQuAD 2.0 dataset (Rajpurkar et al., 2018) with 3 sample questions and the labeled answer spans.

crowd workers multiple context documents and asked to come up with questions that require reasoning about all of the documents.

The fact that questions in datasets like SQuAD or HotpotQA are created by annotators who have first read the passage may make their questions easier to answer, since the annotator may (subconsciously) make use of words from the answer text.

A solution to this possible bias is to make datasets from questions that were not written with a passage in mind. The **TriviaQA** dataset (Joshi et al., 2017) contains 94K questions written by trivia enthusiasts, together with supporting documents from Wikipedia and the web resulting in 650K question-answer-evidence triples.

MS MARCO

MS MARCO (Microsoft Machine Reading Comprehension) is a collection of datasets, including 1 million real anonymized questions from Microsoft Bing query logs together with a human generated answer and 9 million passages (Nguyen et al., 2016), that can be used both to test retrieval ranking and question answering. The **Natural Questions** dataset (Kwiatkowski et al., 2019) similarly incorporates real anonymized queries to the Google search engine. Annotators are presented a query, along with a Wikipedia page from the top 5 search results, and annotate a paragraph-length long answer and a short span answer, or mark null if the text doesn't contain the paragraph. For example the question "When are hops added to the brewing process?" has the short answer *the boiling process* and a long answer which the surrounding entire paragraph from the Wikipedia page on *Brewing*. In using this dataset, a reading comprehension model is given a question and a Wikipedia page and must return a long answer, short answer, or 'no answer' response.

Natural Questions

TyDi QA

The above datasets are all in English. The **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). Many questions have no answer. The various languages in the dataset bring up challenges for QA systems like morphological variation between the question and the answer, or complex issue with word segmentation or multiple alphabets.

In the reading comprehension task, a system is given a question and the passage in which the answer should be found. In the full two-stage QA task, however, sys-

tems are not given a passage, but are required to do their own retrieval from some document collection. A common way to create open-domain QA datasets is to modify a reading comprehension dataset. For research purposes this is most commonly done by using QA datasets that annotate Wikipedia (like SQuAD or HotpotQA). For training, the entire *(question, passage, answer)* triple is used to train the reader. But at inference time, the passages are removed and system is given only the question, together with access to the entire Wikipedia corpus. The system must then do IR to find a set of pages and then read them.

14.3.2 Reader algorithms: Answer Span Extraction

The job of the **reader** is to take a passage as input and produce the answer. Here we introduce the **span extraction** style of reader, in which the answer is a span of text in the passage. For example given a question like *"How tall is Mt. Everest?"* and a passage that contains the clause *Reaching 29,029 feet at its summit*, a reader will output *29,029 feet*.

span

The answer extraction task is commonly modeled by **span labeling**: identifying in the passage a **span** (a continuous string of text) that constitutes an answer. Neural algorithms for reading comprehension are given a question q of n tokens $q_1,...,q_n$ and a passage p of m tokens $p_1,...,p_m$. Their goal is thus to compute the probability $P(a|q,p)$ that each possible span a is the answer.

If each span a starts at position a_s and ends at position a_e, we make the simplifying assumption that this probability can be estimated as $P(a|q,p) = P_{\text{start}}(a_s|q,p)P_{\text{end}}(a_e|q,p)$. Thus for for each token p_i in the passage we'll compute two probabilities: $p_{\text{start}}(i)$ that p_i is the start of the answer span, and $p_{\text{end}}(i)$ that p_i is the end of the answer span.

A standard baseline algorithm for reading comprehension is to pass the question and passage to any encoder like BERT (Fig. 14.12), as strings separated with a [SEP] token, resulting in an encoding token embedding for every passage token p_i.

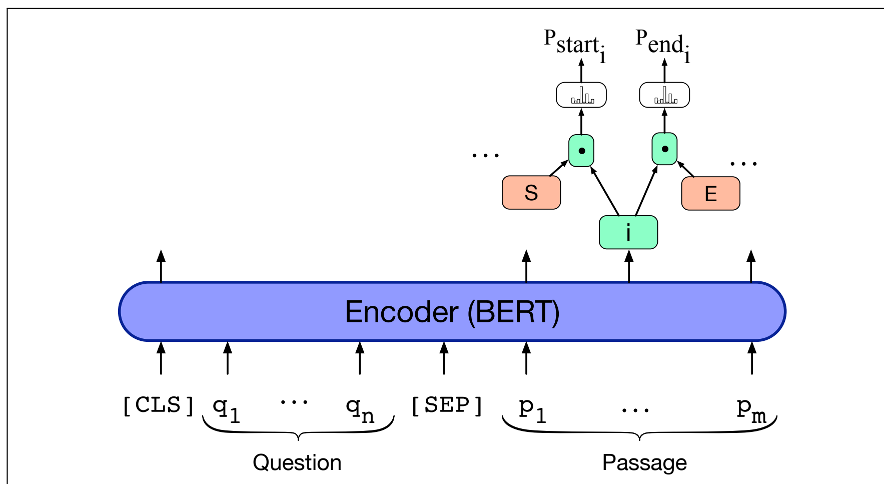


Figure 14.12 An encoder model (using BERT) for span-based question answering from reading-comprehension-based question answering tasks.

For span-based question answering, we represent the question as the first sequence and the passage as the second sequence. We'll also need to add a linear layer that will be trained in the fine-tuning phase to predict the start and end position of the

span. We'll add two new special vectors: a span-start embedding S and a span-end embedding E, which will be learned in fine-tuning. To get a span-start probability for each output token p'_i, we compute the dot product between S and p'_i and then use a softmax to normalize over all tokens p'_i in the passage:

$$P_{\text{start}_i} = \frac{\exp(S \cdot p'_i)}{\sum_j \exp(S \cdot p'_j)} \tag{14.19}$$

We do the analogous thing to compute a span-end probability:

$$P_{\text{end}_i} = \frac{\exp(E \cdot p'_i)}{\sum_j \exp(E \cdot p'_j)} \tag{14.20}$$

The score of a candidate span from position i to j is $S \cdot p'_i + E \cdot p'_j$, and the highest scoring span in which $j \geq i$ is chosen is the model prediction.

The training loss for fine-tuning is the negative sum of the log-likelihoods of the correct start and end positions for each instance:

$$L = -\log P_{\text{start}_i} - \log P_{\text{end}_i} \tag{14.21}$$

Many datasets (like SQuAD 2.0 and Natural Questions) also contain (question, passage) pairs in which the answer is not contained in the passage. We thus also need a way to estimate the probability that the answer to a question is not in the document. This is standardly done by treating questions with no answer as having the [CLS] token as the answer, and hence the answer span start and end index will point at [CLS] (Devlin et al., 2019).

For many datasets the annotated documents/passages are longer than the maximum 512 input tokens BERT allows, such as Natural Questions whose gold passages are full Wikipedia pages. In such cases, following Alberti et al. (2019), we can create multiple pseudo-passage observations from the labeled Wikipedia page. Each observation is formed by concatenating [CLS], the question, [SEP], and tokens from the document. We walk through the document, sliding a window of size 512 (or rather, 512 minus the question length n minus special tokens) and packing the window of tokens into each next pseudo-passage. The answer span for the observation is either labeled [CLS] (= no answer in this particular window) or the gold-labeled span is marked. The same process can be used for inference, breaking up each retrieved document into separate observation passages and labeling each observation. The answer can be chosen as the span with the highest probability (or nil if no span is more probable than [CLS]).

14.3.3 Reader algorithms: Retrieval-Augmented Generation

The second 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**.

retrieval-augmented generation 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 $\mathrm{R}(q)$. For example, we could have a prompt like:

```
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,

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

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. Finally, 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).

multi-hop

14.4 Evaluating Retrieval-based Question Answering

mean reciprocal rank
MRR

Question answering is commonly 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

$$\text{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{14.22}$$

Alternatively, question answering systems can be evaluated with exact match, or with F_1 score. This is common for datasets like SQuAD which are evaluated (first ignoring punctuation and articles like *a, an, the*) via both (Rajpurkar et al., 2016):

- **Exact match**: The % of predicted answers that match the gold answer exactly.
- **F_1 score**: The average word/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.

Other recent datasets include the AI2 Reasoning Challenge (ARC) (Clark et al., 2018) of multiple choice questions designed to be hard to answer from simple lexical methods, like this question

> Which property of a mineral can be determined just by looking at it?
> (A) luster [correct] (B) mass (C) weight (D) hardness

in which the correct answer *luster* is unlikely to co-occur frequently with phrases like *looking at it*, while the word *mineral* is highly associated with the incorrect answer *hardness*.

14.5 Summary

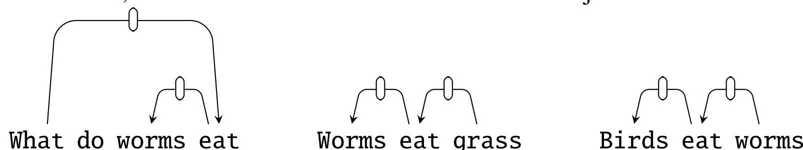
This chapter introduced the tasks of **question answering** and **information retrieval**.

- **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.
- **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 an 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 can either be a span-based **extractor**, that predicts a span of text in the retrieved documents to return as the answer, or a **retrieval-augmented generator**, in which a large language model is used to generate a novel answer after reading the documents and the query.
- QA can be evaluated by exact match with a known answer if only a single answer is given, or with mean reciprocal rank if a ranked set of answers is given.

Bibliographical and Historical Notes

Question answering was one of the earliest NLP tasks, and early versions of the text-based and knowledge-based paradigms were developed by the very early 1960s. The text-based algorithms generally relied on simple parsing of the question and of the sentences in the document, and then looking for matches. This approach was used very early on (Phillips, 1960) but perhaps the most complete early system, and one that strikingly prefigures modern relation-based systems, was the Protosynthex system of Simmons et al. (1964). Given a question, Protosynthex first 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:



The alternative knowledge-based paradigm was implemented in the BASEBALL system (Green et al., 1961). This system 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. Simmons (1965) summarizes other early QA systems.

Another important progenitor of the knowledge-based paradigm for question-answering is work that 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 a couple decades later, drawing on new machine learning approaches in NLP, Zelle and Mooney (1996) proposed to treat knowledge-based QA 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).

Meanwhile, the information-retrieval paradigm for question answering 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 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 com-

mon. It is described in a series of papers in volume 56 of the IBM Journal of Research and Development, e.g., Ferrucci (2012).

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 modern 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). Retrieval-augmented generation algorithms were first introduced as a way to improve language modeling (Khandelwal et al., 2019), but were quickly applied to question answering (Izacard et al., 2022; Ram et al., 2023; Shi et al., 2023).

Exercises

Alberti, C., K. Lee, and M. Collins. 2019. A BERT baseline for the natural questions. http://arxiv.org/abs/1901.08634.

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.

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.

Clark, P., I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord. 2018. Think you have solved question answering? Try ARC, the AI2 reasoning challenge. ArXiv preprint arXiv:1803.05457.

Dahl, M., V. Magesh, M. Suzgun, and D. E. Ho. 2024. Large legal fictions: Profiling legal hallucinations in large language models. ArXiv preprint.

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.

Devlin, J., M.-W. Chang, K. Lee, and K. Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. *NAACL HLT*.

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.

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*.

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*.

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, editor, *Extended Finite State Models of Language*, pages 6–15. Cambridge University Press.

Joshi, M., E. Choi, D. S. Weld, and L. Zettlemoyer. 2017. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. *ACL*.

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, editor, *Extended Finite State Models of Language*, pages 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.

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

Nguyen, T., M. Rosenberg, X. Song, J. Gao, S. Tiwary, R. Majumder, and L. Deng. 2016. Ms marco: A human generated machine reading comprehension dataset. *NeurIPS*.

Phillips, A. V. 1960. A question-answering routine. Technical Report 16, MIT AI Lab.

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*.

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)*.

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, editor, *Advances in Computers*, pages 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.

Yang, Z., P. Qi, S. Zhang, Y. Bengio, W. Cohen, R. Salakhutdinov, and C. D. Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. *EMNLP*.

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*.