Today's agenda:

1. classic retrieval (keyword-based)
2. neural retrieval
3. RAG
4. General examples of compound AI systems + challenges

Classic IR / BM25

- scope: large corpus of text documents (eg, Wikipedia - potentially split into passages)
- input: textual query (e.g., question)
- output: Top-K <u>RANKING</u> of relevant docs

* simple case: "single-term" queries

- build term document matrix
- each cell says how many times a term appears in a doc (perhaps w/ reweighting)
- if query has a single term, just return K documents w/ largest weights for a single term

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10
against	0	0	0	1	0	0	3	2	3	0
age	0	0	0	1	0	3	1	0	4	0
agent	0	0	0	0	0	0	0	0	0	0
ages	0	0	0	0	0	2	0	0	0	0
ago	0	0	0	2	0	0	0	0	3	0
agree	0	1	0	0	0	0	0	0	0	0
ahead	0	0	0	1	0	0	0	0	0	0
ain't	0	0	0	0	0	0	0	0	0	0
air	0	0	0	0	0	0	0	0	0	0
aka	0	0	0	1	0	0	0	0	0	0

* "against" has dif. weight for d4, d7

Stanford cs 224u slides. Omar Khattab

* what about "multi-term" queries?

$$\text{RelevanceScore}(\text{query}, \text{doc}) = \sum_{\text{term} \in \text{query}} \text{weight}_{\text{doc}, \text{term}}$$

* core questions in classical IR:

- How do we best tokenize queries / documents?
- How do we best WEIGHT term-document pair?

* Intuitions behind term-document weighting:

1) FREQUENCY
   - if term **t** occurs frequently in document **d** - doc is more likely to be relevant for queries w/ t

2) NORMALIZATION
   - if term **t** is rare overall, any document **d** including **t** is likely very relevant
   - if d is overall short and contains **t** → probably important

TF-IDF: Term-Document weighting

- $N = |\text{collection}|$
- $df(\text{term}) = |\{\text{doc} \in \text{collection} : \text{term} \in \text{doc}\}|$ which docs contain it

term freq

$$TF(\text{term}, \text{doc}) = \log(1 + \text{Freq}(\text{term}, \text{doc}))$$

↳ grows sublinearly w/ frequency

$$IDF(term) = \log\left(\frac{N}{df(term)}\right)$$

*if many docs contain term, IDF will be smaller

inverse doc frequency

↳ grows sublinearly w/ 1/DF

*accounts for normalization

$$TF.IDF(term, doc) = TF(term, doc) \times IDF(term)$$

$$TF.IDF(query, doc) = \sum_{term \in query} TFIDF(term, doc)$$

*BM25: inspired from TFIDF, but better

→ change: term frequency saturates to constant value and penalizes LONGER docs

$$IDF(term) = \log\left(1 + \frac{N - df(term) + 0.5}{df(term) + 0.5}\right)$$

$$TF(term, doc) = \frac{freq(term, doc) \cdot (k+1)}{freq(term, doc) + k \cdot \left(1 - b + b \cdot \frac{|doc|}{|avg\ doclen|}\right)}$$

*k, b are params

· Efficient Retrieval: Inverted Indexing

raw collection: docs → terms

term doc matrix: terms → docs

↳ could be very sparse!

* inverted index: sparse representation of term-doc matrix (maps each unique term T to "posting list"

↳ posting list contains list of docs where term appears, + frequency of term t in each doc

* in terms of calculating TF-IDF:

- when we see a query:

- for each term:

→ df(term) stored in inverted index

→ tf(term, doc) stored

→ can calculate TF-IDF

* until 2019, when BERT-based ranking came out - BM25 was easy to deploy, strong, baseline

- JUDGING IR SYSTEMS:

1) effective (good accuracy) and efficient

various measures

- latency per query
- throughput (queries/sec)
- space for index
- HW required
- scaling to larger indexes

→ success and MRR

- let rank {1,2,3...} be position in returned top-k of 1st RELEVANT DOC

success @k = { 1 if rank ≤ k ; 0 otherwise } → if user needs returned list to contain anything in top-k

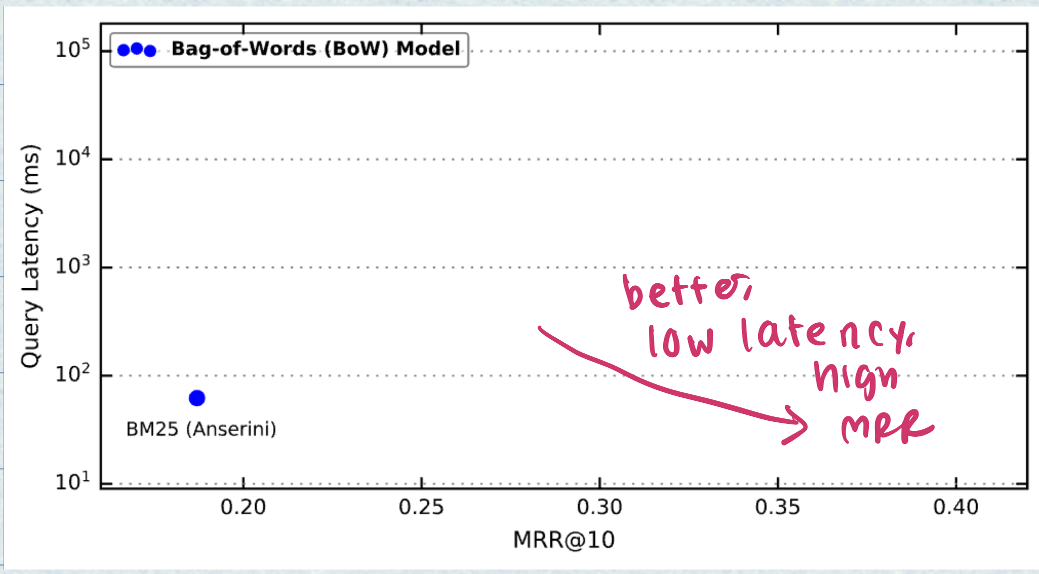
reciprocal rank at k = { 1/rank if rank ≤ k ; 0 otherwise } → cares abt positi

↳ MRR → mean of this across queries

→ look at k = 1, 5, 100, 1000...

→ for bm25, we get:

low latency, but, low MRR?



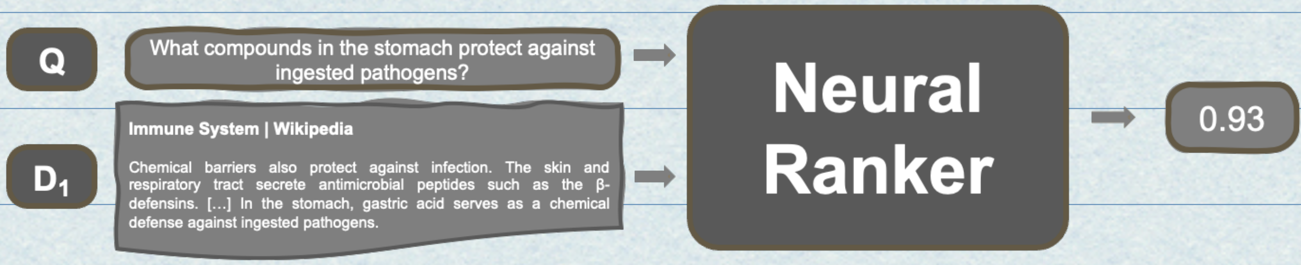
- how do we increase MRR (accuracy) for some cost of query latency?

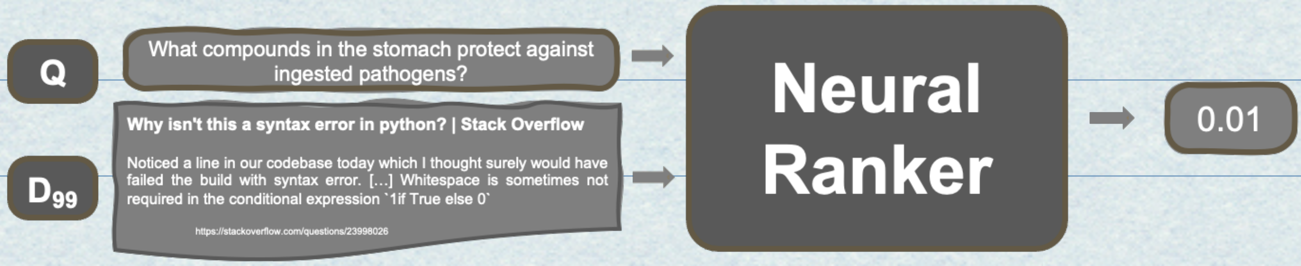
* MS MARCO: dataset widely used for retrieval (Bing queries, 9 mil. passages)

what would input and output of neural rankings be?

- if neural retriever were a blackbox:

Stanford cs 224u slides. Omar Khattab





Om

- input = query, document

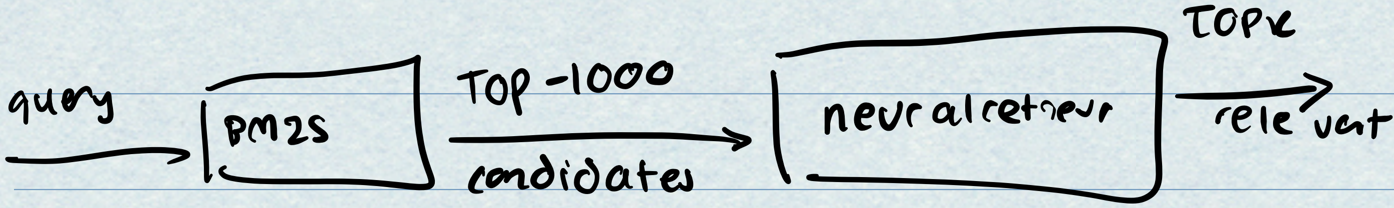
- output = score (relevance, higher = more relevant)

- TOPk docs = ranked ordering of docs according to score

* if we have 9 billion passages, and each query,doc pair inference takes 1ns → still 9s per query (unacceptable!)

IDEA:

- do reranking:

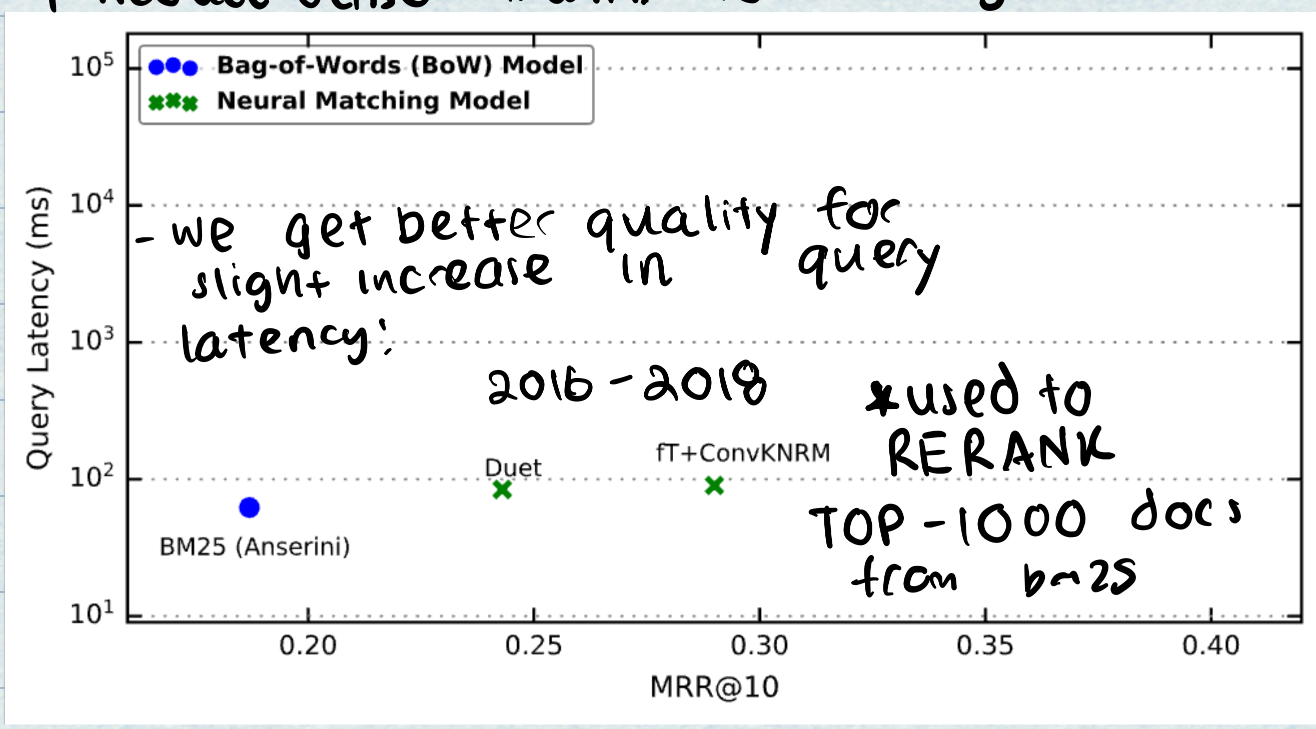


* potential problem: if BM25 misses

some docs, we won't get them in our ans ("artificial recall ceiling")

HOW DO WE CAPTURE QUERY-DOCUMENT RELEVANCE?

1. Tokenize query and document
2. Embed tokens into static vector repr
3. Build query-document interaction matrix
   - ↳ similarity scores btwn each pair of word embeddings across query/doc, such as cosine similarity
4. Reduce dense matrix to a single score



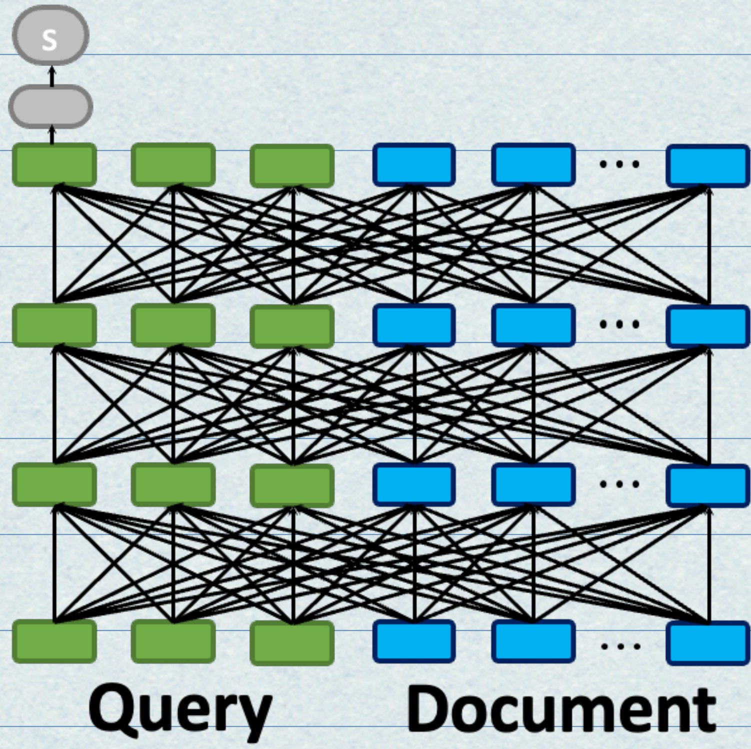
- we get better quality for query slight increase in latency!

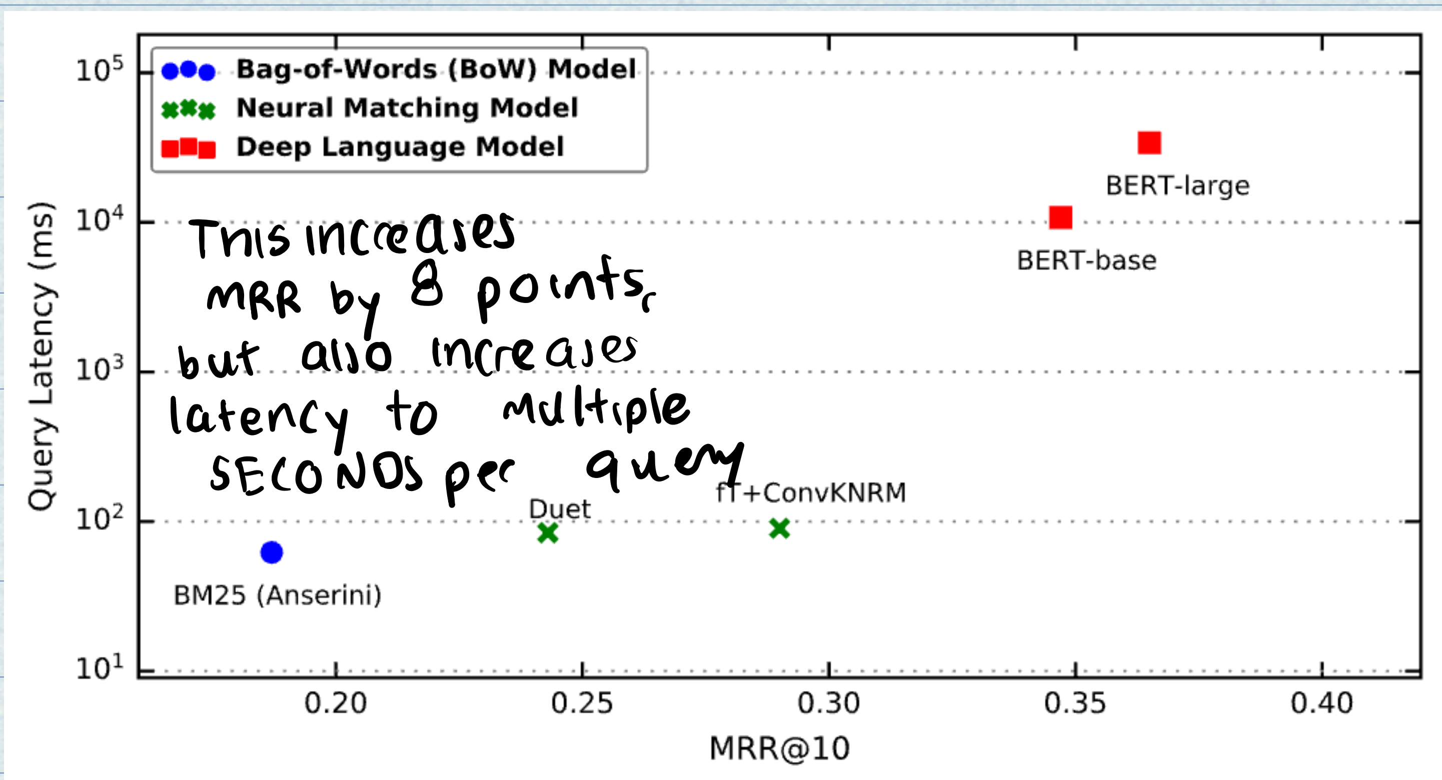
2016-2018

*used to RERANK TOP-1000 docs from bm25

POST 2018: All-to-All interaction w/ BERT

1. Feed BERT the query and document as one sentence w/ two segments
2. Run this through BERT layers
3. Extract final output embedding
   - use custom linear layer to reduce to single score





This increases MRR by 8 points, but also increases latency to multiple SECONDS per query

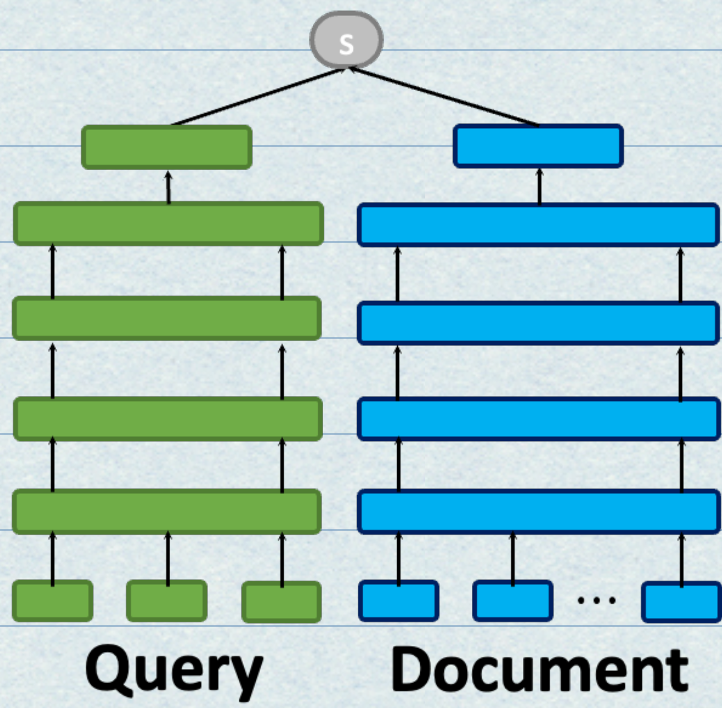
How do we INCREASE quality but keep latency low?

* Bert rankers slow as computation are redundant:
  - repr. query (1000 times for 1000 docs)
  - repr. doc (for EACH inference)
  - matching
* can we precompute document representations? and "cache" these for use across queries?

REPRESENTATION SIMILARITY:

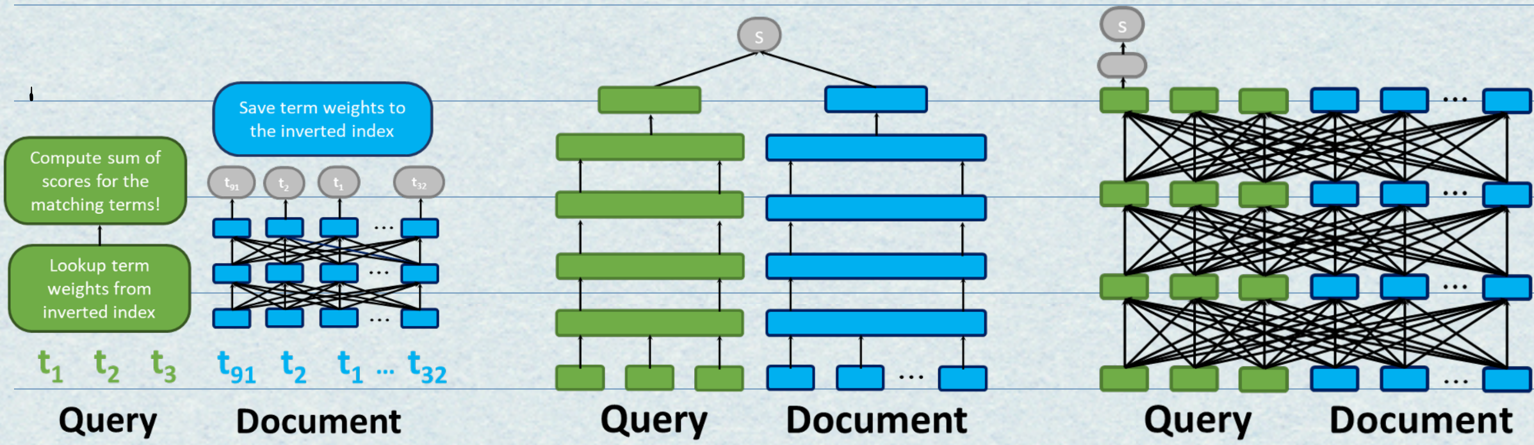
- tokenize query /doc
- independently encode query /doc
- use dot product to estimate similarity

- doc reprs can be precomputed!
- query reprs can be reused across dot products



- Downsides of JUST doing representation similarity:
  - single-vector repr could be coarse-grained
  - no FINE-GRAINED interaction
    ↳ e.g. btwn individual tokens in query to docs

SUMMARY SO FAR



(a) Learned Term Weights

✓ Independent Encoding
✗ Bag-of-Words Matching

(b) Representation Similarity

✓ Independent, Dense Encoding
✗ Coarse-Grained Representation

(c) Query–Document Interaction

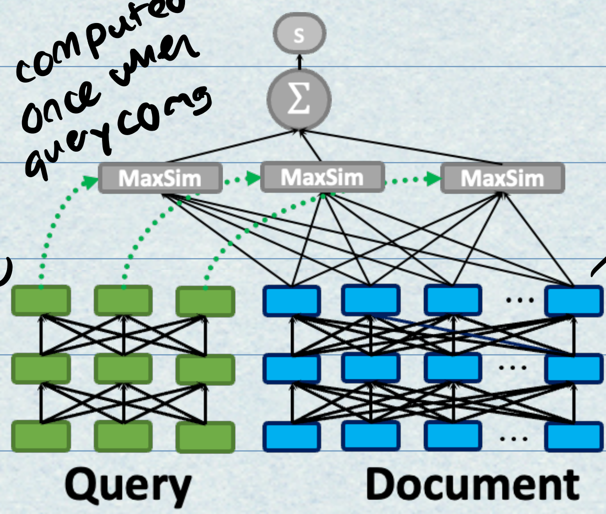
✓ Fine-Grained Interactions
✗ Expensive Joint Conditioning

- we didn't look at this specifically but this is like bm25 w/ better weights

LATE INTERACTION: keep precomputation, but have fine-grained interactions (ColBERT, Khattab & Zaharia)

- insight: feed BERT into query / each doc, but KEEP output embeddings corresponding to each token (don't feed in output to final linear layer)
  - ↳ recall model produces 1xd vector per token
- now for each query token embedding, compute max similarity score across all doc embeddings
  - sum all these partial scores.

can be computed once when query comes



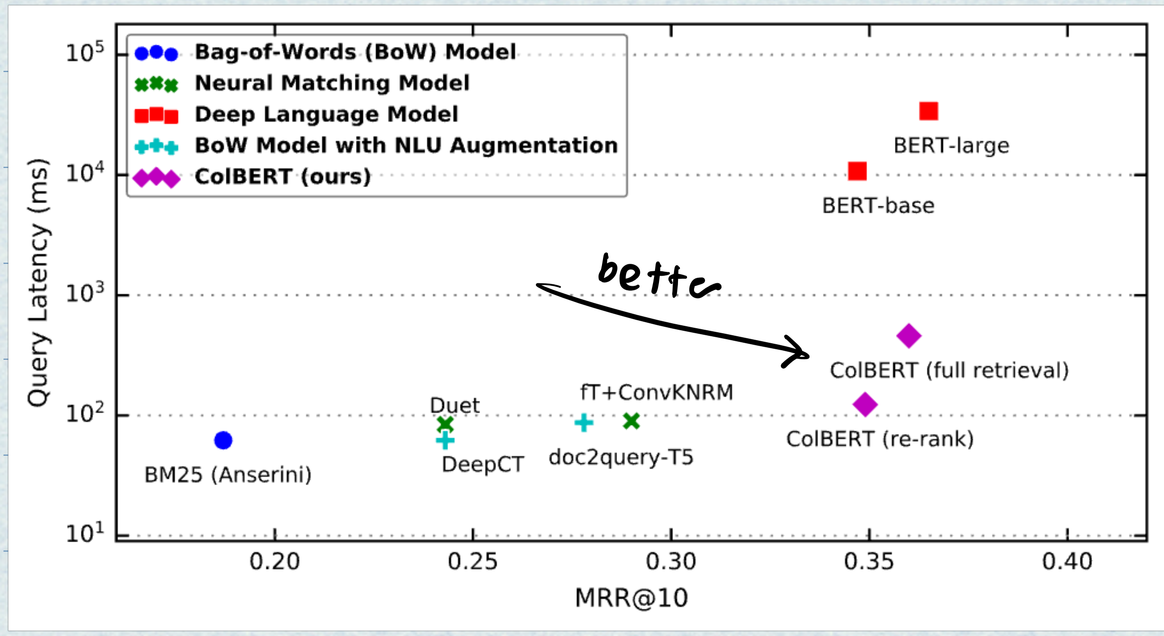
can be computed ahead of time

(d) Late Interaction
(i.e., ColBERT)

Omar Khattab and Matei Zaharia. "ColBERT: Efficient and effective passage search via contextualized late interaction over BERT." SIGIR'20.

- note that colbert represents doc as Nxd matrix (rather than a single vector)

- w/ late interaction, we can resolve mismatches from before!



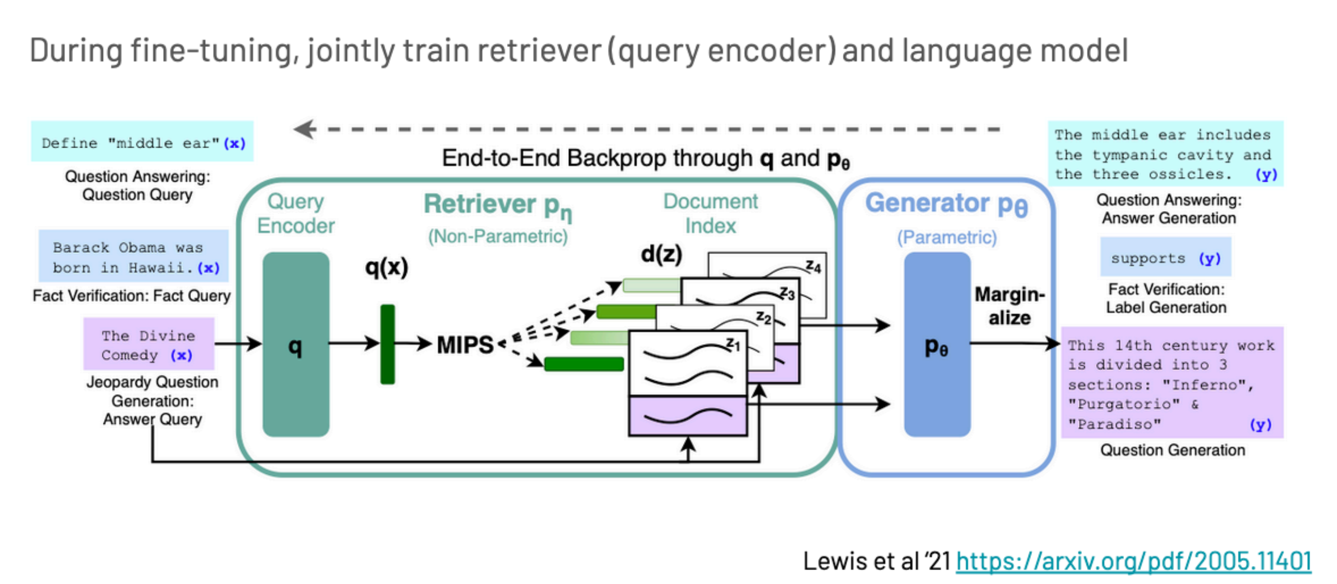
RAG: LLMs struggle on knowledge-intensive tasks

1. retrieve docs relevant to input from an external DB (either retriever OR vector DB)
2. Augment LLM input w/ retrieved docs.
3. Generate response on augmented input

- we need:
  - LM
  - retrieval model (turn query into embeddings)
  - DB w/ representations of docs - to search on query with

Early RAG approaches: fine-tune pretrained LM and query encoder:

(doc encoder frozen)



- can change weights of LM <u>OR</u> query encoder via backprop