Your First RAG

This chapter covers

- Extracting text from PDFs for language models
- Retrieving document context using keyword matching and vector embeddings
- Building a basic RAG pipeline for generating answers with relevant context

In this chapter, you are going to learn how to create your first RAG application for question answering over a document. Companies like Adobe are adopting these Q&A and chatting over documents as beta capabilities. Done right, these are powerful capabilities, empowering readers to gain novel insights from documents and save valuable time. Through building this app, you will encounter the multiple aspects necessary for successfully performing tasks like Q&A over documents. These include extracting information, retrieving the relevant context, and utilizing this context to generate accurate results.

RAG Q&A Document Preparation

By the end of this chapter, you will have built a question answering over a 10-Q financial document, taking the Amazon Q1 2023 financial statement as the representative document, following the steps shown in figure 1. First, we are going to discuss how to extract information from this document. Second, we will look at breaking the document into smaller chunks, to fit into LLM context windows. Third, we will discuss two strategies to save documents for future retrieval. One is storing the text as is for keyword based retrieval. The other is converting text into vector embeddings, for more efficient retrieval. Fourth, we will discuss saving this to a relevant database. Fifth, we will discuss obtaining relevant chunks based on user inputs. Finally, we will discuss how to incorporate relevant document chunks as part of LLM context, for generating the output. Steps 1 through 4 are referred to as the indexing pipeline, wherein documents are indexed in a database offline, prior to user interactions. Steps 5 and 6 happen in real-time as the user is querying the application.

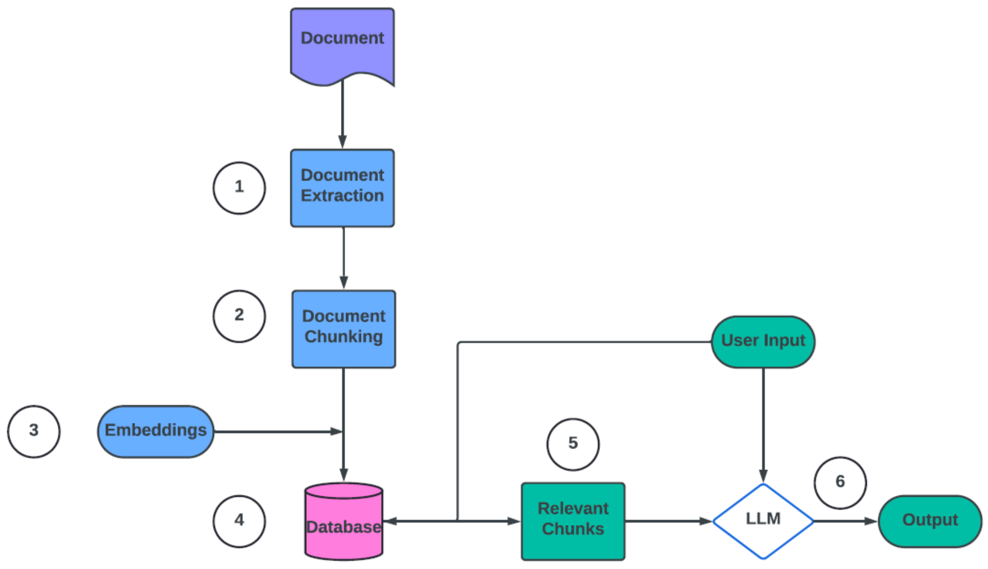


Figure 1 Components Of A Basic RAG Pipeline

The first step for answering questions over documents is to extract information as text for LLM. In my experience, the step of extraction is often the most overlooked factor, critical to the success of RAG applications. This is because ultimately, the quality of answers from the LLM depends on the data context that is provided. If this data has accuracy or consistency issues, this will lead to poor results overall. This section goes into the ways to extract data for RAG applications, focusing on extracting data from PDF documents in particular. You can think of the entire stage from extracting, to ultimately storing of data in the right database as similar to the traditional extract, transform, load (ETL) process where information is retrieved from an original data source, undergoes a series of modifications (including data cleansing and formatting), and is subsequently stored in a target data repository.

The basic way to extract text is to extract all the information from the PDF as a large string. This string can then be broken down into smaller chunks, to fit into LLM context windows.

PyMuDF is one such library that makes it easy to extract text from PDF documents as a string. There are other text parsers like PyPDF and PDFMiner with similar functionality. The advantage of PyMuPDF is that it supports parsing of multiple formats including txt, xps, images, etc., which is not possible in some of the other packages mentioned. Below, you can see how to extract text from the Amazon Q1-2023 PDF document using PyMuPDF, as a string:

Listing 1 Extracting Text Using PyMuPDF

```
import requests
import fitz
import io
```

```
url =
"https://s2.q4cdn.com/299287126/files/doc_financials/2023/q1/Q1-2023-A
mazon-Earnings-Release.pdf"
request = requests.get(url)
filestream = io.BytesIO(request.content)
with fitz.open(stream=filestream, filetype="pdf") as doc:
#concatenating text to a string and printing out the first 10
characters
    text = ""
    for page in doc:
        text += page.get_text()
print(text[:10])
```

Chunking Data

A natural first question is - why do all this, why not just send all the text to the LLM and let it answer questions? Let's take the Amazon Q1 2023 document for example. The entire text is ~50k characters. If you try passing all the text as context as follows, you get an error due to the context being too long.

Listing 2 Context Limits For LLMs

```
prompt=f"""What was the sales increase for Amazon in the first quarter
based on the context below?

Context:

```
{text+text}
```
"""

print(get_completion(prompt))
```

LLMs typically have a token limit (each token is roughly 3/4th a word). Let's see how to solve this with chunking. Chunking involves dividing a lengthy text into smaller sections that an LLM can process more efficiently.
Figure 2 outlines how to build a basic RAG that utilizes an LLM over custom documents for question answering. The first part is splitting multiple documents into manageable chunks. The associated parameter is the maximum chunk length. These chunks should be of the typical (minimum) size of text that contain the answers to the typical questions asked. This is because sometimes the question you ask might have answers at multiple locations within the document.

For example, you might ask the question “What was X company’s performance from 2015 to 2020?” And you might have a large document (or multiple documents) containing specific information about company performance over the years in different parts of the document. You would ideally want to capture all disparate parts of the document(s) containing this information, link them together, and pass to an LLM for answering based on these filtered and concatenated document chunks.

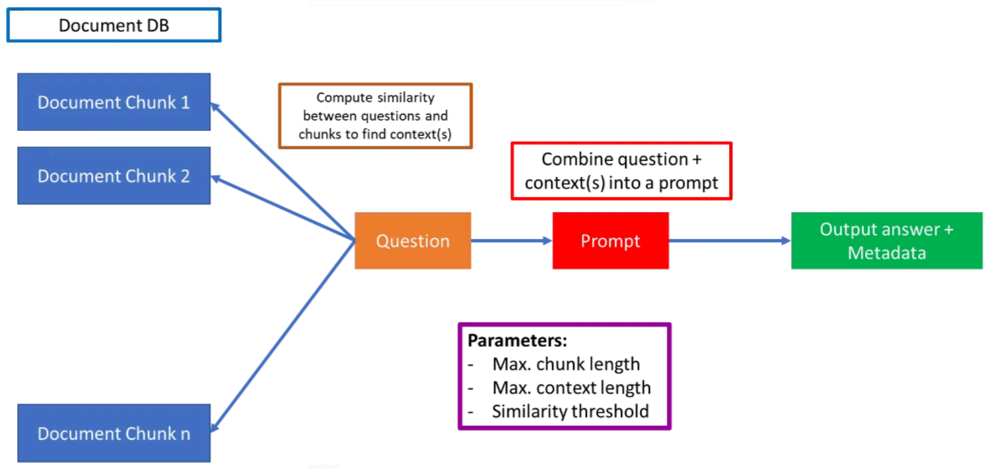


Figure 2 RAG Components

The maximum context length is basically the maximum length for concatenating various chunks together, leaving some space for the question itself and the output answer. Remember that LLMs like GPT3.5 have a strict length limit that includes all the content: question, context, and answer. Finding the right chunking strategy is crucial for building high quality RAG applications. There are different methods to chunk based on use-case. Here are five levels of chunking based on the complexity and effectiveness.

- **Fixed Size Chunking:** This is the most basic method, where the text is split into chunks of a specified number of characters, without considering the content or structure. It's simple to implement but may result in chunks that lack coherence or context.
- **Recursive Chunking:** This method splits the text into smaller chunks using a set of separators (like newlines or spaces) in a hierarchical and iterative manner. If the initial splitting doesn't produce chunks of the desired size, it recursively calls itself on the resulting chunks with a different separator.
- **Document Based Chunking:** In this approach, the text is split based on its inherent structure, such as markdown formatting, code syntax, or table layouts. This method preserves the flow and context of the content but may not be effective for documents lacking clear structure.
- **Semantic Chunking:** This strategy aims to extract semantic meaning from embeddings and assess the semantic relationship between chunks. It adaptively picks breakpoints between sentences using embedding similarity, keeping together chunks that are semantically related.

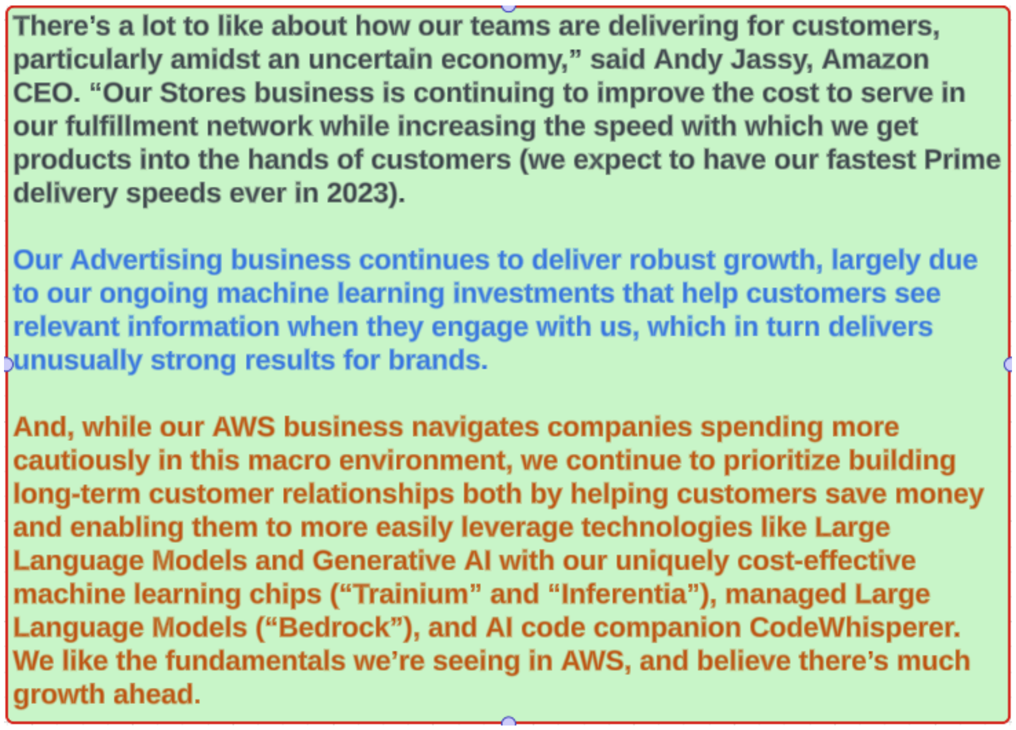
- **Agentic Chunking:** This approach explores the possibility of using a language model to determine how much and what text should be included in a chunk based on the context. It generates initial chunks using propositional retrieval and then employs an LLM-based agent to determine whether a proposition should be included in an existing chunk or if a new chunk should be created.

The similarity threshold is the way to compare the question with document chunks, to find the top chunks, most likely to contain the answer. Cosine similarity is the typical metric used, but you might want to weigh different metrics, such as including a keyword metric to weight contexts with certain keywords more. For example, you might want to weight contexts that contain the words “abstract” or “summary” when you ask the question to an LLM to summarize a document.

Let’s use simple fixed chunking in our first RAG app, splitting chunks by sentences where necessary. For this, we need to split up the texts into chunks, when they reach a provided maximum token length. The OpenAI tokenizer below, can be used to tokenize text, and calculate the number of tokens.

```
tokenizer = tiktoken.get_encoding("cl100k_base")

df=pd.DataFrame([text]).T
df.columns = ['text']

df['n_tokens'] = df.text.apply(lambda x: len(tokenizer.encode(x)))
```

This text can then be split into multiple contexts as below, for LLM comprehension, based on a token limit. For this, the text is split into sentences from the period delimiter, and sentences are appended to a chunk. If the chunk length is beyond the token limit, that chunk is truncated, and the next chunk is started. In figure 3, you can see an example of chunking by sentences, where three chunks are displayed as three distinct paragraphs.



There's a lot to like about how our teams are delivering for customers, particularly amidst an uncertain economy," said Andy Jassy, Amazon CEO. "Our Stores business is continuing to improve the cost to serve in our fulfillment network while increasing the speed with which we get products into the hands of customers (we expect to have our fastest Prime delivery speeds ever in 2023).

Our Advertising business continues to deliver robust growth, largely due to our ongoing machine learning investments that help customers see relevant information when they engage with us, which in turn delivers unusually strong results for brands.

And, while our AWS business navigates companies spending more cautiously in this macro environment, we continue to prioritize building long-term customer relationships both by helping customers save money and enabling them to more easily leverage technologies like Large Language Models and Generative AI with our uniquely cost-effective machine learning chips ("Trainium" and "Inferentia"), managed Large Language Models ("Bedrock"), and AI code companion CodeWhisperer. We like the fundamentals we're seeing in AWS, and believe there's much growth ahead.

Fig. 3 Sample Fixed Chunking By Sentences

Here is the split_into_many function that does the same:

Listing 3 Splitting Text Into Chunks

```
def split_into_many(text: str, tokenizer: tiktoken.Encoding,
max_tokens: int = 1024) -> list:
    """ Function to split a string into many strings of a specified
number of tokens """


    sentences = text.split('. ')
    n_tokens = [len(tokenizer.encode(" " + sentence))
                for sentence in sentences]

    chunks = []
    tokens_so_far = 0
    chunk = []

    for sentence, token in zip(sentences, n_tokens):

        if tokens_so_far + token > max_tokens:
            chunks.append(". ".join(chunk) + ".")
            chunk = []
            tokens_so_far = 0

        if token > max_tokens:
            continue
        chunk.append(sentence)
        tokens_so_far += token + 1

    return chunks
```

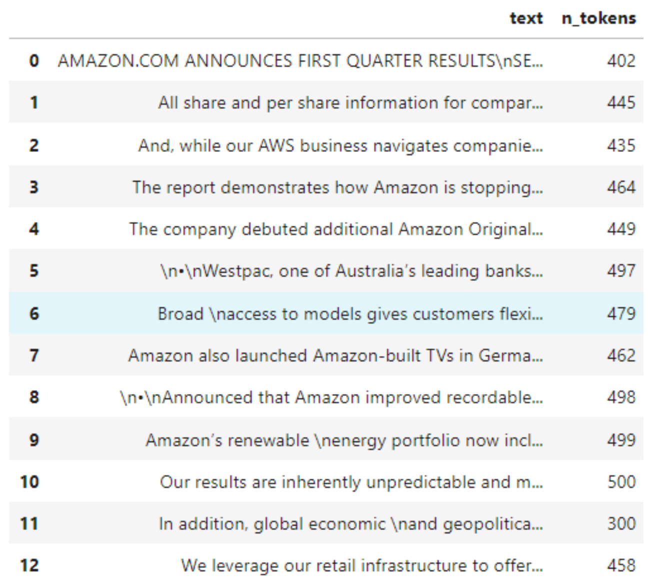
Finally, you can tokenize the entire text by calling the tokenize function, that concatenates the logic from above:

Listing 4 Tokenizing Text Chunks

```
def tokenize(text,max_tokens) -> pd.DataFrame:
    """ Function to split the text into chunks of a maximum number of
tokens """


    tokenizer = tiktoken.get_encoding("cl100k_base")

    df=pd.DataFrame(['0',text]).T
    df.columns = ['title', 'text']

    df['n_tokens'] = df.text.apply(lambda x: len(tokenizer.encode(x)))

    shortened = []

    for row in df.iterrows():

        if row[1]['text'] is None:
            continue

        if row[1]['n_tokens'] > max_tokens:
            shortened += split_into_many(row[1]['text'], tokenizer,
max_tokens)
```

```
        Else:
            shortened.append(row[1]['text'])


    df = pd.DataFrame(shortened, columns=['text'])
    df['n_tokens'] = df.text.apply(lambda x: len(tokenizer.encode(x)))



    return df
```

In figure 4, you can see how the entire dataframe looks, after running `tokenize(text,500)`. Each chunk is a separate row, and there are 13 chunks in total. The chunk text is in the ‘text’ column, the number of tokens for that text in the ‘n_tokens’ column.



	text	n_tokens
0	AMAZON.COM ANNOUNCES FIRST QUARTER RESULTS\nSE...	402
1	All share and per share information for compar...	445
2	And, while our AWS business navigates companie...	435
3	The report demonstrates how Amazon is stopping...	464
4	The company debuted additional Amazon Original...	449
5	\n•\nWestpac, one of Australia's leading banks...	497
6	Broad \naccess to models gives customers flexi...	479
7	Amazon also launched Amazon-built TVs in Germa...	462
8	\n•\nAnnounced that Amazon improved recordable...	498
9	Amazon's renewable \nenergy portfolio now incl...	499
10	Our results are inherently unpredictable and m...	500
11	In addition, global economic \nand geopolitica...	300
12	We leverage our retail infrastructure to offer...	458

Fig. 4 Chunked Data

Retrieval Methods

The next step, after document extraction and chunking, is to store these documents in an appropriate format so that relevant documents or passages can be easily retrieved in response to future queries. In the following sections, you are going to see two characteristic methods to retrieve relevant LLM context: keyword based retrieval and vector embeddings based retrieval.

Keyword Based Retrieval

The easiest way to sort relevant documents is to do a keyword match and find documents with the highest match. For this, we need to first define a way to match documents based on keywords. In information retrieval, two important concepts form the foundation of many ranking algorithms: Term Frequency (TF) and Inverse Document Frequency (IDF).
Term Frequency (TF) measures how often a term appears in a document. It's based on the assumption that the more times a term occurs in a document, the more relevant that document is to the term.

TF(t,d) = Number of times term t appears in document d / Total number of terms in document d

Inverse Document Frequency (IDF) measures the importance of a term across the entire corpus of documents. It assigns higher importance to terms that are rare in the corpus and lower importance to terms that are common.

IDF(t) = Total number of documents / Number of documents containing term t

The TF-IDF score is then calculated by multiplying TF and IDF:
TF-IDF(t,d) = TF(t,d) * IDF(t)

While TF-IDF is useful, it has limitations. This is where the Okapi BM25 algorithm comes in, offering a more sophisticated approach to document ranking.

The Okapi BM25 is a common algorithm for matching documents based on keywords, as shown in figure 5.

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i,D)\cdot(k_1+1)}{f(q_i,D)+k_1\cdot\left(1-b+b\cdot\frac{|D|}{\text{avgdl}}\right)}$$

Figure 5 Okapi BM25 Algorithm

Given a query Q containing keywords q1,q2,...the BM25 score of a document D is as above. The function f(qi, D) is the number of times qi occurs in D, and k1, b are constants. IDF denotes the inverse document frequency of the word qi. IDF measures the importance of a term in the entire corpus. It assigns higher importance to terms that are rare in the corpus and lower importance to terms that are common. This is used to normalize contributions of common words like 'The', or 'and' from search results. avgdl is the average document length in the text collection from which documents are drawn.

The BM25 formula can be understood as an extension of TF-IDF:

1. It uses IDF to weigh the importance of terms across the corpus, similar to TF-IDF.

2. The term frequency component (f(qi,D)) is normalized using a saturation function, which prevents the score from increasing linearly with term frequency. This addresses a limitation of basic TF-IDF.
3. It incorporates document length normalization (|D| / avgdl), adjusting for the fact that longer documents are more likely to have higher term frequencies simply due to their length.

By considering these additional factors, BM25 often provides more accurate and nuanced document ranking compared to simpler TF-IDF approaches, making it a popular choice in information retrieval systems.

The BM25 algorithm returns a value between 0 (no keyword overlaps between Query and Document) and 1 (Document contains all keywords in Query). For example, if the user input is “windy day” and the document is “It is quite windy” - the BM25 algorithm would yield a non-zero result. Here is a snippet of a Python implementation of BM25:

Listing 5 BM25 Based Keyword Retrieval

```
from rank_bm25 import BM25Okapi

corpus = [
    "Hello there how are you!",
    "It is quite windy in Boston",
    "How is the weather tomorrow?"
]
tokenized_corpus = [doc.split(" ") for doc in corpus]
bm25 = BM25Okapi(tokenized_corpus)
query = "windy day"
tokenized_query = query.split(" ")
doc_scores = bm25.get_scores(tokenized_query)
have keyword overlap with the query.
Output:
array([0.        , 0.48362189, 0.        ]) #Only the second document
and the query have an overlap, the others do not
```

The user input here is “windy day”. As you can see, there is an overlap between the second document in the corpus (“It is quite windy in Boston”), and the input, which is reflected by the second score being the highest (0.48).

However, you also see how the third document (“How is the weather tomorrow?”) is related to the input (as both discuss the weather). We would like the third document to have some non-zero score.This is where the concept of semantic similarity and vector embeddings comes in. A classic example of this is where the user searches for “Wild West” and expects information about cowboys. Semantic search means that the algorithm is intelligent enough to know that cowboys and the wild west are similar concepts (while having different words). This becomes important for RAG as it is quite possible the user types in a query that is not exactly present in the document, for which we need a good measure of semantic similarity to find the relevant documents, according to the users intent.

Vector Embeddings

Vector search helps in choosing what the relevant context is when you have vast amounts of data, including hundreds or more documents.Vector search is a technique in information retrieval and machine learning that uses vector representations of data points to efficiently find similar items in a large dataset. It involves encoding data into high-dimensional vectors and using distance metrics to measure similarity between these vectors.
In figure 6, you can see a simplified two-dimensional vector space:
X-axis: Size (small = 0, big = 1)
Y-axis: Type (tree = 0, animal = 1)
This example illustrates both direction and magnitude:
A small tree might be represented as (0, 0)
A big tree as (1, 0)
A small animal as (0, 1)
A big animal as (1, 1)
The direction of the vector indicates the combination of features, while the magnitude (length) of the vector represents the strength or prominence of those features.
This is just a conceptual example and can be scaled to hundreds or more dimensions, each representing different attributes of the data. In real-world applications, these vectors often have much higher dimensionality, allowing for more nuanced representations of complex data.
The same can also be done with text as below, and yields better semantic similarity as compared to keyword search. An appropriate embedding algorithm would be able to judge which contexts are most relevant to user input, and which contexts are not as relevant, crucial for the retrieval step in RAG applications. Once this relevant context is found, this can be added to the user input, and passed to an LLM for generating the appropriate output, sent back to the user.

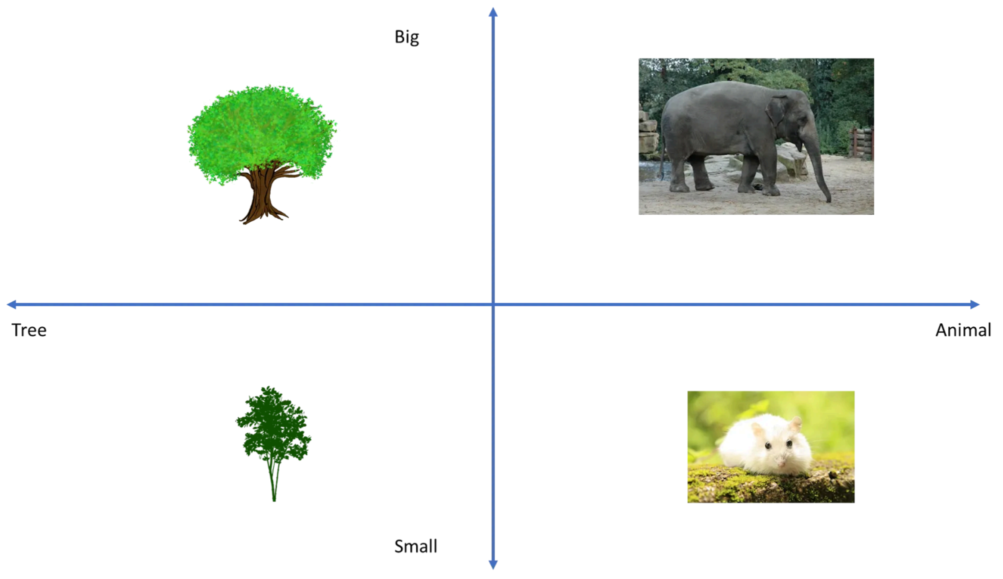


Fig. 6 Vector Search 101

Notice how in the figure 7, the vectorization is able to capture the semantic representation (i.e,. it knows that a sentence talking about a bird swooping in on a baby chipmunk should be in the (small, animal) quadrant, whereas the sentence talking about yesterday's storm when a large tree fell on the road should be in the (big, tree) quadrant). In reality, there are more than two dimensions. For example, the OpenAI embedding model has 1,536 dimensions.

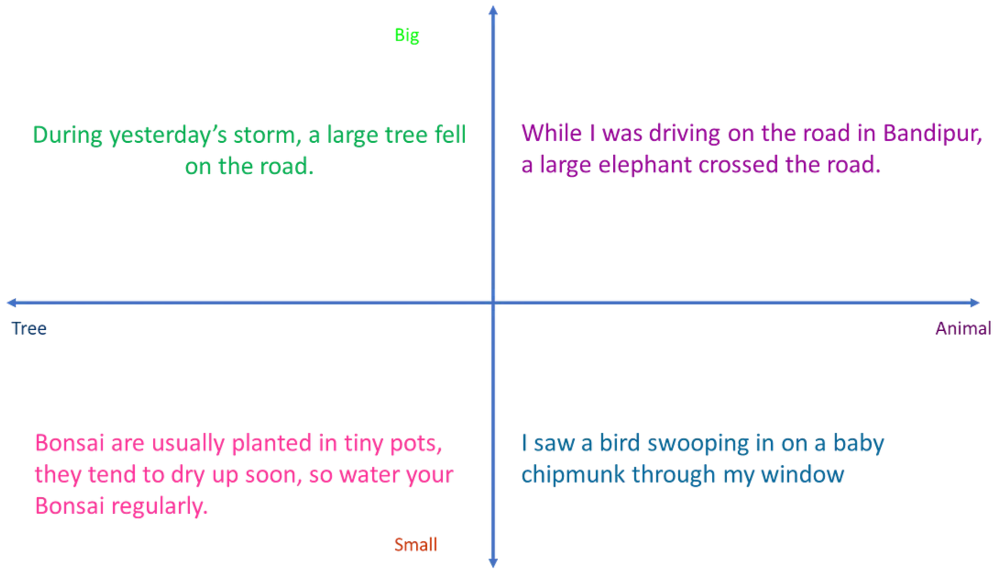

Big
During yesterday's storm, a large tree fell on the road.
While I was driving on the road in Bandipur, a large elephant crossed the road.
Tree
Animal
Bonsai are usually planted in tiny pots, they tend to dry up soon, so water your Bonsai regularly.
I saw a bird swooping in on a baby chipmunk through my window
Small

Fig. 7 Vector Search 101 With Words

Obtaining embeddings is quite easy from OpenAI's embedding model. In chapter 3 we will benchmark embedding models. For the rest of this chapter, we will use OpenAI embedding and

LLM models. The OpenAI embedding model costs $0.10 /1M tokens, where each token is roughly 3/4th a word. A token is a word/subword. When text is passed through a tokenizer, it encodes the input based on a specific scheme and emits specialized vectors that can be understood by the LLM. The cost is quite minimal - roughly 10 cents per 3000 pages, but can add up as the number of documents and users scale up.

Listing 6 Vector Embeddings

```
import openai
from getpass import getpass
api_key = getpass('Enter the OpenAI API Key in the cell  ')

client = openai.OpenAI(api_key=api_key)
openai.api_key =api_key



def get_embedding(text, model="text-embedding-ada-002"):
 return client.embeddings.create(input = [text],
model=model).data[0].embedding


e1=get_embedding('the boy went to a party')
e2=get_embedding('the boy went to a party')
e3=get_embedding("""We found evidence of bias in our models via
running the SEAT (May et al, 2019) and the Winogender (Rudinger et al,
2018) benchmarks. Together, these benchmarks consist of 7 tests that
measure whether models contain implicit biases when applied to
gendered names, regional names, and some stereotypes.

For example, we found that our models more strongly associate (a)
European American names with positive sentiment, when compared to
African American names, and (b) negative stereotypes with black
women.""")
```

The first two texts (corresponding to embeddings e1 and e2) are the same, so we would expect their embeddings to be the same, while the third text is completely different. To find the similarity between embedding vectors, we use cosine similarity. Cosine similarity measures the similarity between two vectors, measured by the cosine of the angle between the two vectors. Cosine similarity of 0 means that these texts are completely different, whereas cosine similarity of 1 implies identical or near identical text. We use the query below to find the cosine similarity:

```
1-spatial.distance.cosine(e1,e2)
```

```
Output:
1
1-spatial.distance.cosine(e1,e3)
Output:
0.69
```

As you can see, the cosine similarity (1-the cosine distance) is 1 for the same text, but less than 1 for the texts that are different.

Vector Embeddings For Finding Relevant Context

Let's now see how well vector embeddings do for choosing the right context for answering a question. Let's say we want to ask this question, corresponding to Q1 2023 for Amazon and ask the question below:

Listing 7 GPT Completions Endpoint For LLM Calls

```
prompt="""What was the sales increase for Amazon in the first
quarter?"""
```

We can get the answer from the GPT3.5 (ChatGPT) API as below:

```
def get_completion(prompt, model="gpt-3.5-turbo"):
    response = openai.chat.completions.create(
                   model="gpt-3.5-turbo",
                   temperature=0,
        messages=[{"role": "user", "content": prompt}]
    )

    return response.choices[0].message.content
```

```
Answer:
The sales increase for Amazon in the first quarter was 9%, with net
sales increasing to $127.4 billion compared to $116.4 billion in the
first quarter of 2022. Excluding the impact of foreign exchange rates,
the net sales increased by 11% compared to the first quarter of 2022.
```

As you can see, while the above answer is not wrong, it is not the one we are looking for (we are looking for the sales increase for Q1 2023, not Q1 2022). So it is important to feed the right context to the LLM - in this case, this would be context related to sales performance in Q1 2023. Let's say we have a choice of the three contexts below to append to the LLM:

Listing 8 Example Contexts

```
context1="""Net sales increased 9% to $127.4 billion in the first
quarter, compared with $116.4 billion in first quarter 2022.
Excluding the $2.4 billion unfavorable impact from year-over-year
changes in foreign exchange rates throughout the
quarter, net sales increased 11% compared with first quarter 2022.
North America segment sales increased 11% year-over-year to $76.9
billion.
International segment sales increased 1% year-over-year to $29.1
billion, or increased 9% excluding changes
in foreign exchange rates.
AWS segment sales increased 16% year-over-year to $21.4 billion."""

context2="""Operating income increased to $4.8 billion in the first
quarter, compared with $3.7 billion in first quarter 2022. First
quarter 2023 operating income includes approximately $0.5 billion of
charges related to estimated severance costs.
North America segment operating income was $0.9 billion, compared with
operating loss of $1.6 billion in
first quarter 2022.
International segment operating loss was $1.2 billion, compared with
operating loss of $1.3 billion in first
quarter 2022.
AWS segment operating income was $5.1 billion, compared with operating
income of $6.5 billion in first
quarter 2022.
"""

context3="""Net income was $3.2 billion in the first quarter, or $0.31
per diluted share, compared with net loss of $3.8 billion, or
$0.38 per diluted share, in first quarter 2022. All share and per
share information for comparable prior year periods
```

```
throughout this release have been retroactively adjusted to reflect
the 20-for-1 stock split effected on May 27, 2022.
• First quarter 2023 net income includes a pre-tax valuation loss of
$0.5 billion included in non-operating
expense from the common stock investment in Rivian Automotive, Inc.,
compared to a pre-tax valuation loss
of $7.6 billion from the investment in first quarter 2022."""
```

Measuring the cosine similarity between the query embeddings and three context embeddings, shows that the context1 has the highest cosine similarity with the query embeddings. Thus, appending this context to the user input and sending it to the LLM is more likely to give an answer relevant to the user input. We can feed this relevant context into the prompt as follows:

~~~
prompt=f"""What was the sales increase for Amazon in the first quarter
based on the context below?
Context:
```
{context1}
```
"""
print(get_completion(prompt))
~~~

The answer given by the LLM is now the one we wanted as below, since it is the sales increase for Q1 2023:

```
The sales increase for Amazon in the first quarter was 9% based on the
reported net sales of $127.4 billion compared to $116.4 billion in the
first quarter of the previous year.
```

Augmented Generation

The steps discussed above are for preparing documents for when the user interacts with the RAG application by posing a query.

In this section we are going to look at how to use the chunked, embedded information as relevant context when the user queries the application. This step is to retrieve the context in real-time based on user input, and use the retrieved context to generate the LLM output. Let's take the example that the user input is the question "What was the sales increase for Amazon in the first quarter?" based on the 10-Q Amazon document for Q1 2023. To answer this question, we have to first find the right contexts from the document chunks created above.

Let's define a create_context function for this. As you can see, the create_context function below requires three inputs for this - the user input query to embed, the dataframe containing the documents to find the subset of relevant context(s) to the user input, and the maximum context length, as shown in figure 8.

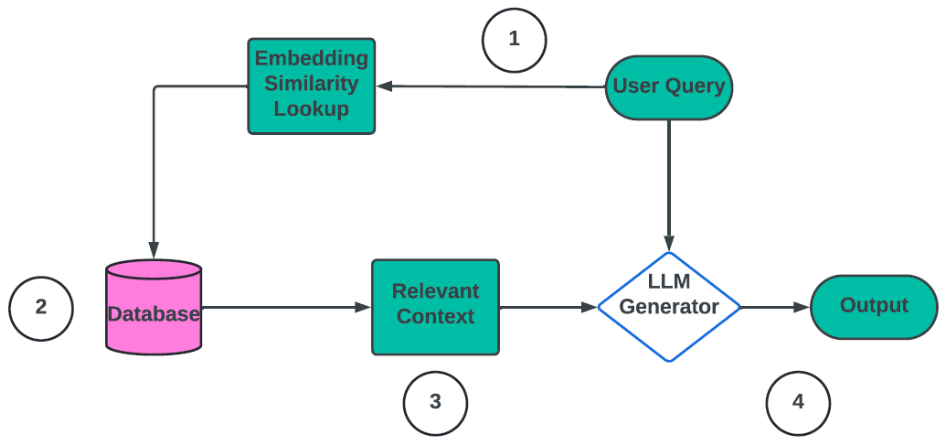


Fig. 8 Retrieval And Generation

The logic here is to get the embeddings for the question as in the figure above, compute pairwise distances between the input query embedding, and context embeddings (step 2), and append these contexts, ranked by similarity (step 3). If the running context length is greater than the maximum context length, the context is truncated. Finally, both the user query and relevant context are sent to the LLM, for generating the output.

Listing 9 Creating Context

```
def create_context(question: str, df: pd.DataFrame,max_len: int =
1800) -> str:
    """
    Create a context for a question by finding the most similar
context from the dataframe
    """

    q_embeddings = get_embedding(question)

    df['distances'] = df['embeddings'].apply(lambda x:
spatial.distance.cosine(q_embeddings,x))

    returns = []
    cur_len = 0

    for i, row in df.sort_values('distances',
ascending=True).iterrows():

        cur_len += row['n_tokens']
        if cur_len > max_len:
            break

        returns.append(row["text"])

    return "\n\n###\n\n".join(returns)
```

)

Here is the query and corresponding partial context created from running this line below:

`create_context("What was the sales increase for Amazon in the first quarter",df)`

Listing 10 Example Context

```
AMAZON.COM ANNOUNCES FIRST QUARTER RESULTS\nSEATTLE—(BUSINESS WIRE)
April 27, 2023—Amazon.com, Inc. (NASDAQ: AMZN) today announced
financial results \nfor its first quarter ended March 31, 2023.
\n•\nNet sales increased 9% to $127.4 billion in the first quarter,
compared with $116.4 billion in first quarter 2022.\nExcluding the
$2.4 billion unfavorable impact from year-over-year changes in foreign
exchange rates throughout the\nquarter, net sales increased 11%
compared with first quarter 2022.\n•\nNorth America segment sales
increased 11% year-over-year to $76.9 billion….
```

As you can see, the context is quite relevant. However, this is not formatted well. This is where the LLM shines as below, where the LLM can answer the questions from the created context:

Listing 11 LLM Generator

```
def answer_question(
    df: pd.DataFrame,
    question: str
):
    """
    Answer a question based on the most similar context from the
dataframe texts
    """
    context = create_context(
        question,
        df
```

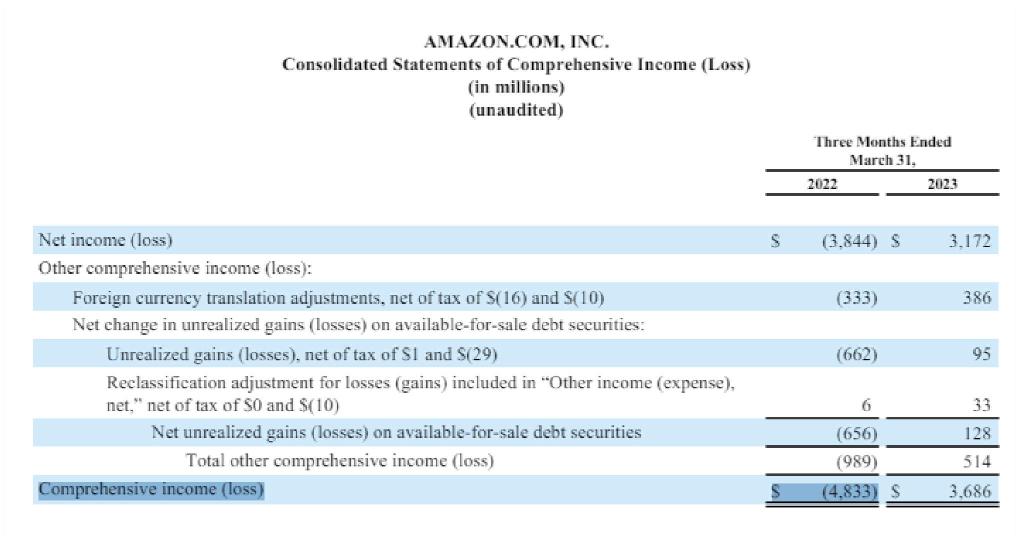
~~~
    )


    prompt=f"""Answer the question based on the context provided.

    Question:
    ```{question}.```

    Context:
    ```{context}```
    """

    response = openai.chat.completions.create(
                    model="gpt-3.5-turbo",
                    temperature=0,
        messages=[{"role": "user", "content": prompt}]
    )

    return response.choices[0].message.content
~~~

Finally, here is the corresponding answer generated from the query, and dataframe:

```
answer_question(df, question="What was the sales increase for Amazon
in the first quarter")
```

`The sales increase for Amazon in the first quarter was 9%, reaching $127.4 billion compared to $116.4 billion in the first quarter of 2022.`

Congratulations, you have now built your first RAG app. While this works well for questions where the answer is explicit within the text context, the answer is not always accurate when retrieved from tables. Let’s ask the question “What was the Comprehensive income (loss) for Amazon for the Three Months Ended March 31, 2022?” - where the answer is present in a table as $ 4,833 million as shown in figure 9:


AMAZON.COM, INC.
Consolidated Statements of Comprehensive Income (Loss)
(in millions)
(unaudited)

	Three Months Ended March 31, 2022	Three Months Ended March 31, 2023
Net income (loss)	$ (3,844)	$ 3,172
Other comprehensive income (loss):		
Foreign currency translation adjustments, net of tax of $(16) and $(10)	(333)	386
Net change in unrealized gains (losses) on available-for-sale debt securities:		
Unrealized gains (losses), net of tax of $1 and $(29)	(662)	95
Reclassification adjustment for losses (gains) included in "Other income (expense), net," net of tax of $0 and $(10)	6	33
Net unrealized gains (losses) on available-for-sale debt securities	(656)	128
Total other comprehensive income (loss)	(989)	514
Comprehensive income (loss)	$ (4,833)	$ 3,686

Fig. 9 Answer Within A Table

The answer from the application is:
The Comprehensive income (loss) for Amazon for the Three Months Ended March 31, 2022 was a net loss of $3.8 billion.
As you can see, it gave the net income (loss), instead of the comprehensive income (loss). This illustrates the limitations of the basic RAG architecture we built.
In the next chapter, we will learn about advanced document extraction, chunking, and retrieval mechanisms that build on the concepts learnt here. We will learn about evaluating the quality of responses from our RAG application using various metrics. We will learn how to use different techniques, guided by evaluation results, and make iterative improvements to performance.

Summary

- Extracting data in a format that is readable by LLMs is the first step, often overlooked for developing a RAG application. Libraries like PyMuPDF can extract text from PDFs.
- The next step after document extraction, comes the retrieval of relevant documents or passages, corresponding to input queries. For example, you might have tens or hundreds of documents, but only one of them is relevant.
- The easiest way to sort relevant documents is to do a keyword match, and find documents with the highest match. For this, we need to first define a way to match documents based on keywords. In information retrieval, the Okapi BM25 is a common algorithm for matching documents based on keywords.
- Vector search helps in choosing what the relevant context is, when you have vast amounts of data, including 100s (or more) documents. The same can also be done with text, and text embeddings yield better semantic similarity as compared to keyword search.

- Document chunking is important for retrieving high quality results by converting documents into manageable pieces, getting embeddings, and ranking chunks based on similarity to the query.
- The augmented generation aspect of RAG corresponds to using the real-time retrieved content to generate the LLM output.
- The above basic RAG prototype works well for cases where the text explicitly contains the relevant information, but not for advanced use-cases such as when information is in tables, or asking for summaries.