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—**information-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, interacting with a search engine, or querying a database. Most question answering systems 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?

In this chapter we describe the two major paradigms for factoid question answering. **Information-retrieval (IR) based QA**, sometimes called **open domain QA**, relies on the vast amount of text on the web or in collections of scientific papers like PubMed. Given a user question, information retrieval is used to find relevant passages. Then neural **reading comprehension** algorithms read these retrieved passages and draw an answer directly from **spans of text**.

In the second paradigm, **knowledge-based question answering**, a system instead builds a semantic representation of the query, such as mapping *What states border Texas?* to the logical representation: $\lambda x.state(x) \wedge borders(x,texas)$, or *When was Ada Lovelace born?* to the gapped relation: `birth-year (Ada Lovelace, ?x)`. These meaning representations are then used to query databases of facts.

We'll also briefly discuss two other QA paradigms. We'll see how to query a language model directly to answer a question, relying on the fact that huge pretrained language models have already encoded a lot of factoids. And we'll sketch classic pre-neural hybrid question-answering algorithms that combine information from IR-based and knowledge-based sources.

We'll explore the possibilities and limitations of all these approaches, along the way also introducing two technologies that are key for question answering but also

[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*.

relevant throughout NLP: **information retrieval** (a key component of IR-based QA) and **entity linking** (similarly key for knowledge-based QA). We'll start in the next section by introducing the task of information retrieval.

The focus of this chapter is factoid question answering, but there are many other QA tasks the interested reader could pursue, including **long-form question answering** (answering questions like "why" questions that require generating long answers), **community question answering**, (using datasets of community-created question-answer pairs like Quora or Stack Overflow), or even answering questions on human exams like the New York Regents Science Exam (Clark et al., 2019) as an **NLP/AI benchmark** to measure progress in the field.

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
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. 14.1.

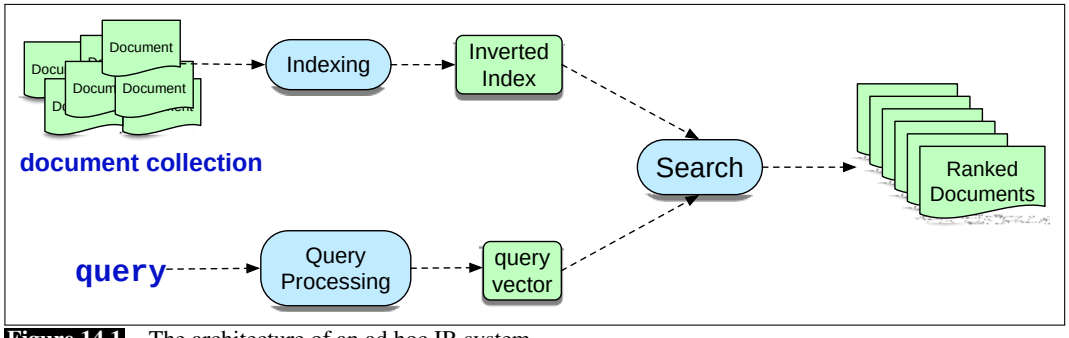


Figure 14.1 The architecture of an ad hoc IR system.

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

BM25

introduced in Chapter 6, and a slightly more powerful variant called **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. Because we can't take the log of 0, we normally add 1 to the count:[3]

$$\text{tf}_{t,d} = \log_{10}(\text{count}(t,d)+1) \tag{14.3}$$

If we use log weighting, terms which occur 0 times in a document would have $\text{tf} = \log_{10}(1) = 0$, 10 times in a document $\text{tf} = \log_{10}(11) = 1.04$, 100 times $\text{tf} = \log_{10}(101) = 2.004$, 1000 times $\text{tf} = 3.00044$, 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

[3] Or we can use this alternative: $\text{tf}_{t,d} = \begin{cases} 1+\log_{10}\text{count}(t,d) & \text{if } \text{count}(t,d) > 0 \\ 0 & \text{otherwise} \end{cases}$

[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:

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

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:

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

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

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

In practice, it's common to approximate Eq. 14.8 by simplifying the query processing. Queries are usually very short, so each query word is likely to have a count of 1. And the cosine normalization for the query (the division by $|q|$) will be the same for all documents, so won't change the ranking between any two documents D_i and D_j So we generally use the following simple score for a document d given a query q:

$$\text{score}(q,d) = \sum_{t\in q} \frac{\text{tf-idf}(t,d)}{|d|} \tag{14.9}$$

Let's 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 values and the document vector length $|d|$ for the first two documents using Eq. 14.3, Eq. 14.4, and Eq. 14.5 (computations for documents 3 and 4 are left as an exercise for the reader).

Fig. 14.3 shows the scores of the 4 documents, reranked according to Eq. 14.9. The ranking follows intuitively from the vector space model. Document 1, which has both terms including two instances of *sweet*, is the highest ranked, above document 3 which has a larger length $|d|$ in the denominator, and also a smaller tf for *sweet*. Document 3 is missing one of the terms, and Document 4 is missing both.

	Document 1					Document 2				
word	**count**	**tf**	**df**	**idf**	**tf-idf**	**count**	**tf**	**df**	**idf**	**tf-idf**
love	1	0.301	2	0.301	0.091	0	0	2	0.301	0
sweet	2	0.477	3	0.125	0.060	1	0.301	3	0.125	0.038
sorrow	0	0	1	0.602	0	1	0.301	1	0.602	0.181
how	0	0	1	0.602	0	0	0	1	0.602	0
nurse	1	0.301	2	0.301	0.091	0	0	2	0.301	0
is	0	0	1	0.602	0	0	0	1	0.602	0
$\lvert d_1\rvert = \sqrt{.091^2 + .060^2 + .091^2} = .141$						$\lvert d_2\rvert = \sqrt{.038^2 + .181^2} = .185$				

Figure 14.2 Computation of tf-idf for nano-documents 1 and 2, using Eq. 14.3, Eq. 14.4, and Eq. 14.5.

Doc	$\lvert d\rvert$	**tf-idf**(*sweet*)	**tf-idf**(*love*)	**score**
1	.141	.060	.091	1.07
3	.274	.038	.091	0.471
2	.185	.038	0	0.205
4	.090	0	0	0

Figure 14.3 Ranking documents by Eq. 14.9.

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:

$$\sum_{t \in q} \overbrace{\log\left(\frac{N}{df_t}\right)}^{\text{IDF}} \overbrace{\frac{tf_{t,d}}{k\left(1-b+b\left(\frac{|d|}{|d_{\text{avg}}|}\right)\right)+tf_{t,d}}}^{\text{weighted tf}} \tag{14.10}$$

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

stop list

14.1.3 Inverted Index

In order to compute scores, we need to efficiently find documents that contain words in the query. (As we saw in Fig. 14.3, 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. (2017a) 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:

$$Precision = \frac{|R|}{|T|} \qquad Recall = \frac{|R|}{|U|} \tag{14.11}$$

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

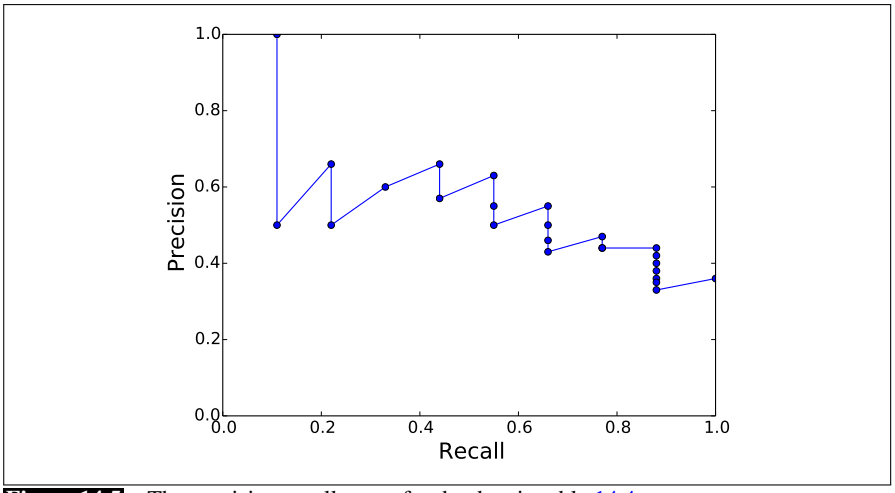


Figure 14.5 The precision recall curve for the data in table 14.4.

Let's turn to an example. Assume the table in Fig. 14.4 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. 14.5 for the data in table 14.4.

precision-recall curve

Fig. 14.5 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

$$\text{IntPrecision}(r) = \max_{i>=r} \text{Precision}(i) \tag{14.12}$$

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. 14.6 and Fig. 14.7 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 14.6 Interpolated data points from Fig. 14.4.

Given curves such as that in Fig. 14.7 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. 14.4). 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 documents at or above r, then the **average precision** (**AP**) for a single query is

$$\text{AP} = \frac{1}{|R_r|} \sum_{d \in R_r} \text{Precision}_r(d) \tag{14.13}$$

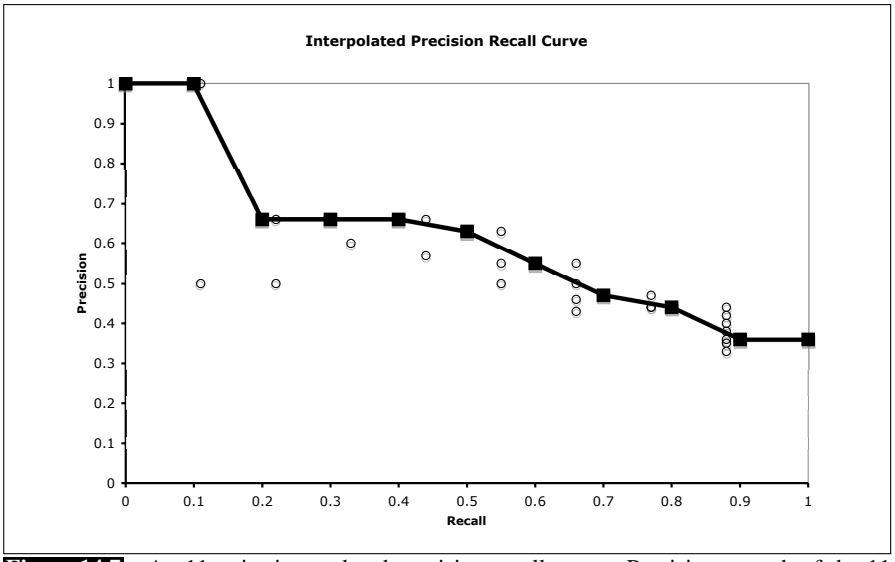


Figure 14.7 An 11 point interpolated precision-recall curve. Precision at each of the 11 standard recall levels is interpolated for each query from the maximum at any higher level of recall. The original measured precision recall points are also shown.

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:

$$\text{MAP} = \frac{1}{|Q|}\sum_{q\in Q}\text{AP}(q) \tag{14.14}$$

The MAP for the single query (hence = AP) in Fig. 14.4 is 0.6.

14.1.5 IR 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 to discuss the issue. As Lin et al. (2021) put it, the user might decide to search for a *tragic love story* but Shakespeare writes instead about *star-crossed lovers*. This is 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 proposed quite early with the LSI approach (Deerwester et al., 1990), but modern methods all make use of encoders like BERT. In what is sometimes called a **bi-encoder** we use two separate encoder models, one to encode the query and one to encode the document, and use the dot product between these two vectors as the score (Fig. 14.8. 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):

$$\begin{aligned} h_q &= \text{BERT}_Q(\text{q})\texttt{[CLS]} \\ h_d &= \text{BERT}_D(\text{d})\texttt{[CLS]} \\ \text{score}(d,q) &= h_q \cdot h_d \end{aligned} \tag{14.15}$$

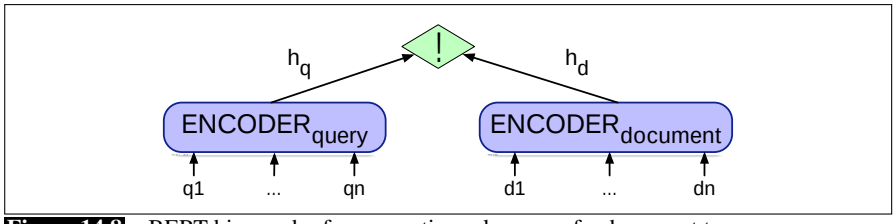


Figure 14.8 BERT bi-encoder for computing relevance of a document to a query.

More complex versions can use other ways to represent the encoded text, such as using average pooling over the BERT outputs of all tokens instead of using the CLS token, or can add extra weight matrices after the encoding or dot product steps (Liu et al. 2016a, Lee et al. 2019).

Using dense vectors for IR or the retriever component of question answerers is still an open area of research. Among the many areas of active research are how to do the fine-tuning of the encoder modules on the IR task (generally by fine-tuning on query-document combinations, with various clever ways to get negative examples), and how to deal with the fact that documents are often longer than encoders like BERT can process (generally by breaking up documents into passages).

Efficiency is also an issue. At the core of every IR engine is the need to rank every possible document for its similarity to the query. For sparse word-count vectors, the inverted index allows this very efficiently. For dense vector algorithms like those based on BERT or other Transformer encoders, finding the set of dense document vectors that have the highest dot product with a dense query vector is an example of nearest neighbor search. Modern systems therefore make use of approximate nearest neighbor vector search algorithms like **Faiss** (Johnson et al., 2017).

Faiss

14.2 IR-based Factoid Question Answering

IR-based QA

The goal of **IR-based QA** (sometimes called **open domain QA**) is to answer a user's question by finding short text segments from the web or some other large collection of documents. Figure 14.9 shows some sample factoid questions and their answers.

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.

retrieve and read

The dominant paradigm for IR-based QA is 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, usually using a search engines of the type we saw in the previous section. In the second stage, a neural **reading comprehension** algorithm passes over each passage and finds spans that are likely to answer the question.

reading comprehension

Some question answering systems focus only on the second task, the **reading comprehension** task. Reading comprehension systems are given a factoid question q and a passage p that could contain the answer, and return an answer s (or perhaps declare that there is no answer in the passage, or in some setups make a choice from

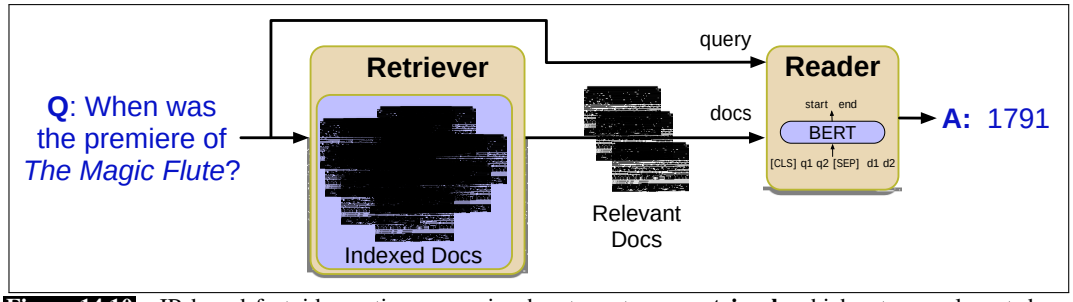


Figure 14.10 IR-based factoid question answering has two stages: **retrieval**, which returns relevant documents from the collection, and **reading**, in which a neural reading comprehension system extracts answer spans.

a set of possible answers). Of course this setup does not match the information need of users who have a question they need answered (after all, if a user knew which passage contained the answer, they could just read it themselves). Instead, this task was originally modeled on children's reading comprehension tests—pedagogical instruments in which a child is given a passage to read and must answer questions about it—as a way to evaluate natural language processing performance (Hirschman et al., 1999). Reading comprehension systems are still used that way, but have also evolved to function as the second stage of the modern **retrieve and read** model.

Other question answering systems address the entire retrieve and read task; they are given a factoid question and a large document collection (such as Wikipedia or a crawl of the web) and return an answer, usually a span of text extracted from a document. This task is often called **open domain QA**.

In the next few sections we'll lay out the various pieces of IR-based QA, starting with some commonly used datasets.

14.2.1 IR-based QA: Datasets

Datasets for IR-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 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.

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.

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.

Natural Questions

The **Natural Questions** dataset (Kwiatkowski et al., 2019) 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.

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, systems 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.

14.2.2 IR-based QA: Reader (Answer Span Extraction)

The first stage of IR-based QA is a retriever, for example of the type we saw in Section 14.1. The second stage of IR-based question answering is the **reader**. The reader's job is to take a passage as input and produce the answer. In the **extractive QA** we discuss here, the answer is a span of text in the passage.[5] 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*.

extractive QA

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.

span

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.

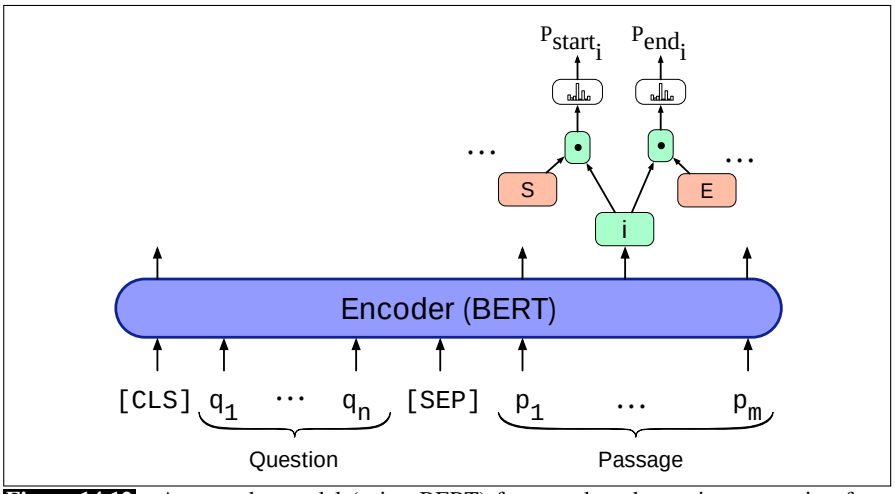


Figure 14.12 An encoder model (using BERT) for span-based question answering from reading-comprehension-based question answering tasks.

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:

$$P_{\text{start}_i} = \frac{\exp(S \cdot p'_i)}{\sum_j \exp(S \cdot p'_j)} \tag{14.16}$$

[5] Here we skip the more difficult task of **abstractive QA**, in which the system can write an answer which is not drawn exactly from the passage.

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

$$P_{\text{end}_i} = \frac{\exp(E \cdot p'_i)}{\sum_j \exp(E \cdot p'_j)} \tag{14.17}$$

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:

$$L = -\log P_{\text{start}_i} - \log P_{\text{end}_i} \tag{14.18}$$

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 Entity Linking

We've now seen the first major paradigm for question answering, IR-based QA. Before we turn to the second major paradigm for question answering, knowledge-based question answering, we introduce the important core technology of **entity linking**, since it is required for any knowledge-based QA algorithm.

entity linking

Entity linking is the task of associating a mention in text with the representation of some real-world entity in an ontology (Ji and Grishman, 2011).

The most common ontology for factoid question-answering is Wikipedia, since Wikipedia is often the source of the text that answers the question. In this usage, each unique Wikipedia page acts as the unique id for a particular entity. This task of deciding which Wikipedia page corresponding to an individual is being referred to by a text mention has its own name: **wikification** (Mihalcea and Csomai, 2007).

wikification

Since the earliest systems (Mihalcea and Csomai 2007, Cucerzan 2007, Milne and Witten 2008), entity linking is done in (roughly) two stages: **mention detection** and **mention disambiguation**. We'll give two algorithms, one simple classic baseline that uses **anchor dictionaries** and information from the Wikipedia graph structure (Ferragina and Scaiella, 2011) and one modern neural algorithm (Li et al.,

2020). We'll focus here mainly on the application of entity linking to questions rather than other genres.

14.3.1 Linking based on Anchor Dictionaries and Web Graph

As a simple baseline we introduce the TAGME linker (Ferragina and Scaiella, 2011) for Wikipedia, which itself draws on earlier algorithms (Mihalcea and Csomai 2007, Cucerzan 2007, Milne and Witten 2008). Wikification algorithms define the set of entities as the set of Wikipedia pages, so we'll refer to each Wikipedia page as a unique entity e. TAGME first creates a catalog of all entities (i.e. all Wikipedia pages, removing some disambiguation and other meta-pages) and indexes them in a standard IR engine like Lucene. For each page e, the algorithm computes an **in-link** count in(e): the total number of in-links from other Wikipedia pages that point to e. These counts can be derived from Wikipedia dumps.

anchor texts

Finally, the algorithm requires an **anchor dictionary**. An anchor dictionary lists for each Wikipedia page, its **anchor texts**: the hyperlinked spans of text on other pages that point to it. For example, the web page for Stanford University, `http://www.stanford.edu`, might be pointed to from another page using anchor texts like *Stanford* or *Stanford University*:

```
<a href="http://www.stanford.edu">Stanford University</a>
```

We compute a Wikipedia anchor dictionary by including, for each Wikipedia page e, e's title as well as all the anchor texts from all Wikipedia pages that point to e. For each anchor string a we'll also compute its total frequency freq(a) in Wikipedia (including non-anchor uses), the number of times a occurs as a link (which we'll call $link(a)$), and its link probability linkprob$(a) = \text{link}(a)/\text{freq}(a)$. Some cleanup of the final anchor dictionary is required, for example removing anchor strings composed only of numbers or single characters, that are very rare, or that are very unlikely to be useful entities because they have a very low linkprob.

Mention Detection Given a question (or other text we are trying to link), TAGME detects mentions by querying the anchor dictionary for each token sequence up to 6 words. This large set of sequences is pruned with some simple heuristics (for example pruning substrings if they have small linkprobs). The question:

When was Ada Lovelace born?

might give rise to the anchor *Ada Lovelace* and possibly *Ada*, but substrings spans like *Lovelace* might be pruned as having too low a linkprob, and but spans like *born* have such a low linkprob that they would not be in the anchor dictionary at all.

Mention Disambiguation If a mention span is unambiguous (points to only one entity/Wikipedia page), we are done with entity linking! However, many spans are ambiguous, matching anchors for multiple Wikipedia entities/pages. The TAGME algorithm uses two factors for disambiguating ambiguous spans, which have been referred to as *prior probability* and *relatedness/coherence*. The first factor is $p(e|a)$, the probability with which the span refers to a particular entity. For each page $e \in \mathcal{E}(a)$, the probability $p(e|a)$ that anchor a points to e, is the ratio of the number of links into e with anchor text a to the total number of occurrences of a as an anchor:

$$\text{prior}(a \rightarrow e) \;=\; p(e|a) = \frac{\text{count}(a \rightarrow e)}{\text{link}(a)} \tag{14.19}$$

Let's see how that factor works in linking entities in the following question:

What Chinese Dynasty came before the Yuan?

The most common association for the span *Yuan* in the anchor dictionary is the name of the Chinese currency, i.e., the probability $p(\texttt{Yuan\_currency}|\ yuan)$ is very high. Rarer Wikipedia associations for *Yuan* include the common Chinese last name, a language spoken in Thailand, and the correct entity in this case, the name of the Chinese dynasty. So if we chose based only on $p(e|a)$, we would make the wrong disambiguation and miss the correct link, `Yuan_dynasty`.

To help in just this sort of case, TAGME uses a second factor, the **relatedness** of this entity to other entities in the input question. In our example, the fact that the question also contains the span *Chinese Dynasty*, which has a high probability link to the page `Dynasties_in_Chinese_history`, ought to help match `Yuan_dynasty`.

Let's see how this works. Given a question q, for each candidate anchors span a detected in q, we assign a relatedness score to each possible entity $e \in \mathcal{E}(a)$ of a. The relatedness score of the link $a \rightarrow e$ is the weighted average relatedness between e and all other entities in q. Two entities are considered related to the extent their Wikipedia pages share many in-links. More formally, the relatedness between two entities A and B is computed as

$$\text{rel}(A,B) = \frac{\log(\max(|\text{in}(A)|,|\text{in}(B)|)) - \log(|\text{in}(A) \cap \text{in}(B)|)}{\log(|W|) - \log(\min(|\text{in}(A)|,|\text{in}(B)|))} \tag{14.20}$$

where $\text{in}(x)$ is the set of Wikipedia pages pointing to x and W is the set of all Wikipedia pages in the collection.

The vote given by anchor b to the candidate annotation $a \rightarrow X$ is the average, over all the possible entities of b, of their relatedness to X, weighted by their prior probability:

$$\text{vote}(b,X) = \frac{1}{|\mathcal{E}(b)|} \sum_{Y \in \mathcal{E}(b)} \text{rel}(X,Y)p(Y|b) \tag{14.21}$$

The total relatedness score for $a \rightarrow X$ is the sum of the votes of all the other anchors detected in q:

$$\text{relatedness}(a \rightarrow X) = \sum_{b \in \mathcal{X}_q \setminus a} \text{vote}(b,X) \tag{14.22}$$

To score $a \rightarrow X$, we combine relatedness and prior by choosing the entity X that has the highest $\text{relatedness}(a \rightarrow X)$, finding other entities within a small ϵ of this value, and from this set, choosing the entity with the highest prior $P(X|a)$. The result of this step is a single entity assigned to each span in q.

The TAGME algorithm has one further step of pruning spurious anchor/entity pairs, assigning a score averaging link probability with the coherence.

$$\text{coherence}(a \rightarrow X) = \frac{1}{|S|-1} \sum_{B \in \mathcal{S} \setminus X} \text{rel}(B,X)$$

$$\text{score}(a \rightarrow X) = \frac{\text{coherence}(a \rightarrow X) + \text{linkprob}(a)}{2} \tag{14.23}$$

Finally, pairs are pruned if $\text{score}(a \rightarrow X) < \lambda$, where the threshold λ is set on a held-out set.

14.3.2 Neural Graph-based linking

More recent entity linking models are based on **biencoders**, encoding a candidate mention span, encoding an entity, and computing the dot product between the encodings. This allows embeddings for all the entities in the knowledge base to be precomputed and cached (Wu et al., 2020). Let's sketch the ELQ linking algorithm of Li et al. (2020), which is given a question q and a set of candidate entities from Wikipedia with associated Wikipedia text, and outputs tuples (e, m_s, m_e) of entity id, mention start, and mention end. As Fig. 14.13 shows, it does this by encoding each Wikipedia entity using text from Wikipedia, encoding each mention span using text from the question, and computing their similarity, as we describe below.

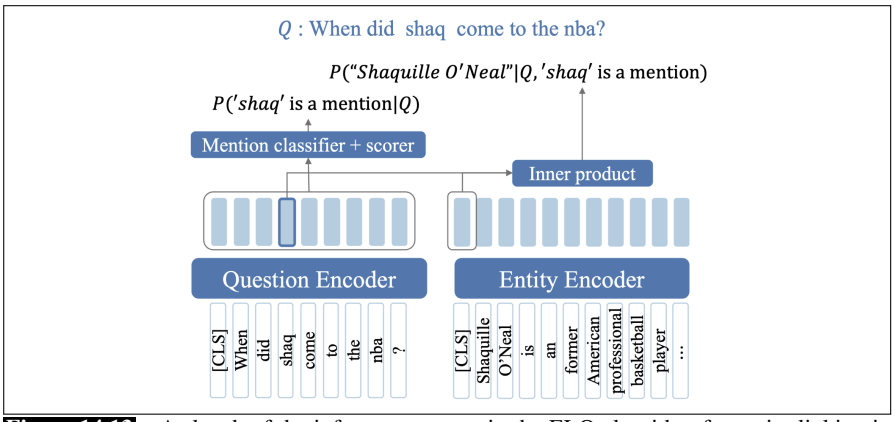


Figure 14.13 A sketch of the inference process in the ELQ algorithm for entity linking in questions (Li et al., 2020). Each candidate question mention span and candidate entity are separately encoded, and then scored by the entity/span dot product.

Entity Mention Detection To get an h-dimensional embedding for each question token, the algorithm runs the question through BERT in the normal way:

$$[\mathbf{q}_1 \cdots \mathbf{q}_n] = \text{BERT}([\text{CLS}]q_1 \cdots q_n[\text{SEP}]) \tag{14.24}$$

It then computes the likelihood of each span $[i, j]$ in q being an entity mention, in a way similar to the span-based algorithm we saw for the reader above. First we compute the score for i/j being the start/end of a mention:

$$s_{\text{start}}(i) = \mathbf{w}_{\text{start}} \cdot \mathbf{q}_i, \quad s_{\text{end}}(j) = \mathbf{w}_{\text{end}} \cdot \mathbf{q}_j, \tag{14.25}$$

where $\mathbf{w}_{\text{start}}$ and $\mathbf{w}_{\text{end}}$ are vectors learned during training. Next, another trainable embedding, $\mathbf{w}_{\text{mention}}$ is used to compute a score for each token being part of a mention:

$$s_{\text{mention}}(t) = \mathbf{w}_{\text{mention}} \cdot \mathbf{q}_t \tag{14.26}$$

Mention probabilities are then computed by combining these three scores:

$$p([i, j]) = \sigma\left(s_{\text{start}}(i) + s_{\text{end}}(j) + \sum_{t=i}^{j} s_{\text{mention}}(t)\right) \tag{14.27}$$

Entity Linking To link mentions to entities, we next compute embeddings for each entity in the set $\mathcal{E} = e_1, \cdots, e_i, \cdots, e_w$ of all Wikipedia entities. For each entity e_i we'll get text from the entity's Wikipedia page, the title $t(e_i)$ and the first 128 tokens of the Wikipedia page which we'll call the description $d(e_i)$. This is again run through BERT, taking the output of the CLS token $\text{BERT}_{[\text{CLS}]}$ as the entity representation:

$$\mathbf{x}_e = \text{BERT}_{[\text{CLS}]}([\texttt{CLS}]t(e_i)[\texttt{ENT}]d(e_i)[\texttt{SEP}]) \tag{14.28}$$

Mention spans can be linked to entities by computing, for each entity e and span $[i, j]$, the dot product similarity between the span encoding (the average of the token embeddings) and the entity encoding.

$$\begin{aligned} \mathbf{y}_{i,j} &= \frac{1}{(j-i+1)} \sum_{t-i}^{j} \mathbf{q}_t \\ s(e,[i,j]) &= \mathbf{x}_e \cdot \mathbf{y}_{i,j} \end{aligned} \tag{14.29}$$

Finally, we take a softmax to get a distribution over entities for each span:

$$p(e|[i,j]) = \frac{\exp(s(e,[i,j]))}{\sum_{e' \in \mathcal{E}} \exp(s(e',[i,j]))} \tag{14.30}$$

Training The ELQ mention detection and entity linking algorithm is fully supervised. This means, unlike the anchor dictionary algorithms from Section 14.3.1, it requires datasets with entity boundaries marked and linked. Two such labeled datasets are WebQuestionsSP (Yih et al., 2016), an extension of the WebQuestions (Berant et al., 2013) dataset derived from Google search questions, and GraphQuestions (Su et al., 2016). Both have had entity spans in the questions marked and linked (Sorokin and Gurevych 2018, Li et al. 2020) resulting in entity-labeled versions $\text{WebQSP}_{\text{EL}}$ and $\text{GraphQ}_{\text{EL}}$ (Li et al., 2020).

Given a training set, the ELQ mention detection and entity linking phases are trained jointly, optimizing the sum of their losses. The mention detection loss is a binary cross-entropy loss

$$\mathcal{L}_{\text{MD}} = -\frac{1}{N} \sum_{1 \le i \le j \le min(i+L-1,n)} \left(y_{[i,j]} \log p([i,j]) + (1 - y_{[i,j]}) \log(1 - p([i,j])) \right) \tag{14.31}$$

with $y_{[i,j]} = 1$ if $[i, j]$ is a gold mention span, else 0. The entity linking loss is:

$$\mathcal{L}_{\text{ED}} = -log p(e_g|[i,j]) \tag{14.32}$$

where e_g is the gold entity for mention $[i, j]$.

See the end of the chapter for more discussion of other applications of entity linking outside of question answering.

14.4 Knowledge-based Question Answering

While an enormous amount of information is encoded in the vast amount of text on the web, information obviously also exists in more structured forms. We use the term **knowledge-based question answering** for the idea of answering a natural

language question by mapping it to a query over a structured database. Like the text-based paradigm for question answering, this approach dates back to the earliest days of natural language processing, with systems like BASEBALL (Green et al., 1961) that answered questions from a structured database of baseball games and stats.

Two common paradigms are used for knowledge-based QA. The first, **graph-based QA**, models the knowledge base as a graph, often with entities as nodes and relations or propositions as edges between nodes. The second, **QA by semantic parsing**, using the semantic parsing methods we saw in Chapter 20. Both of these methods require some sort of entity linking that we described in the prior section.

14.4.1 Knowledge-Based QA from RDF triple stores

Let's introduce the components of a simple knowledge-based QA system after entity linking has been performed. We'll focus on the very simplest case of graph-based QA, in which the dataset is a set of factoids in the form of **RDF triples**, and the task is to answer questions about one of the missing arguments. Recall from Chapter 21 that an RDF triple is a 3-tuple, a predicate with two arguments, expressing some simple relation or proposition. Popular such ontologies are often derived from Wikipedia; DBpedia (Bizer et al., 2009) has over 2 billion RDF triples, or Freebase (Bollacker et al., 2008), now part of Wikidata (Vrandečić and Krötzsch, 2014). Consider an RDF triple like the following:

subject	**predicate**	**object**
Ada Lovelace	birth-year	1815

This triple can be used to answer text questions like "When was Ada Lovelace born?" or "Who was born in 1815?".

A number of such question datasets exist. SimpleQuestions (Bordes et al., 2015) contains 100K questions written by annotators based on triples from Freebase. For example, the question *"What American cartoonist is the creator of Andy Lippincott?"*. was written based on the triple `(andy lippincott, character created by, garry trudeau)`. FreebaseQA (Jiang et al., 2019), aligns the trivia questions from TriviaQA (Joshi et al., 2017) and other sources with triples in Freebase, aligning for example the trivia question *"Which 18th century author wrote Clarissa (or The Character History of a Young Lady), said to be the longest novel in the English language?"* with the triple `(Clarissa, book.written-work.author, Samuel Richardson)`. Another such family of datasets starts from WEBQUESTIONS (Berant et al., 2013), which contains 5,810 questions asked by web users, each beginning with a wh-word, containing exactly one entity, and paired with handwritten answers drawn from the Freebase page of the question's entity. WEBQUESTIONSSP (Yih et al., 2016) augments WEBQUESTIONS with human-created semantic parses (SPARQL queries) for those questions answerable using Freebase. COMPLEXWEBQUESTIONS augments the dataset with compositional and other kinds of complex questions, resulting in 34,689 questions, along with answers, web snippets, and SPARQL queries (Talmor and Berant, 2018).

Let's assume we've already done the stage of **entity linking** introduced in the prior section. Thus we've mapped already from a textual mention like *Ada Lovelace* to the canonical entity ID in the knowledge base. For simple triple relation question answering, the next step is to determine which relation is being asked about, mapping from a string like "When was ... born" to canonical relations in the knowledge base like *birth-year*. We might sketch the combined task as:

"When was Ada Lovelace born?" → `birth-year (Ada Lovelace, ?x)`

"What is the capital of England?" → `capital-city(?x, England)`

The next step is relation detection and linking. For simple questions, where we assume the question has only a single relation, relation detection and linking can be done in a way resembling the neural entity linking models: computing similarity (generally by dot product) between the encoding of the question text and an encoding for each possible relation. For example, in the algorithm of (Lukovnikov et al., 2019), the CLS output of a BERT model is used to represent the question span for the purposes of relation detection, and a separate vector is trained for each relation r_i. The probability of a particular relation r_i is then computed by softmax over the dot products:

$$\begin{aligned} \mathbf{m}_r &= \text{BERT}_{\text{CLS}}([\text{CLS}]q_1 \cdots q_n[\text{SEP}]) \\ s(\mathbf{m_r}, r_i) &= \mathbf{m_r} \cdot \mathbf{w_{r_i}} \\ p(r_i|q_1, \cdots, q_n) &= \frac{\exp(s(\mathbf{m_r}, r_i))}{\sum_{k=1} N_R \exp(s(\mathbf{m_r}, r_k))} \end{aligned} \tag{14.33}$$

Ranking of answers Most algorithms have a final stage which takes the top j entities and the top k relations returned by the entity and relation inference steps, searches the knowledge base for triples containing those entities and relations, and then ranks those triples. This ranking can be heuristic, for example scoring each entity/relation pairs based on the string similarity between the mention span and the entities text aliases, or favoring entities that have a high in-degree (are linked to by many relations). Or the ranking can be done by training a classifier to take the concatenated entity/relation encodings and predict a probability.

14.4.2 QA by Semantic Parsing

The second kind of knowledge-based QA uses a **semantic parser** to map the question to a structured program to produce an answer. These logical forms can take the form of some version of predicate calculus, a query language like SQL or SPARQL, or some other executable program like the examples in Fig. 14.14.

The logical form of the question is thus either in the form of a query or can easily be converted into one (predicate calculus can be converted to SQL, for example). The database can be a full relational database, or some other structured knowledge store.

As we saw in Chapter 20, semantic parsing algorithms can be supervised fully with questions paired with a hand-built logical form, or can be weakly supervised by questions paired with an answer (the **denotation**), in which the logical form is modeled only as a latent variable.

For the fully supervised case, we can get a set of questions paired with their correct logical form from datasets like the GEOQUERY dataset of questions about US geography (Zelle and Mooney, 1996), the DROP dataset of complex questions (on history and football games) that require reasoning (Dua et al. 2019), or the ATIS dataset of flight queries, all of which have versions with SQL or other logical forms (Iyer et al. 2017, Wolfson et al. 2020, Oren et al. 2020).

The task is then to take those pairs of training tuples and produce a system that maps from new questions to their logical forms. A common baseline algorithm is a simple sequence-to-sequence model, for example using BERT to represent question tokens, passing them to an encoder-decoder (Chapter 13), as sketched in Fig. 14.15. Any other of the semantic parsing algorithms described in Chapter 20 would also be appropriate.

Question	Logical form
What states border Texas?	$\lambda x.\text{state}(x) \wedge \text{borders}(x, \text{texas})$
What is the largest state?	$\text{argmax}(\lambda x.\text{state}(x), \lambda x.\text{size}(x))$
I'd like to book a flight from San Diego to Toronto	SELECT DISTINCT f1.flight_id FROM flight f1, airport_service a1, city c1, airport_service a2, city c2 WHERE f1.from_airport=a1.airport_code AND a1.city_code=c1.city_code AND c1.city_name= 'san diego' AND f1.to_airport=a2.airport_code AND a2.city_code=c2.city_code AND c2.city_name= 'toronto'
How many people survived the sinking of the Titanic?	`(count (!fb:event.disaster.survivors fb:en.sinking_of_the_titanic))`
How many yards longer was Johnson's longest touchdown compared to his shortest touchdown of the first quarter?	ARITHMETIC diff(SELECT num(ARGMAX(SELECT)) SELECT num(ARGMIN(FILTER(SELECT))))

Figure 14.14 Sample logical forms produced by a semantic parser for question answering, including two questions from the GeoQuery database of questions on U.S. Geography (Zelle and Mooney, 1996) with predicate calculus representations, one ATIS question with SQL (Iyer et al., 2017), a program over Freebase relations, and a program in QDMR, the Question Decomposition Meaning Representation (Wolfson et al., 2020).

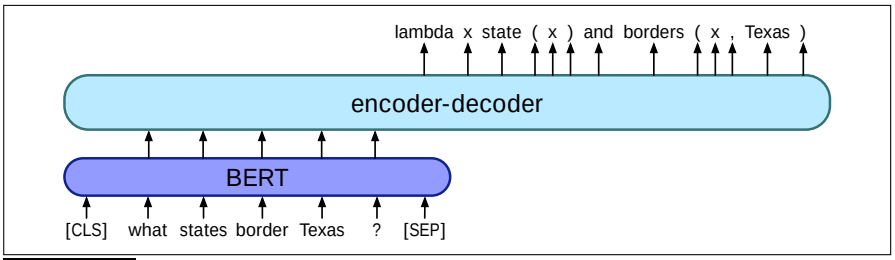


Figure 14.15 An encoder-decoder semantic parser for translating a question to logical form, with a BERT pre-encoder followed by an encoder-decoder (biLSTM or Transformer).

14.5 Using Language Models to do QA

An alternative approach to doing QA is to query a pretrained language model, forcing a model to answer a question solely from information stored in its parameters. For example Roberts et al. (2020) use the T5 language model, which is an encoder-decoder architecture pretrained to fill in masked spans of task. Fig. 14.16 shows the architecture; the deleted spans are marked by <M>, and the system is trained to have the decoder generating the missing spans (separated by <M>).

Roberts et al. (2020) then finetune the T5 system to the question answering task, by giving it a question, and training it to output the answer text in the decoder. Using the largest 11-billion-parameter T5 model does competitively, although not quite as well as systems designed specifically for question answering.

Language modeling is not yet a complete solution for question answering; for example in addition to not working quite as well, they suffer from poor interpretability (unlike standard QA systems, for example, they currently can't give users more context by telling them what passage the answer came from). Nonetheless, the study of extracting answer from language models is an intriguing area for future question

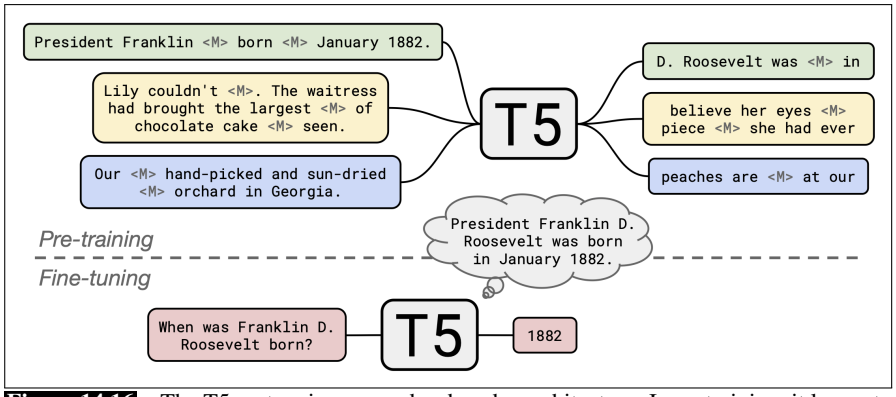


Figure 14.16 The T5 system is an encoder-decoder architecture. In pretraining, it learns to fill in masked spans of task (marked by <M>) by generating the missing spans (separated by <M>) in the decoder. It is then fine-tuned on QA datasets, given the question, without adding any additional context or passages. Figure from Roberts et al. (2020).

answer research.

14.6 Classic QA Models

While neural architectures are the state of the art for question answering, pre-neural architectures using hybrids of rules and feature-based classifiers can sometimes achieve higher performance. Here we summarize one influential classic system, the Watson DeepQA system from IBM that won the Jeopardy! challenge in 2011 (Fig. 14.17). Let's consider how it handles these Jeopardy! examples, each with a category followed by a question:

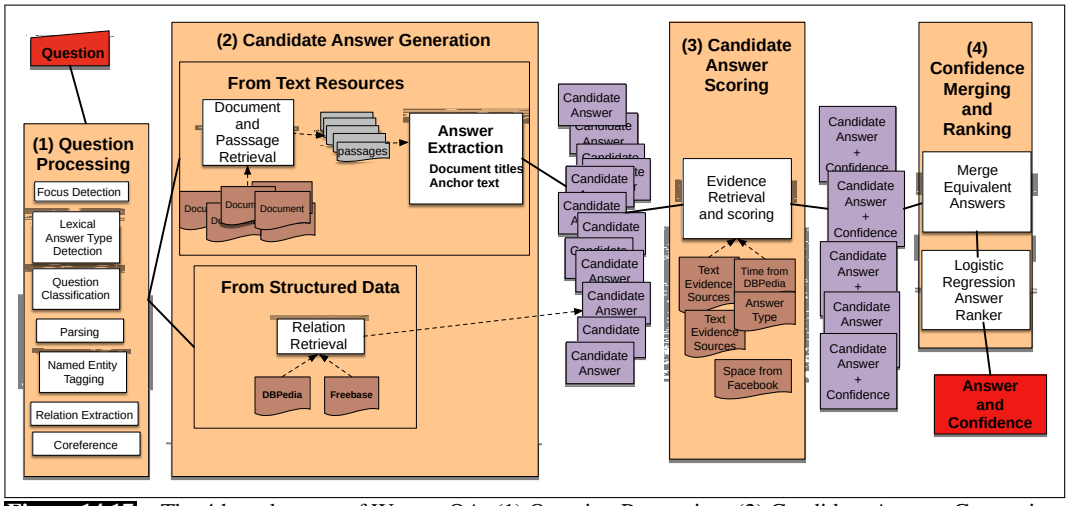


Figure 14.17 The 4 broad stages of Watson QA: (1) Question Processing, (2) Candidate Answer Generation, (3) Candidate Answer Scoring, and (4) Answer Merging and Confidence Scoring.

Poets and Poetry: **He** was a bank clerk in the Yukon before he published "Songs of a Sourdough" in 1907.

THEATRE: A new play based on **this Sir Arthur Conan Doyle canine classic** opened on the London stage in 2007.

Question Processing In this stage the questions are parsed, named entities are extracted (*Sir Arthur Conan Doyle* identified as a PERSON, *Yukon* as a GEOPOLITICAL ENTITY, *"Songs of a Sourdough"* as a COMPOSITION), coreference is run (*he* is linked with *clerk*).

focus

The question **focus**, shown in bold in both examples, is extracted. The focus is the string of words in the question that corefers with the answer. It is likely to be replaced by the answer in any answer string found and so can be used to align with a supporting passage. In DeepQA the focus is extracted by handwritten rules—made possible by the relatively stylized syntax of Jeopardy! questions—such as a rule extracting any noun phrase with determiner "this" as in the Conan Doyle example, and rules extracting pronouns like *she*, *he*, *hers*, *him*, as in the poet example.

lexical answer type

The **lexical answer type** (shown in blue above) is a word or words which tell us something about the semantic type of the answer. Because of the wide variety of questions in Jeopardy!, DeepQA chooses a wide variety of words to be answer types, rather than a small set of named entities. These lexical answer types are again extracted by rules: the default rule is to choose the syntactic headword of the focus. Other rules improve this default choice. For example additional lexical answer types can be words in the question that are coreferent with or have a particular syntactic relation with the focus, such as headwords of appositives or predicative nominatives of the focus. In some cases even the Jeopardy! category can act as a lexical answer type, if it refers to a type of entity that is compatible with the other lexical answer types. Thus in the first case above, *he*, *poet*, and *clerk* are all lexical answer types. In addition to using the rules directly as a classifier, they can instead be used as features in a logistic regression classifier that can return a probability as well as a lexical answer type. These answer types will be used in the later 'candidate answer scoring' phase as a source of evidence for each candidate. Relations like the following are also extracted:

authorof(focus,"Songs of a sourdough")
publish (e1, he, "Songs of a sourdough")
in (e2, e1, 1907)
temporallink(publish(...), 1907)

Finally the question is classified by type (definition question, multiple-choice, puzzle, fill-in-the-blank). This is generally done by writing pattern-matching regular expressions over words or parse trees.

Candidate Answer Generation Next we combine the processed question with external documents and other knowledge sources to suggest many candidate answers from both text documents and structured knowledge bases. We can query structured resources like DBpedia or IMDB with the relation and the known entity, just as we saw in Section 14.4. Thus if we have extracted the relation `authorof(focus,"Songs of a sourdough")`, we can query a triple store with `authorof(?x,"Songs of a sourdough")` to return an author.

To extract answers from text DeepQA uses simple versions of Retrieve and Read. For example for the IR stage, DeepQA generates a query from the question by eliminating stop words, and then upweighting any terms which occur in any relation with the focus. For example from this query:

MOVIE-"ING": Robert Redford and Paul Newman starred in this depression-era grifter flick. *(Answer: "The Sting")*

the following weighted query might be passed to a standard IR system:

(2.0 Robert Redford) (2.0 Paul Newman) star depression era grifter (1.5 flick)

DeepQA also makes use of the convenient fact that the vast majority of Jeopardy! answers are the title of a Wikipedia document. To find these titles, we can do a second text retrieval pass specifically on Wikipedia documents. Then instead of extracting passages from the retrieved Wikipedia document, we directly return the titles of the highly ranked retrieved documents as the possible answers.

Once we have a set of passages, we need to extract candidate answers. If the document happens to be a Wikipedia page, we can just take the title, but for other texts, like news documents, we need other approaches. Two common approaches are to extract all **anchor texts** in the document (anchor text is the text between <a> and used to point to a URL in an HTML page), or to extract all noun phrases in the passage that are Wikipedia document titles.

anchor texts

Candidate Answer Scoring Next DeepQA uses many sources of evidence to score each candidate. This includes a classifier that scores whether the candidate answer can be interpreted as a subclass or instance of the potential answer type. Consider the candidate "difficulty swallowing" and the lexical answer type "manifestation". DeepQA first matches each of these words with possible entities in ontologies like DBpedia and WordNet. Thus the candidate "difficulty swallowing" is matched with the DBpedia entity "Dysphagia", and then that instance is mapped to the WordNet type "Symptom". The answer type "manifestation" is mapped to the WordNet type "Condition". The system looks for a hyponymy, or synonymy link, in this case finding hyponymy between "Symptom" and "Condition".

Other scorers are based on using time and space relations extracted from DBpedia or other structured databases. For example, we can extract temporal properties of the entity (when was a person born, when died) and then compare to time expressions in the question. If a time expression in the question occurs chronologically before a person was born, that would be evidence against this person being the answer to the question.

Finally, we can use text retrieval to help retrieve evidence supporting a candidate answer. We can retrieve passages with terms matching the question, then replace the focus in the question with the candidate answer and measure the overlapping words or ordering of the passage with the modified question.

The output of this stage is a set of candidate answers, each with a vector of scoring features.

Answer Merging and Scoring DeepQA finally merges equivalent candidate answers. Thus if we had extracted two candidate answers *J.F.K.* and *John F. Kennedy*, this stage would merge the two into a single candidate, for example using the anchor dictionaries described above for entity linking, which will list many synonyms for Wikipedia titles (e.g., *JFK, John F. Kennedy, Senator John F. Kennedy, President Kennedy, Jack Kennedy*). We then merge the evidence for each variant, combining the scoring feature vectors for the merged candidates into a single vector.

Now we have a set of candidates, each with a feature vector. A classifier takes each feature vector and assigns a confidence value to this candidate answer. The classifier is trained on thousands of candidate answers, each labeled for whether it is correct or incorrect, together with their feature vectors, and learns to predict a probability of being a correct answer. Since, in training, there are far more incorrect answers than correct answers, we need to use one of the standard techniques for dealing with very imbalanced data. DeepQA uses *instance weighting*, assigning an

instance weight of .5 for each incorrect answer example in training. The candidate answers are then sorted by this confidence value, resulting in a single best answer.

DeepQA's fundamental intuition is thus to propose a very large number of candidate answers from both text-based and knowledge-based sources and then use a rich variety of evidence features for scoring these candidates. See the papers mentioned at the end of the chapter for more details.

14.7 Evaluation of Factoid Answers

mean reciprocal rank
MRR

Factoid 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

$$\text{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{14.34}$$

Reading comprehension systems on datasets like SQuAD are evaluated (first ignoring punctuation and articles like *a, an, the*) via two metrics (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.

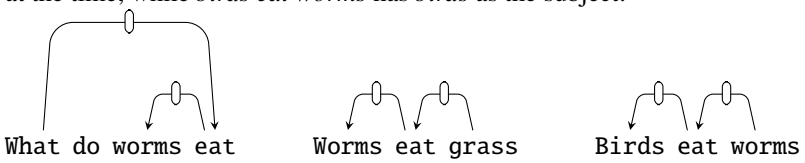
A number of test sets are available for question answering. Early systems used the TREC QA dataset: `https://trec.nist.gov/data/qa/t8_qadata.html`. More recent competitions uses the datasets described in Section 14.2.1. Other recent datasets include the AI2 Reasoning Challenge (ARC) (Clark et al., 2018) 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*.

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

LUNAR

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

(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 common. 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. (2017a), 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. Some recent systems include the IR component as part of a single end-to-end architecture (Lee et al., 2019).

Other question-answering tasks include Quiz Bowl, which has timing considerations since the question can be interrupted (Boyd-Graber et al., 2018). Question answering is also an important function of modern personal assistant dialog systems; see Chapter 15.

Exercises