Question Answering, Information Retrieval, and Retrieval-Augmented Generation

Motivation and History

- Question answering systems are designed to fill human information needs.
- Systems in the early 1960s were answering questions about baseball statistics and scientific facts.
- Because so much knowledge is encoded in text, QA systems were performing at human levels even before LLMs: IBM's Watson system won the TV game-show Jeopardy! in 2011
- question answering is closely related to the task behind search engines.

- Question answering systems often focus on a useful subset of information needs: factoid questions, questions of fact or reasoning that can be answered with simple facts expressed in short or medium-length texts, like the following:
- Where is the Louvre Museum located?
- Where does the energy in a nuclear explosion come from?
- How to get a script l in latex?

- Modern NLP systems answer these questions using large language models, in one of two ways.
- The first is to make use of the method of prompting a pretrained and instruction-tuned LLM, an LLM that has been finetuned on question/answer datasets with the question in the prompt.
- For example, we could prompt a causal language model with a string like
- Q: Where is the Louvre Museum located? A:
- It does conditional generation given this prefix, and take the response as the answer.
- The idea is that 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.

Problem with simple prompting based approach

- Simply prompting an LLM can be a useful approach to answer many factoid questions. But it is not yet a complete solution for question answering.
- The first and main problem is that large language models often give the wrong answer! Large language models hallucinate.
- For example, Dahl et al. (2024) found that when asked questions about the legal domain (like about particular legal cases), large language models hallucinated from 69% to 88% of the time!

- And it’s not always possible to tell when language models are hallucinating, partly because LLMs aren’t well-calibrated.
- In a calibrated system, the confidence of a system in the correctness of its answer is highly correlated with the probability of an answer being correct.
- But since language models are not well-calibrated, they often give a very wrong answer with complete certainty

- 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 about data like our personal email or medical records or a company may have internal documents that contain answers for customer service or internal use.
- Finally, static large language models also have problems with questions about rapidly changing information

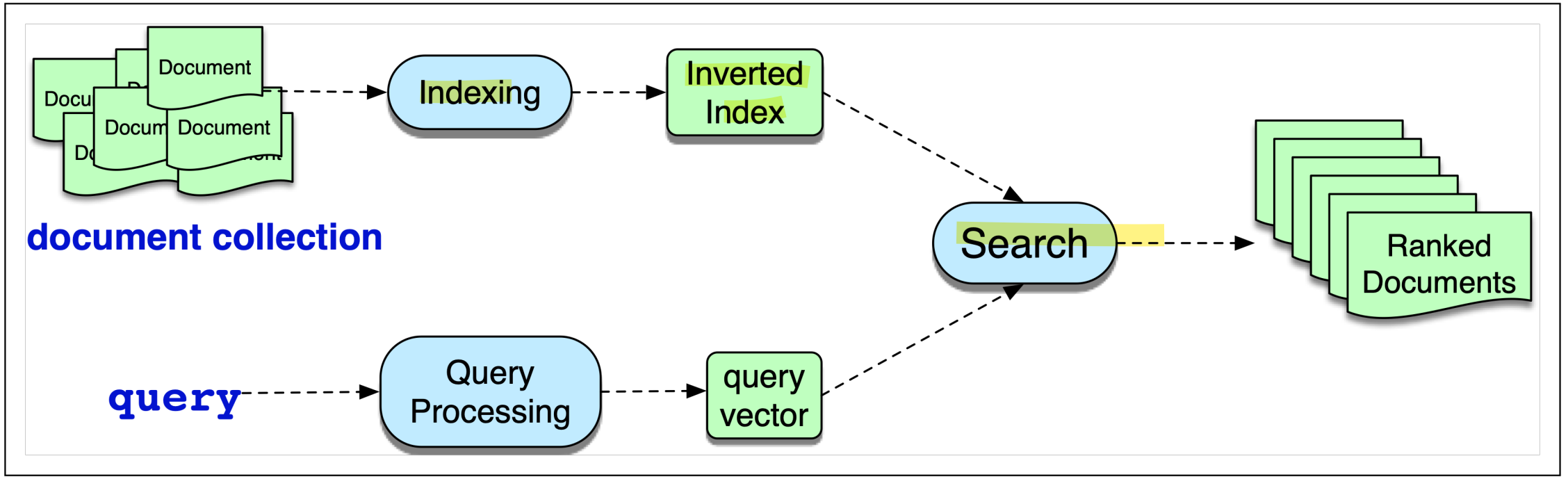
Retrieval-Augmented Generation or RAG

- For this reason the most common way to do question-answering with LLMs is retrieval-augmented generation or RAG
- In RAG we use information retrieval (IR) techniques to retrieve documents that are likely to have information that might help answer the question.
- Then we use a large language model to generate an answer given these documents.

Information Retrieval

- 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.
- 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
- 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.

Information Retrieval



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

Term weighting and document scoring

- 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 earlier, and a slightly more powerful variant called BM25.

Document Scoring using tf-idf

- We score document d by the cosine of its vector d with the query vector q:

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

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

Example

Query: sweet love

Doc 1: Sweet sweet nurse! Love?

Doc 2: Sweet sorrow

Doc 3: How sweet is love?

Doc 4: Nurse!

Query						
word	**cnt**	**tf**	**df**	**idf**	**tf-idf**	**n'lized** = tf-idf/$\|q\|$
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
$\|q\| = \sqrt{.125^2 + .301^2} = .326$						

	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
$\|d_1\| = \sqrt{.163^2 + .301^2 + .301^2} = .456$						$\|d_2\| = \sqrt{.125^2 + .602^2} = .615$				
Cosine: $\sum$ of column:					**0.747**	Cosine: $\sum$ of column:				**0.0779**

BM25

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

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

- Manning et al. (2008) suggest reasonable values are k = [1.2,2] and b = 0.75.

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

- An inverted index, given a query term, gives a list of documents that contain the term.
- It consists of two parts, a dictionary and the postings.
- The dictionary is a list of terms (designed to be efficiently accessed), each pointing to a postings list for the term.
- A postings list is the list of document IDs associated with each term, which can also contain information like the term frequency or even the exact positions of terms in the document.
- The dictionary can also store the document frequency for each term.

- For example, a simple inverted index for our 4 sample documents above, with each word containing its document frequency in {}, and a pointer to a postings list that contains document IDs and term counts in [], might look like the following:

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

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.

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.
- 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.
- U documents in the collection as a whole are relevant to this request.
- Precision and recall are then defined as:

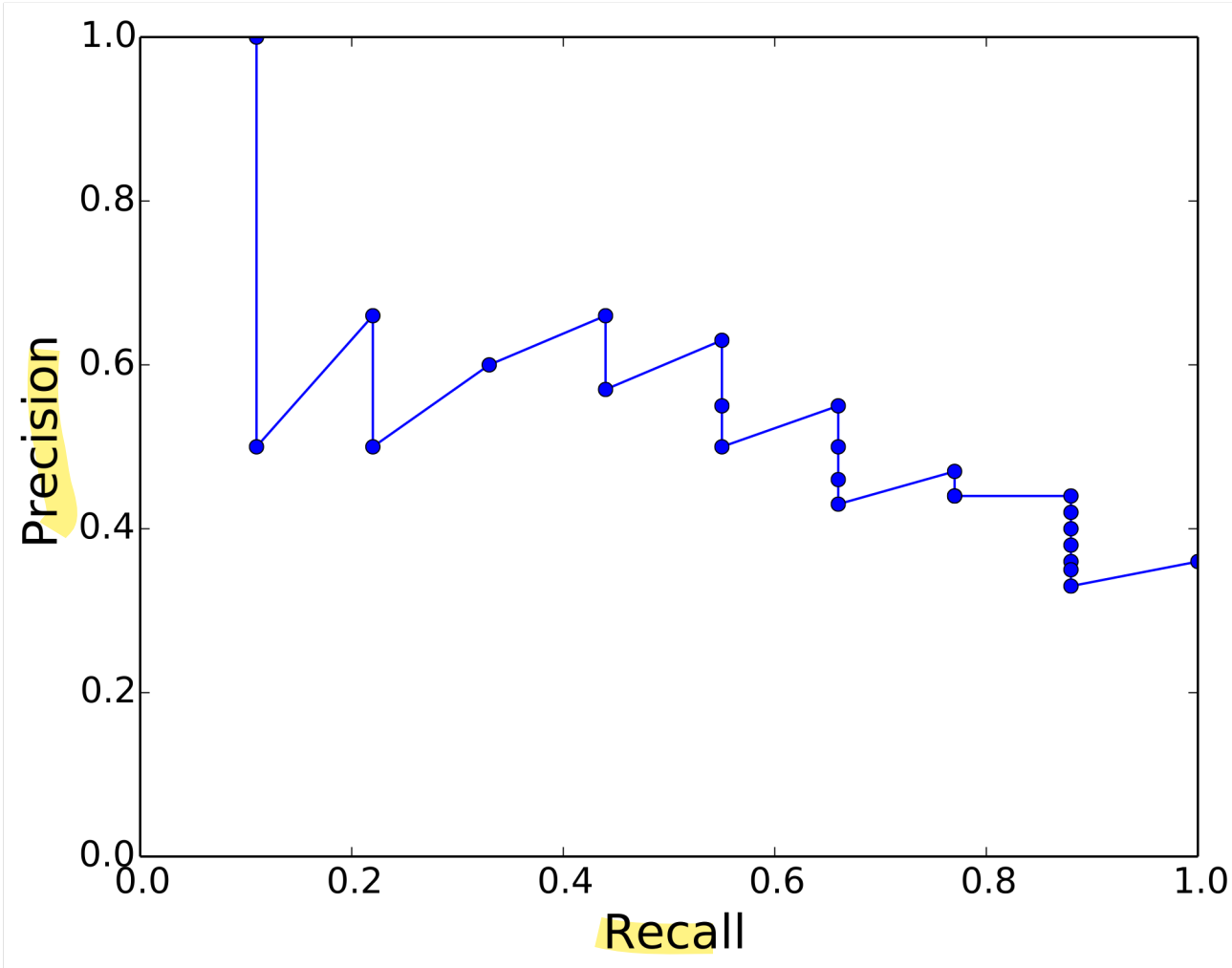
$$Precision = \frac{|R|}{|T|} \qquad Recall = \frac{|R|}{|U|}$$

- 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

- Find the table given a 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.
- Recalls are 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.

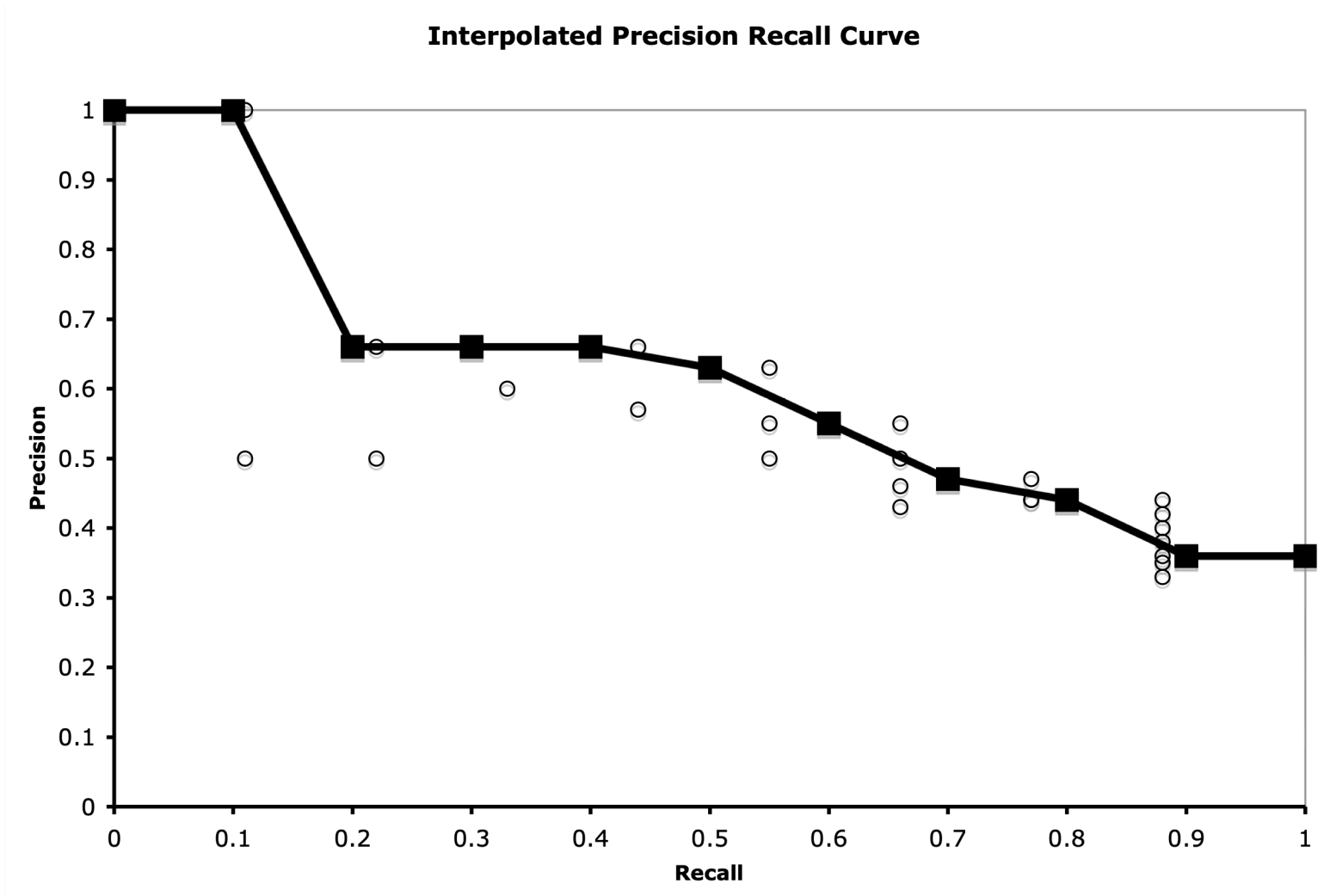
- Note that recall is non-decreasing; when a relevant document is encountered, recall increases, and when a non-relevant document is found it remains unchanged.
- Precision, on the other hand, jumps up and down, increasing when relevant documents are found, and decreasing otherwise.
- The most common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


1.0
0.8
0.6
0.4
0.2
0.0
Precision
0.0
0.2
0.4
0.6
0.8
1.0
Recall

- Last figure 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 precision at or above the one we're calculating. In other words,

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

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


Interpolated Precision Recall Curve
Precision
Recall
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

- Given such curves, we can compare two systems or approaches by comparing their curves.
- Clearly, curves that are higher in precision across all recall values are preferred.

Mean Average Precision (MAP)

- 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
- 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 Rr is the set of relevant documents at or above r, then the average precision (AP) for a single query is

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

- where Precisionr(d) is the precision measured at the rank at which document d was found.

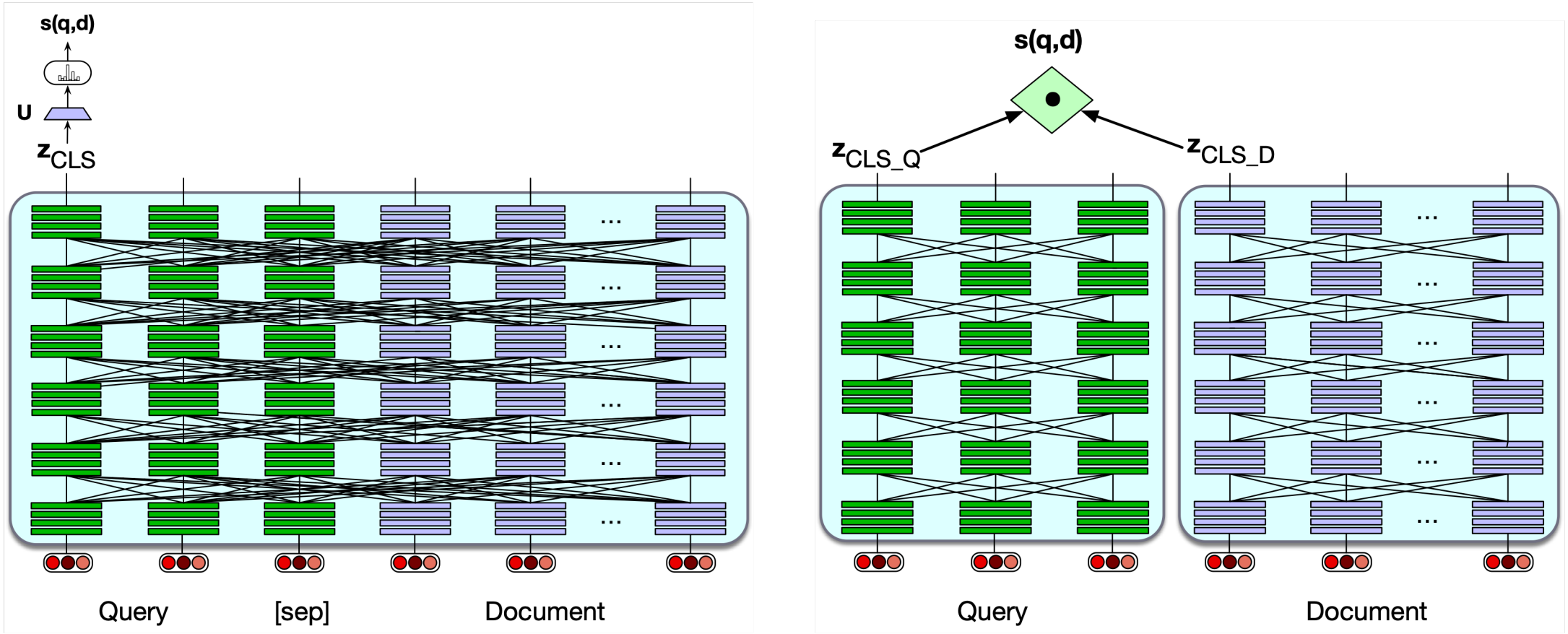
- For an ensemble of queries Q, we then average over these averages, to get our final MAP measure:

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

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.

- 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.
- We 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
- Then a linear layer can be put on top of the [CLS] token to predict a similarity score for the query/document tuple



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.

- 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 U can then be fine-tuned for the relevance task by gathering a tuning dataset of relevant and non-relevant passages.
- The problem with the full BERT architecture shown in Fig (a) 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

- 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

$$\mathbf{z}_q = \text{BERT}_Q(\text{q})[\text{CLS}]$$
$$\mathbf{z}_d = \text{BERT}_D(\text{d})[\text{CLS}]$$
$$\text{score}(q,d) = \mathbf{z}_q \cdot \mathbf{z}_d$$

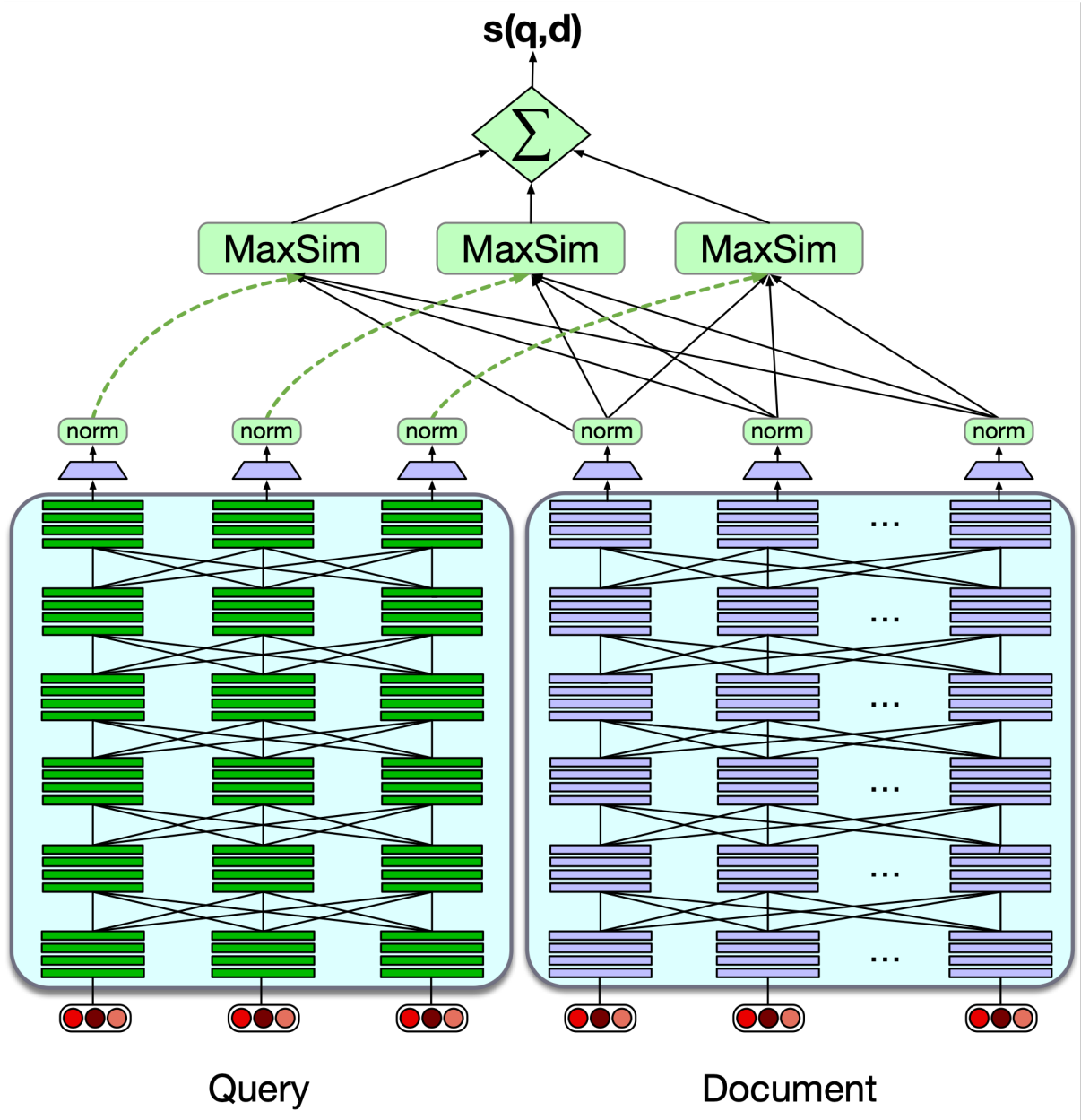
ColBERT approach

- There are numerous approaches that lie in between the full encoder and the bi-encoder, ColBERT is one such approach.
- 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 $[q_1,...,q_n]$, 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 $\mathbf{q = [q_1,...,q_N]}$.

- The passage d with tokens [d_1,...,d_m], is processed similarly, including a [CLS] and special [D] token.
- A linear layer is applied on top of d and 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 E_q (length N) and E_d (length m).

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

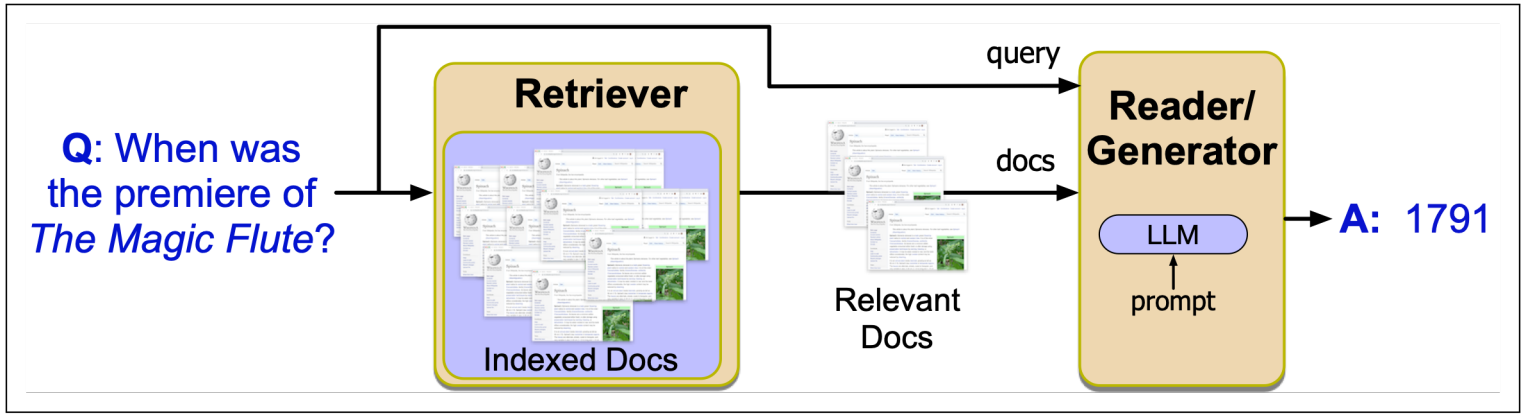

s(q,d)
∑
MaxSim
MaxSim
MaxSim
norm
norm
norm
norm
norm
norm
Query
Document

- the ColBERT architecture 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, optimizing model parameters using a cross-entropy loss.

Answering Questions with RAG

- The dominant paradigm for question answering is to answer a user's question by first finding supportive text segments from the web or another other large collection of documents, and then generating an answer based on the documents.
- The method of generating based on retrieved documents is called retrieval-augmented generation or RAG, and the two components are sometimes called the retriever and the reader

- In the first stage of the 2-stage retrieve and read model, we retrieve relevant passages from a text collection, for example using the dense retrievers of the previous section.
- In the second reader stage, we generate the answer via retrieval-augmented generation.



Retrieval-Augmented Generation

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

- 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, 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 R(q).
- For example, we could have a prompt like:

Schematic of a RAG Prompt

```
retrieved passage 1

retrieved passage 2

...

retrieved passage n

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

Or more formally,

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

- Some complex questions may require multi-hop architectures,
- In multi-hop architectures a query is used to retrieve documents, which are then appended to the original query for a second stage of retrieval.

Question Answering Datasets

- There are plenty of question answering datasets, used both for instruction tuning and for evaluation of the question answering abilities of language models.
- There are datasets like **Natural Questions** (Kwiatkowski et al., 2019), a set of anonymized English queries to the Google search engine and their answers.
- The answers are created by annotators based on Wikipedia information, and include a paragraph-length long answer and a short span answer.
- A similar natural question set is the **MS MARCO** (Microsoft Machine Reading Comprehension) collection of datasets, including 1 million real anonymized English questions from Microsoft Bing query logs together with a human generated answer and 9 million passages (Bajaj et al., 2016), that can be used both to test retrieval ranking and question answering.

- The **DuReader** dataset is a Chinese QA resource based on search engine queries and community QA (He et al., 2018).
- TyDi QA dataset contains 204K question-answer pairs from 11 typologically diverse languages, including Arabic, Bengali, Kiswahili, Russian, and Thai

Evaluating Question Answering

- Three techniques are commonly employed to evaluate question-answering systems
- For multiple choice questions, we try to find **Exact match**: The % of predicted answers that match the gold answer exactly.
- For questions with free text answers , we commonly evaluated with token F1 score to roughly measure the partial string overlap between the answer and the reference answer:
- Finally, in some situations QA systems give multiple ranked answers. In such cases we evaluated using mean reciprocal rank, or MRR (Voorhees, 1999).

- MRR is designed for systems that return a short ranked list of answers or passages for each test set question which we can compare against the (human-labeled) correct answer.
- First, each test set question is scored with the reciprocal of the rank of the first correct answer.
- For example if the system returned five answers to a question but the first three are wrong (so the highest-ranked correct answer is ranked fourth), the reciprocal rank for that question is 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.

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