RAG_and_LangChain_splitting_embedding_round2

November 30, 2023

#Document Splitting

LangChain lib is proposing different type of text_splitter:

- CharacterTextSplitter
- RecursiveCharacterTextSplitter
- TokenTextSplitter
- HTMLHeaderTextSplitter
- MarkdownHeaderTextSplitter
- PythonCodeTextSplitter
- LatexTextSplitter
- ....

We'll be using the first 4 in this notebook

[ ]:
```
!pip install langchain
```

0.1 Load a pdf

[2]:
```
!pip install pypdf
```

```
Collecting pypdf
  Downloading pypdf-3.17.1-py3-none-any.whl (277 kB)
                             277.6/277.6
kB 4.8 MB/s eta 0:00:00
Installing collected packages: pypdf
Successfully installed pypdf-3.17.1
```

Here, we will be chating with the RAG paper: "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks"

[3]:
```
pdf_url = "https://arxiv.org/pdf/2005.11401.pdf"
from langchain.document_loaders import PyPDFLoader
loader = PyPDFLoader(pdf_url)
pages = loader.load()

print(f"Number of pages: {len(pages)}")

# each page is a document having two elements: page_content and metadata
```

```
page = pages[0]

print("######## PAGE CONTENT ########")
print(page.page_content[:400])

print("\n######## PAGE METDATA ########")
print(page.metadata)
```

```
Number of pages: 19
######## PAGE CONTENT ########
Retrieval-Augmented Generation for
Knowledge-Intensive NLP Tasks
Patrick Lewis†‡, Ethan Perez ,
Aleksandra Piktus†, Fabio Petroni†, Vladimir Karpukhin†, Naman Goyal†, Heinrich
Küttler†,
Mike Lewis†, Wen-tau Yih†, Tim Rocktäschel†‡, Sebastian Riedel†‡, Douwe Kiela†
†Facebook AI Research;‡University College London; New York University;
plewis@fb.com
Abstract
Large pre-trained language models have be

######## PAGE METDATA ########
{'source': 'https://arxiv.org/pdf/2005.11401.pdf', 'page': 0}
```

0.2 Text Splitter: RecursiveCharacterTextSplitter, CharacterTextSplitter

[4]:
```
from langchain.text_splitter import RecursiveCharacterTextSplitter,␣
↪CharacterTextSplitter
```

0.2.1 Toy Examples

[37]:
```
text = """
Pre-\ntrained models with a differentiable access mechanism to explicit␣
↪non-parametric\nmemory have so far been only investigated for extractive␣
↪downstream tasks.

We\nexplore a general-purpose  ne-tuning recipe for retrieval-augmented␣
↪generation\n(RAG) - models which combine pre-trained parametric and␣
↪non-parametric mem-\nory for language generation.
We introduce RAG models where the parametric\nmemory is a pre-trained seq2seq␣
↪model and the non-parametric memory is a dense\nvector index of Wikipedia,␣
↪accessed with a pre-trained neural retriever.
"""
```

[42]:
```
chunk_size = 200
chunk_overlap = 0
```

```
character_splitter = CharacterTextSplitter(
    chunk_size=chunk_size,
    chunk_overlap=chunk_overlap, #When you want n characters at the end of the␣
 ↪sequence to be repeated in the following one
    separator='. '
)

character_splitter.split_text(text)
# by default the character is the "newline character"
```

[42]:
```
['Pre-\ntrained models with a differentiable access mechanism to explicit non-
parametric\nmemory have so far been only investigated for extractive downstream
tasks',
 'We\nexplore a general-purpose  ne-tuning recipe for retrieval-augmented
generation\n(RAG) - models which combine pre-trained parametric and non-
parametric mem-\nory for language generation',
 'We introduce RAG models where the parametric\nmemory is a pre-trained seq2seq
model and the non-parametric memory is a dense\nvector index of Wikipedia,
accessed with a pre-trained neural retriever.']
```

[41]:
```
recursive_splitter = RecursiveCharacterTextSplitter(
    chunk_size=chunk_size,
    chunk_overlap=chunk_overlap,
    separators = ['\n\n','. ']
)

recursive_splitter.split_text(text)
```

[41]:
```
['Pre-\ntrained models with a differentiable access mechanism to explicit non-
parametric\nmemory have so far been only investigated for extractive downstream
tasks.',
 'We\nexplore a general-purpose  ne-tuning recipe for retrieval-augmented
generation\n(RAG) - models which combine pre-trained parametric and non-
parametric mem-\nory for language generation',
 '. \nWe introduce RAG models where the parametric\nmemory is a pre-trained
seq2seq model and the non-parametric memory is a dense\nvector index of
Wikipedia, accessed with a pre-trained neural retriever.\n']
```

0.2.2 Chunking the PDF

[100]:
```
chunk_size = 1500
chunk_overlap = 150

r_splitter = RecursiveCharacterTextSplitter(
    chunk_size=chunk_size,
    chunk_overlap=chunk_overlap,
```

```
    separators = ['\n\n','\n','. ']
)

chunks = r_splitter.split_documents(pages)
```

[ ]:
```
# chunks[:2]
```

To note: With LangChain each chunk is a document type in which there are the page_content and the metada sources. This one will be useful for contextual retrieval

---

[44]:
```
print(f"Number of pages={len(pages)}")
print(f"Number of chunks={len(chunks)}")
print(f"in the second page: the text length is {len(pages[1].page_content)}")
print(f"in the second chunk: the text length is {len(chunks[1].page_content)}")
```

```
Number of pages=19
Number of chunks=57
in the second page: the text length is 4564
in the second chunk: the text length is 1364
```

[46]:
```
chunks[0].page_content[-200:]
```

[46]:
```
'and the non-parametric memory is a dense\nvector index of Wikipedia, accessed
with a pre-trained neural retriever. We com-\npare two RAG formulations, one
which conditions on the same retrieved passages'
```

[45]:
```
chunks[1].page_content[:200]
```

[45]:
```
'pare two RAG formulations, one which conditions on the same retrieved
passages\nacross the whole generated sequence, and another which can use
different passages\nper token. We  ne-tune and evaluate our'
```

[27]:
```
pages[0].page_content[len(chunks[0].page_content)-300:len(chunks[1].
↪page_content)+300]
```

[27]:
```
'uage generation. We introduce RAG models where the parametric\nmemory is a pre-
trained seq2seq model and the non-parametric memory is a dense\nvector index of
Wikipedia, accessed with a pre-trained neural retriever. We com-\npare two RAG
formulations, one which conditions on the same retrieved passages\nacross the
whole generated sequence, and another which can use different passages\nper
token. We  ne-tune and evaluate our models on a wide range of
knowledge-\nintensive NLP tasks and set th'
```

0.3 Token Splitting

Using token splitting can be useful because LLMs often have context windows designated in tokens.

TokenTextSplitter is using Tiktoken library form OpenAI.

In LangChain documentation: https://api.python.langchain.com/en/latest/_modules/langchain/text_splitter.htr

- "gpt2" model is used per defalut

```
> [docs]class TokenTextSplitter(TextSplitter):
    """Splitting text to tokens using model tokenizer."""

> [docs]    def __init__(
        self,
        encoding_name: str = "gpt2",
        model_name: Optional[str] = None,
        allowed_special: Union[Literal["all"], AbstractSet[str]] = set(),
        disallowed_special: Union[Literal["all"], Collection[str]] = "all",
        **kwargs: Any,
```

- The encodings available:

```
ENCODING_CONSTRUCTORS is a list of string : ['gpt2', 'r50k_base', 'p50k_base', 'cl100k_base']
```

'cl100k_base' is the one used for 'gpt-3.5-turbo'.

To know which encoding is used for which model (openai models): use the tiktoken lib like this:

```
model = 'gpt-3.5-turbo'
tiktoken.encoding_for_model(model)
```

result ==> <Encoding 'cl100k_base'>

[ ]:
```
!pip install tiktoken
```

[48]:
```
import tiktoken
from langchain.text_splitter import TokenTextSplitter
```

0.3.1 Toy Examples

[49]:
```
# Examples
token_splitter = TokenTextSplitter(chunk_size=1, chunk_overlap=0) #using gpt2␣
 ↪perf default
text = "foo bar bazzyfoo"
print("token splitter with gpt2", token_splitter.split_text(text))

token_splitter = TokenTextSplitter(chunk_size=1, chunk_overlap=0,␣
 ↪encoding_name='cl100k_base')
text = "foo bar bazzyfoo"
print("token splitter with cl100k_base", token_splitter.split_text(text))
```

```
token splitter with gpt2 ['foo', ' bar', ' b', 'az', 'zy', 'foo']
token splitter with cl100k_base ['foo', ' bar', ' baz', 'zy', 'foo']
```

0.3.2 Chunking the PDF

[51]:
```
t_splitter = TokenTextSplitter(
    chunk_size=800,
    chunk_overlap=0,
    encoding_name='cl100k_base')

chunks = t_splitter.split_documents(pages)
```

[ ]:
```
chunks[:2]
```

[53]:
```
print(f"Number of pages={len(pages)}")
print(f"Number of chunks={len(chunks)}")
print(f"in the second page: the text length is {len(pages[1].page_content)}")
print(f"in the second chunk: the text length is {len(chunks[1].page_content)}")
```

```
Number of pages=19
Number of chunks=34
in the second page: the text length is 4564
in the second chunk: the text length is 3177
```

0.3.3 Number of tokens in each chunks

[54]:
```
encoding = tiktoken.get_encoding("cl100k_base")
for i in range(0, len(chunks)):
  tokens_integer=encoding.encode(chunks[i].page_content)
  print(f"{len(tokens_integer)} is the number of tokens in chunk {i}")
```

```
690 is the number of tokens in chunk 0
800 is the number of tokens in chunk 1
341 is the number of tokens in chunk 2
800 is the number of tokens in chunk 3
136 is the number of tokens in chunk 4
800 is the number of tokens in chunk 5
268 is the number of tokens in chunk 6
800 is the number of tokens in chunk 7
237 is the number of tokens in chunk 8
800 is the number of tokens in chunk 9
401 is the number of tokens in chunk 10
800 is the number of tokens in chunk 11
281 is the number of tokens in chunk 12
800 is the number of tokens in chunk 13
296 is the number of tokens in chunk 14
800 is the number of tokens in chunk 15
80 is the number of tokens in chunk 16
```

```
800 is the number of tokens in chunk 17
195 is the number of tokens in chunk 18
800 is the number of tokens in chunk 19
357 is the number of tokens in chunk 20
800 is the number of tokens in chunk 21
401 is the number of tokens in chunk 22
800 is the number of tokens in chunk 23
383 is the number of tokens in chunk 24
800 is the number of tokens in chunk 25
409 is the number of tokens in chunk 26
800 is the number of tokens in chunk 27
423 is the number of tokens in chunk 28
355 is the number of tokens in chunk 29
576 is the number of tokens in chunk 30
800 is the number of tokens in chunk 31
145 is the number of tokens in chunk 32
478 is the number of tokens in chunk 33
```

0.4 Context Aware Splitter

Here we will practice ==> HTMLHeaderTextSplitter.

With this type of splitter, you can respect and keep the structure of the website in your chunking, which make the retrieval more accurate

[55]:
```
from langchain.text_splitter import HTMLHeaderTextSplitter
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

[61]:
```
url = "https://realpython.github.io/fake-jobs/jobs/senior-python-developer-0.
 ↪html"

headers_to_split_on = [
    ("h1", "Header 1"),
    ("h2", "Header 2"),
    ("h3", "Header 3"),
    ("h4", "Header 4"),
]

html_splitter = HTMLHeaderTextSplitter(headers_to_split_on=headers_to_split_on)

#for local file use html_splitter.split_text_from_file(<path_to_file>)
#for html string use html_splitter.split_text(<HTML text string>)
html_header_splits = html_splitter.split_text_from_url(url)
html_header_splits
```

[61]:
```
[Document(page_content='Fake Python'),
 Document(page_content='Fake Jobs for Your Web Scraping Journey',
metadata={'Header 1': 'Fake Python'}),
```

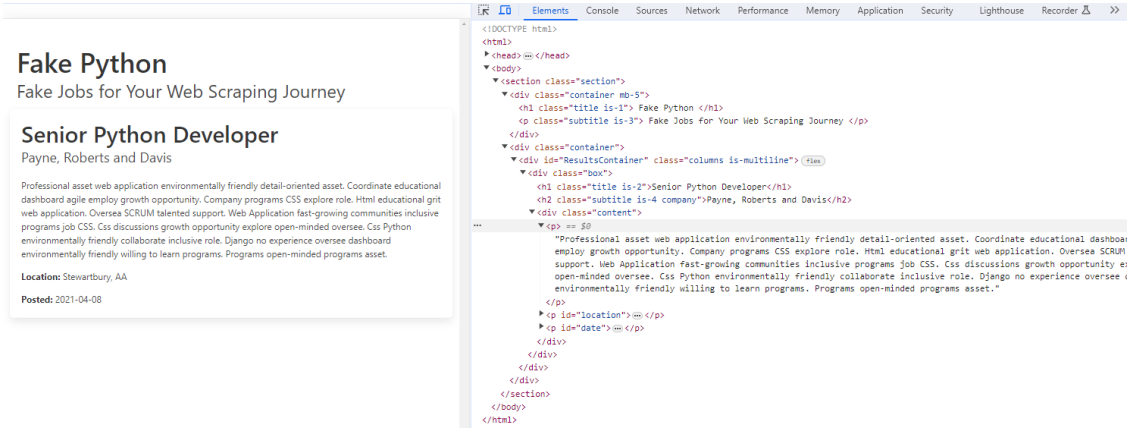
```
 Document(page_content='Senior Python Developer Payne, Roberts and Davis'),
 Document(page_content='Professional asset web application environmentally
friendly detail-oriented asset. Coordinate educational dashboard agile employ
growth opportunity. Company programs CSS explore role. Html educational grit web
application. Oversea SCRUM talented support. Web Application fast-growing
communities inclusive programs job CSS. Css discussions growth opportunity
explore open-minded oversee. Css Python environmentally friendly collaborate
inclusive role. Django no experience oversee dashboard environmentally friendly
willing to learn programs. Programs open-minded programs asset.  \nLocation:
Stewartbury, AA  \nPosted: 2021-04-08', metadata={'Header 1': 'Senior Python
Developer', 'Header 2': 'Payne, Roberts and Davis'})]
```

When it's a paragraph, you get a metadata with the structure of the headers: h1, h2…

When there are headers, you don't have this structure, so you know that's you're splitting headers (h1, h2,…)

[68]:
```
from IPython import display
display.Image(path_image)
```

[68]:



Then we'll split the results into documents, with the usual method of character splitting:

[63]:
```
chunk_size = 300
chunk_overlap = 0

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=chunk_size,
    chunk_overlap=chunk_overlap,
    separators=['\n\n','. ']
)

splits = text_splitter.split_documents(html_header_splits)
splits
```

[63]:
```
[Document(page_content='Fake Python'),
 Document(page_content='Fake Jobs for Your Web Scraping Journey',
metadata={'Header 1': 'Fake Python'}),
 Document(page_content='Senior Python Developer Payne, Roberts and Davis'),
 Document(page_content='Professional asset web application environmentally
friendly detail-oriented asset. Coordinate educational dashboard agile employ
growth opportunity. Company programs CSS explore role. Html educational grit web
application. Oversea SCRUM talented support', metadata={'Header 1': 'Senior
Python Developer', 'Header 2': 'Payne, Roberts and Davis'}),
 Document(page_content='. Web Application fast-growing communities inclusive
programs job CSS. Css discussions growth opportunity explore open-minded
oversee. Css Python environmentally friendly collaborate inclusive role. Django
no experience oversee dashboard environmentally friendly willing to learn
programs', metadata={'Header 1': 'Senior Python Developer', 'Header 2': 'Payne,
Roberts and Davis'}),
 Document(page_content='. Programs open-minded programs asset.  \nLocation:
Stewartbury, AA  \nPosted: 2021-04-08', metadata={'Header 1': 'Senior Python
Developer', 'Header 2': 'Payne, Roberts and Davis'})]
```

[66]:
```
for i, doc in enumerate(splits):
  if doc.metadata:
    if "Header 1" in doc.metadata.keys() and "Header 2" in doc.metadata.keys():
      value_h1 = doc.metadata['Header 1']
      value_h2 = doc.metadata['Header 2']

      print(f"{value_h1}\n\n {value_h2}\n\n {doc.page_content}")
```

```
Senior Python Developer

 Payne, Roberts and Davis

 Professional asset web application environmentally friendly detail-oriented
asset. Coordinate educational dashboard agile employ growth opportunity. Company
programs CSS explore role. Html educational grit web application. Oversea SCRUM
talented support
Senior Python Developer

 Payne, Roberts and Davis

 . Web Application fast-growing communities inclusive programs job CSS. Css
discussions growth opportunity explore open-minded oversee. Css Python
environmentally friendly collaborate inclusive role. Django no experience
oversee dashboard environmentally friendly willing to learn programs
Senior Python Developer

 Payne, Roberts and Davis
```

```
 . Programs open-minded programs asset.
Location: Stewartbury, AA
Posted: 2021-04-08
```

1 Vectorstores and Embedding: Semantic Search

1.1 Embeddings

Either you use Embedding from LangChain or OpenAI

[ ]:
```
!pip install openai
```

1.1.1 LangChain

[69]:
```
from google.colab import userdata
openai_api_key = userdata.get('OPENAI_API_KEY')
```

[72]:
```
from langchain.embeddings.openai import OpenAIEmbeddings
embedding = OpenAIEmbeddings(openai_api_key=openai_api_key)
# the model used under the hood : 'text-embedding-ada-002'
# https://api.python.langchain.com/en/latest/embeddings/langchain.embeddings.
 ↪openai.OpenAIEmbeddings.html
```

[73]:
```
text = """
Pre-\ntrained models with a differentiable access mechanism to explicit␣
 ↪non-parametric\nmemory have so far been only investigated for extractive␣
 ↪downstream tasks.

We\nexplore a general-purpose  ne-tuning recipe for retrieval-augmented␣
 ↪generation\n(RAG) - models which combine pre-trained parametric and␣
 ↪non-parametric mem-\nory for language generation.
We introduce RAG models where the parametric\nmemory is a pre-trained seq2seq␣
 ↪model and the non-parametric memory is a dense\nvector index of Wikipedia,␣
 ↪accessed with a pre-trained neural retriever.
"""
```

[82]:
```
embedding_langchain = embedding.embed_query(text)
print(f"length of embedding vector {len(embedding_langchain)}") #this length is␣
 ↪specific to openai embedding model
print(embedding_langchain[:10])
```

```
length of embedding vector 1536
[-0.02020188083639923, 0.010144510482966114, 0.024747666402209146,
-0.012903932706575449, 0.00287379376003672, 0.019403100351411295,
0.005649555572840759, -0.0011273694886516647, -0.045806419901521525,
-0.04025852750269507]
```

1.1.2 Using Embedding from OpenAI directly

[88]:
```
from openai import OpenAI
client = OpenAI(api_key = openai_api_key)
```

[92]:
```
def embeddings(inpt_text: str,
               model_name: str = 'text-embedding-ada-002'
               ) -> list[float]:

  response = client.embeddings.create(
        model=model_name,
        input=inpt_text
  )
  return response.data[0].embedding

embedding_openai = embeddings(text)
print(f"length of embedding vector {len(embedding_openai)}") #this length is␣
 ↪specific to openai embedding model
print(embedding_openai[:10])
```

```
length of embedding vector 1536
[-0.020201880484819412, 0.010144510306417942, 0.024747665971517563,
-0.012903932482004166, 0.0028737937100231647, 0.01940310001373291,
0.00564955547451973, -0.0011273694690316916, -0.04580641910433769,
-0.040258526802062991]
```

[94]:
```
import numpy as np
```

[96]:
```
print(np.dot(embedding_langchain, embedding_openai))
```

```
0.999999982596679
```

Well, not surprising, the same results

1.2 Vectorstores: Chroma

LangChain has integration with different vectorstores, over 30. Chorma is chosen because it's lightweight in memory, which makes it very easy to get up and start with. There are other vectorstores that offer hosted solutions, which can be useful when you're trying to persist large amounts of data or persist it in cloud storage somewhere.

[ ]:
```
! pip install chromadb
```

Let's resume here: I have my pdfs chunked (from "Chunking the pdf" paragraph), I have a vectorstore "chroma" to store my embedding vectors, coming from neural embedding model 'text-embedding-ada-002'.

[99]:
```
from langchain.vectorstores import Chroma
from langchain.embeddings.openai import OpenAIEmbeddings
```

```
embedding = OpenAIEmbeddings(openai_api_key=openai_api_key)
```

[117]:
```
persist_directory = '/content/drive/MyDrive/02-Articles_ChatGPT/03_notebooks/
  ↪data/chroma2/'
!rm -rf persist_directory # remove old database files if any
```

[ ]:
```
# chunks[:3]
```

[118]:
```
vectordb = Chroma.from_documents(
    documents=chunks,
    embedding=embedding,
    persist_directory=persist_directory
)
```

[112]:
```
print(vectordb._collection.count())
```

```
177
```

1.2.1 Similarity Search

Similarity_search

[123]:
```
question = "What is RAG?"
docs = vectordb.similarity_search(question, k=5)

print(len(docs))

for doc in docs:
  print(doc.page_content[:500])
  print(doc.metadata)
```

```
5
Broader Impact
This work offers several positive societal bene ts over previous work: the fact
that it is more
strongly grounded in real factual knowledge (in this case Wikipedia) makes it
"hallucinate" less
with generations that are more factual, and offers more control and
interpretability. RAG could be
employed in a wide variety of scenarios with direct bene t to society, for
example by endowing it
with a medical index and asking it open-domain questions on that topic, or by
helping people be
{'page': 9, 'source': 'https://arxiv.org/pdf/2005.11401.pdf'}
for knowledge-intensive tasks [ 20]. We treat questions and answers as input-
output text pairs (x,y)
and train RAG by directly minimizing the negative log-likelihood of answers. We
compare RAG to
```

the popular extractive QA paradigm [5,7,31,26], where answers are extracted spans from retrieved
documents, relying primarily on non-parametric knowledge. We also compare to "Closed-Book
QA" approaches [52], which, like RAG, generate answers, but which do not exploit retrieval, instead
relying purely
{'page': 3, 'source': 'https://arxiv.org/pdf/2005.11401.pdf'}
Jeopardy
Question
Gener
-ationWashingtonBART?This state has the largest number of counties in the U.S.
RAG-T It's the only U.S. state named for a U.S. president
RAG-S It's the state where you'll nd Mount Rainier National Park
The Divine
ComedyBART*This epic poem by Dante is divided into 3 parts: the Inferno, the Purgatorio & the Purgatorio
RAG-T Dante's "Inferno" is the rst part of this epic poem
RAG-S This 14th century work is divided into 3 sections: "Inferno", "Purgatorio" & "Paradiso"
For
{'page': 6, 'source': 'https://arxiv.org/pdf/2005.11401.pdf'}
Machine Translation [18,22] and Semantic Parsing [21]. Our approach does have several differences,
including less of emphasis on lightly editing a retrieved item, but on aggregating content from several
pieces of retrieved content, as well as learning latent retrieval, and retrieving evidence documents
rather than related training pairs. This said, RAG techniques may work well in these settings, and
could represent promising future work.
6 Discussion
In this work, we presented hybrid generatio
{'page': 8, 'source': 'https://arxiv.org/pdf/2005.11401.pdf'}
a general-purpose ne-tuning approach which we refer to as retrieval-augmented generation (RAG).
We build RAG models where the parametric memory is a pre-trained seq2seq transformer, and the
non-parametric memory is a dense vector index of Wikipedia, accessed with a pre-trained neural
retriever. We combine these components in a probabilistic model trained end-to-end (Fig. 1). The
retriever (Dense Passage Retriever [26], henceforth DPR) provides latent documents conditioned on
the input, and the
{'page': 1, 'source': 'https://arxiv.org/pdf/2005.11401.pdf'}

Similarity_search_with_relevance_scores

```
?vectordb.similarity_search_with_relevance_scores
###### Docstring:
###### Return docs and relevance scores in the range [0, 1].

###### 0 is dissimilar, 1 is most similar.
Returns:
    List of Tuples of (doc, similarity_score)
```

==> similarity_score : 1-cosine() higher score represents more similarity.

[125]:
```
question = "What is RAG?"
docs = vectordb.similarity_search_with_relevance_scores(question, k=5)
for i, doc in enumerate(docs):
  print(f"doc number = {i}, score = {doc[1]}")
```

```
doc number = 0, score = 0.765787272186013
doc number = 1, score = 0.7597936166770761
doc number = 2, score = 0.756966304006166
doc number = 3, score = 0.7551403893319466
doc number = 4, score = 0.7379775376749862
```

similarity_search_with_score

```
?vectordb.similarity_search_with_score
Lower score represents more similarity.
score = cosine distance
```

[126]:
```
question = "What is RAG?"
docs = vectordb.similarity_search_with_score(question, k=5)
for i, doc in enumerate(docs):
  print(f"doc number = {i}, score = {doc[1]}")
```

```
doc number = 0, score = 0.33122681615493876
doc number = 1, score = 0.33970312506386946
doc number = 2, score = 0.3437015489881398
doc number = 3, score = 0.34628378228415696
doc number = 4, score = 0.3705557198664279
```