Taming The Parrot: A Tutorial on Retrieval Augmented Generation

Luis L. Perez

Special thanks to C. Jones, G. Falcucci, Y. Renard and O. Yiakoumis

Objective

To use a language model to generate content given data sources that were not part of the model's training set.

- also known as *nonparametric* generation

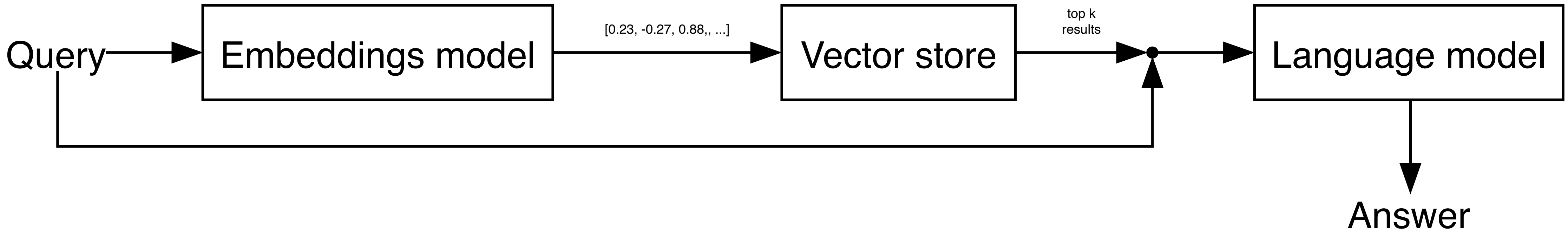
For example, QA *grounded* on a corpus of specialized documents.

- Or, code understanding given a specific codebase.

Often presented as an *alternative to fine-tuning* 🤔

- Cost-effective
- Easy to update
- Provides provenance

Overview of RAG



- *Embeddings model* and *Language model* are pre-trained
- *Vector store* is populated with retrieval corpus

Before we go on: some notes on language models

Main interface: `prompt` $\mapsto$ `completion`

"Simple" completion

`We hold these truths to be self-evident,`

$\mapsto$ `that all men are created equal, that they are` $\cdots$

Question answering

`Why should I switch to a plant-based diet?`

$\mapsto$ `Switching to a plant-based diet can have many health benefits,` $\cdots$

Instruct

`Given the TPC-H schema, write a SQL query to compute total revenue per region.`

$\mapsto$ `SELECT r.region_name, SUM(l.extended_price * (1 - l.discount))` $\cdots$

Before we go on: some notes on language models

Completion calls are stateless

No such thing as "memory" of previous interactions.

- Any notion of "state" lies on the application level

What about my chat interface?

🗣️	`Why should I switch to a plant-based diet?`
🤖	`Switching to a plant-based diet can have many health benefits` ...
🗣️	`Well, what if I don't want to?`

Before we go on: some notes on language models

Completion calls are stateless

What about my chat interface?

- Expanding history. The prompt for the last turn is

```
<|im_start|>system
You are a helpful assistant<|im_end|>
<|im_start|>user
Why should I switch to a plant-based diet?<|im_end|>
<|im_start|>assistant
Switching to a plant-based diet can have many health benefits … <|im_end|>
<|im_start|>user
Well, what if I don't want to?<|im_end|>
<|im_start|>assistant
```

(See ChatML documentation)

Before we go on: some notes on language models

Token limits

Tokens roughly correspond to words and symbols

- e.g. `The` ` quick` ` brown` ` fox` ` jumps` ` over` ` the` ` lazy` ` dog` `.`
- Not always, e.g. `Hola` `,` ` ¿` `c` `ómo` ` est` `ás` `?`

Models have limits on the amount of tokens they can process per call

- This is *total* token count, **input + output**
- e.g. GPT-3.5-Turbo has a limit of 4,096 ("extended" version with 16,384)

(See here for a notebook!)

The RAG prompt

```
Use the following pieces of context to answer the question at the end. If you don't
know the answer, just say that you don't know, don't try to make up an answer.

<context 1>
<context 2>
    ⋮
<context k>

Question: <question>
Helpful Answer:
```

(From langchain's retrieval_qa)

The central problem of RAG is *what* and *how much* `context` to put

⚠️ Remember: we have a token limit!

Retrieval: embeddings

Vector representations of text

Semantically similar pieces of text are closer in vector space

Embeddings model: **`chunk`** $\mapsto$ **`vector`**

```
from sentence_transformers import SentenceTransformer
E = SentenceTransformer("all-mpnet-base-v2")
my_encoding = E.encode("The quick brown fox jumps over the lazy dog")
```

```
>>> my_encoding.shape
(768,)

>>> my_encoding
array([-2.41035875e-02, -2.09894893e-03, -2.44234572e-03, -1.11331595e-02,
        1.96240544e-02,  3.31919305e-02, -1.19652543e-02,  2.41530929e-02,
        ...])
```

Retrieval: embeddings

Distance functions

Cosine: $d(u, v) = 1 - \frac{\sum_i u_i v_i}{\sqrt{\sum_i u_i^2 \times \sum_i v_i^2}}$

- Values between 0 and 2.
- Closer to 0 $\rightarrow$ more similar, i.e. $d(u, u) = 0$

```
from scipy.spatial.distance import cosine
def distance_texts(text1, text2):
    enc_1 = E.encode(text1)
    enc_2 = E.encode(text2)
    return cosine(enc_1, enc_2)
```

Retrieval: embeddings

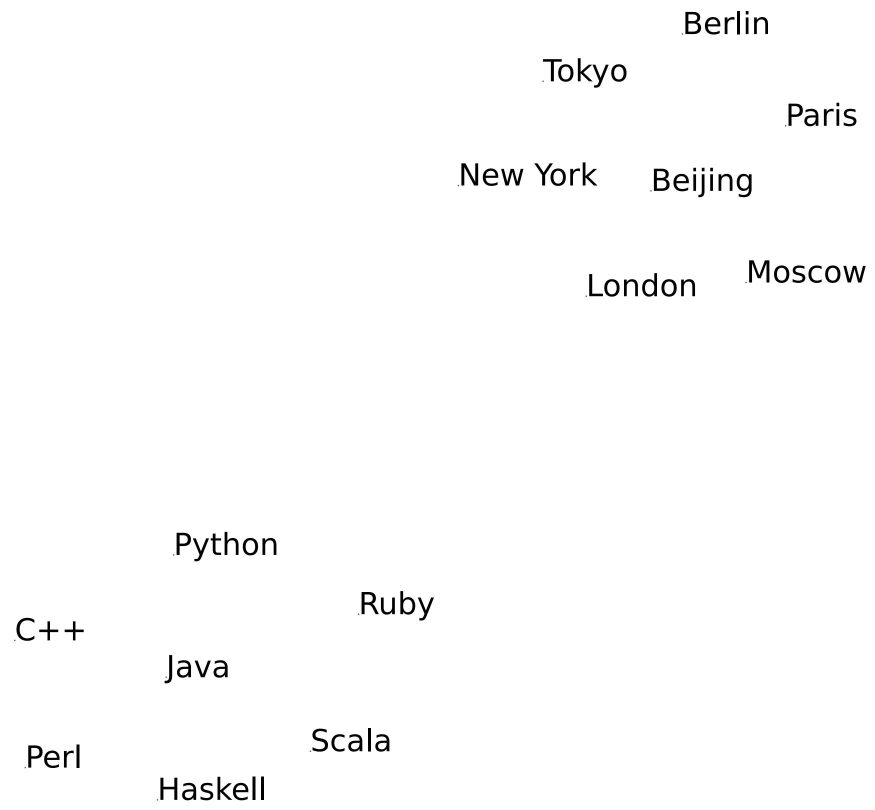
$T_1 :=$ `Do woodchucks actually chuck wood?`

$T_2 :=$ `Why should I switch to a plant-based diet?`

$T_3 :=$ `No. They 'chuck' on dirt as they build burrows.`

$$\texttt{distance\_texts}(T_2, T_3) \approx 0.94$$

$$\texttt{distance\_texts}(T_1, T_3) \approx 0.33$$ ✅



(See a notebook here)

Retrieval: nearest neighbors search

Problem: We have a *Query vector* v_Q and we want to search a database with N vectors for v_Q's k **nearest neighbors**

- That is, the k vectors with smallest distance to v_Q
- Which means: the k pieces of text with closest semantic similarity to our Query

Sounds great, what *is* the problem?

- Naive approach: calculate the distance between v_Q and the whole database, then pick the k vectors with smallest distance. What happens when N gets rather large?
- On my local machine: 100K entries $\approx$ 2s / 1M entries $\approx$ 80s ⚠️

Retrieval: nearest neighbors search

Indexing to the rescue!

Main idea: use a special data structure to search through vectors

- So that you only have to "look" at a few of them

Classical solution: space partitioning trees

- k-d trees, ball trees
  - available on `sklearn`
- Performance degrades with high-dimensional vectors

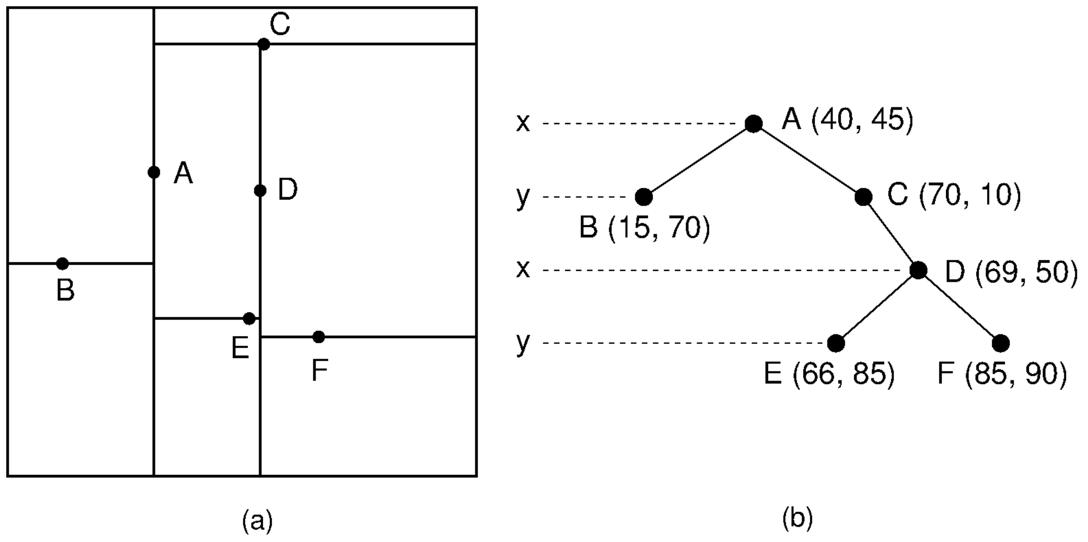


Image source: VTech CS3 class notes

Retrieval: nearest neighbors search

SOTA: neighborhood graph methods

- *approximate* nearest neighbors
- Hierarchical Navigable Small Worlds

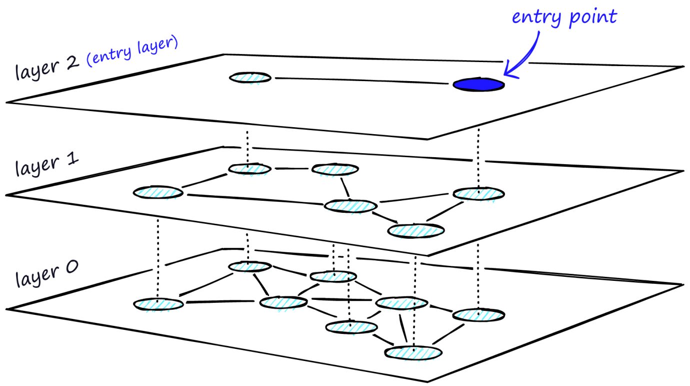


Image source: Pinecone documentation

Index creation

```
import faiss
index = faiss.IndexHNSWFlat(768, 32) # (dimension, neighbors)
index.add(my_database)               # ndarray shape (768,N)
```

Querying

```
distances, indices = index.search(query_vector, k) # distances L2
```


(See notebook here!)

Retrieval: vector stores

- A document database that can be queried with vectors
  - You give: vector, k, (optional *filtering metadata*)
  - You get: k documents with nearest indexing vectors
- Solves some important problems!
  1. Scaling beyond local memory
  2. Complex queries (narrowing with *filtering metadata*)
  3. Concurrency, fault tolerance, CRUD operations
- Lots of options from open source to enterprise: Chroma, Pinecone, Weaviate, Redis, ...

Populating the vector store

We've talked a lot about searching, but how do we build this vector store?

⚠️ **Important!** Embedding models have their own token limits

- e.g. `mpnet`'s is 384, `ada`'s is 8192.
- This means **we cannot embed whole long documents**

🤔 And even if we could, we still have to fit them within our RAG prompt!

The solution: chunking

- Split the document into chunks
  - Naive: 1,000 token document into $\mathtt{C}_1$ (0:384), $\mathtt{C}_2$ (384:768) and $\mathtt{C}_3$ (768:1000)
- Embed each chunk separately
- Insert each $\langle \mathtt{vector}, (\mathtt{chunk}, \mathtt{tags}) \rangle$ in the store as a separate entry
  - Tagged with "parent" document metadata

Populating the vector store

Consider this passage from Sutton & Barto chunked at ~30 tokens:

> In a *Markov* decision process, the probabilities given by p completely characterize the environment's dynamics. That is, the probability of each possible value for S_t and R_t depends only on the immediately preceding state and action, S_{t-1} and A_{t-1}, and, given them, not at all on earlier states and actions. This is best viewed a restriction not on the decision process, but on the *state*. The state must include information about all aspects of the past agent–environment interaction that make a difference for the future. If it does, then the state is said to have the *Markov property*. We will assume the Markov property throughout this book, though starting in Part II we will consider approximation methods that do not rely on it, and in Chapter 17 we consider how a Markov state can be learned and constructed from non-Markov observations.

Naive splitting can lead to loss of context

Becomes much worse with tables, math, code, etc.

Content-aware splitting takes into account text structure: sentence and paragraph boundaries, table elements, code separators such as brackets, function signatures

Populating the vector store

Chunking in `langchain`

```
contextual_splitter = (
  RecursiveCharacterTextSplitter
  .from_tiktoken_encoder(
    "cl100k_base",
    chunk_size=384,
    chunk_overlap=32
  )
)
docs = contextual_splitter.transform_documents(loaded_text)
```

(See an example notebook here)

- Must consider overlap, separators, type of content, etc.

Populating the vector store

Re: tags and metadata

- Provenance: where did we look at to answer `Query`?
  - At a bare minimum, docid.
  - Ideally, full position: page, chapter, section
- Structured retrieval: going beyond "let's look everywhere!"
  - Basic search pattern: consider all chunks in all documents
  - Better: narrow down search space by filtering on tags
    - Topics, entities, content type, etc.
    - Either human-defined or inferred (**more on this later!**)

Putting it all together

Let us recap the basic RAG workflow:

Building the vector store (Only once, for each document)

1. Load document
2. Split into chunks
3. Compute the vector embeddings of each chunk
4. Insert entries $\langle \mathtt{vector}, (\mathtt{chunk}, \mathtt{tags}) \rangle$ in the vector store

Query processing

1. Compute the vector embeddings of the query
2. Search for k nearest chunks
3. Construct RAG prompt with $\mathtt{Query} \oplus \mathtt{Chunks}$, give as input to language model

🤔 **Is this the end of the story?**

Re-ranking for relevance

Concern: are my top k chunks *really* relevant to answering the query?

- You will *always* get k chunks, no matter how distant
- Some irrelevant chunks might come through

Cross-encoder models

- Input: $(\texttt{Query}, \texttt{Passage})$ / Output: relevance score

```
from sentence_transformers import CrossEncoder
CE = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-6-v2')
question = "Do woodchucks actually chuck wood?"
```

```
>>> CE.predict((question, "Chuck is a common name in the English language."))
-6.2294426
```

```
>>> CE.predict((question, "No. They 'chuck' on dirt as they build burrows."))
-2.3234038
```

Re-ranking for relevance

Slight modification to our retrieval procedure:

1. Compute the vector embeddings of the query
2. Search for $K > k$ nearest chunks
3. **Apply cross-encoder on each of the K chunks w.r.t. the `Query`**
4. **Sort and keep only the k chunks with the highest score**
5. Construct RAG prompt with `Query` $\oplus$ `Chunks`, give as input to language model

Hypothetical document embeddings (HyDE)

Observation: we are searching our store for **questions** or **instructions** 🤔

- They don't "look" a lot like the answers
- Unless your documents are FAQs?

Idea: leverage hallucinations! 👽

- Given your `Query`, ask a language model to make up an answer
- Compute the embeddings for that answer, and use it to the search the vector store

Key points:

- It doesn't matter if the answer is wrong. What matters is that it *looks* like an answer
- You can use a different language model here (preferably a "wild" one)

Hypothetical document embeddings (HyDE)

The original HyDE prompt

```
Please write a passage to answer the question
Question: <query>
Passage:
```

(From langchain)

We take `Passage`, compute its vector embeddings and search our vector store.

- Then, do the rest of RAG flow.
- This plays quite nicely with re-ranking with `CE`!

Other interesting extensions

i.e. stuff I'm looking at right now!

Model-driven structured retrieval

- **Automatic**: use a model to tag documents
    - At query time, identify possible tags to narrow down search
- Or, **hierarchical**: use the LM to summarize documents
    - At query time, search against set of summaries first
    - Then, focus on the specific set of documents

"Function calling" to integrate structured sources

- Provide hints to the LM that **you** have certain functions
    - e.g. `run_sql(sql_query)`
- Run it, give back the answer to the LM
    - Interesting path, but challenging on correctness and security

Other interesting extensions

i.e. stuff I'm looking at right now!

Context order

- For some reason, chunk order in RAG prompt matters
- Middle chunks end up being more important
- Creative re-ranking?
- (See *Liu et al* paper in reference list)

Few-shot query expansion

- Generalizes HyDE to few-shot (HyDE is zero-shot)
- Prompt with real life labeled $(\mathtt{Query}, \mathtt{Passage})$ pairs to generate document
- Ground HyDE and allows for customization
- (See *Wang et al* paper in reference list)

Grab the packages

```
pip install \
  sentence-transformers \
  faiss-cpu \
  tiktoken \
  'unstructured[local-inference]' \
  langchain
```

Some references

- P. Lewis et al (2020). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*
- Y. Malkov and D. Yashunin (2016). *Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs*
- L. Gao et al (2022).*Precise Zero-Shot Dense Retrieval without Relevance Labels*
- F. Liu et al (2023). *Lost in the Middle: How Language Models Use Long Contexts*
- L. Wang et al (2023). *Query2doc: Query Expansion with Large Language Models*