5 Levels Of Summarization: Novice to Expert

Summarization is a fundamental building block of many LLM tasks. You'll frequently run into use cases where you would like to distill a large body of text into a succinct set of points.

Depending on the length of the text you'd like to summarize, you have different summarization methods to choose from.

We're going to run through 5 methods for summarization that start with Novice and end up expert. These aren't the only options, feel free to make up your own. If you find another one you like please share it with the community.

5 Levels Of Summarization:

1. **Summarize a couple sentences** - Basic Prompt
2. **Summarize a couple paragraphs** - Prompt Templates
3. **Summarize a couple pages** - Map Reduce
4. **Summarize an entire book** - Best Representation Vectors
5. **Summarize an unknown amount of text** - Agents

First let's import our OpenAI API Key

```
from dotenv import load_dotenv
import os

load_dotenv()

openai_api_key = os.getenv('OPENAI_API_KEY', 'YourAPIKey')
```

Level 1: Basic Prompt - Summarize a couple sentences

If you just have a few sentences you want to one-off summarize you can use a simple prompt and copy and paste your text.

This method isn't scalable and only practical for a few use cases...the perfect level #1!

```
from langchain import OpenAI

llm = OpenAI(temperature=0, openai_api_key=openai_api_key)
```

The important part is to provide instructions for the LLM to know what to do. In thise case I'm telling the model I want a summary of the text below

```
prompt = """
Please provide a summary of the following text
```

```
TEXT:
Philosophy (from Greek: φιλοσοφία, philosophia, 'love of wisdom') \
is the systematized study of general and fundamental questions, \
such as those about existence, reason, knowledge, values, mind, and language.
\
Some sources claim the term was coined by Pythagoras (c. 570 – c. 495 BCE), \
although this theory is disputed by some. Philosophical methods include
questioning, \
critical discussion, rational argument, and systematic presentation.
"""

num_tokens = llm.get_num_tokens(prompt)
print (f"Our prompt has {num_tokens} tokens")

Our prompt has 121 tokens

output = llm(prompt)
print (output)


Philosophy is a systematized study of general and fundamental questions about
existence, reason, knowledge, values, mind, and language. It is believed to
have been coined by Pythagoras, and its methods include questioning, critical
discussion, rational argument, and systematic presentation.
```

Woof 🐶, that summary is still hard to understand. Let me add to my instructions so that the output is easier to understand. I'll tell it to explain it to me like a 5 year old.

```
prompt = """
Please provide a summary of the following text.
Please provide your output in a manner that a 5 year old would understand

TEXT:
Philosophy (from Greek: φιλοσοφία, philosophia, 'love of wisdom') \
is the systematized study of general and fundamental questions, \
such as those about existence, reason, knowledge, values, mind, and language.
\
Some sources claim the term was coined by Pythagoras (c. 570 – c. 495 BCE), \
although this theory is disputed by some. Philosophical methods include
questioning, \
critical discussion, rational argument, and systematic presentation.
"""

num_tokens = llm.get_num_tokens(prompt)
print (f"Our prompt has {num_tokens} tokens")

Our prompt has 137 tokens

output = llm(prompt)
print (output)
```

```
Philosophy is about asking questions and trying to figure out the answers. It
is about thinking about things like existence, knowledge, and values. People
have been doing this for a very long time, and it is still done today.
```

Nice! That's much better, but let's look at something we can automate a bit more

Level 2: Prompt Templates - Summarize a couple paragraphs

Prompt templates are a great way to dynamically place text within your prompts. They are like python f-strings but specialized for working with language models.

We're going to look at 2 short Paul Graham essays

```
from langchain import OpenAI
from langchain import PromptTemplate
import os

paul_graham_essays = ['../data/PaulGrahamEssaySmall/getideas.txt',
'../data/PaulGrahamEssaySmall/noob.txt']

essays = []

for file_name in paul_graham_essays:
    with open(file_name, 'r') as file:
        essays.append(file.read())
```

Let's print out a preview of the essays to see what they look like

```
for i, essay in enumerate(essays):
    print (f"Essay #{i+1}: {essay[:300]}\n")
```

```
Essay #1: January 2023(Someone fed my essays into GPT to make something that
could answer
questions based on them, then asked it where good ideas come from.  The
answer was ok, but not what I would have said. This is what I would have
said.)The way to get new ideas is to notice anomalies: what seems strange,


Essay #2: January 2020When I was young, I thought old people had everything
figured out.
Now that I'm old, I know this isn't true.I constantly feel like a noob. It
seems like I'm always talking to
some startup working in a new field I know nothing about, or reading
a book about a topic I don't understand well
```

Next let's create a prompt template which will hold our instructions and a placeholder for the essay. In this example I only want a 1 sentence summary to come back

```
template = """
Please write a one sentence summary of the following text:

{essay}
"""

prompt = PromptTemplate(
    input_variables=["essay"],
    template=template
)
```

Then let's loop through the 2 essays and pass them to our LLM. I'm applying .strip() on the summaries to remove the white space on the front and back of the output

```
for essay in essays:
    summary_prompt = prompt.format(essay=essay)

    num_tokens = llm.get_num_tokens(summary_prompt)
    print (f"This prompt + essay has {num_tokens} tokens")

    summary = llm(summary_prompt)

    print (f"Summary: {summary.strip()}")
    print ("\n")
```

```
This prompt + essay has 205 tokens
Summary: Exploring anomalies at the frontiers of knowledge is the best way to
generate new ideas.


This prompt + essay has 500 tokens
Summary: This text explores the idea that feeling like a "noob" is actually
beneficial, as it is inversely correlated with actual ignorance and
encourages us to discover new things.
```

Level 3: Map Reduce - Summarize a couple pages multiple pages

If you have multiple pages you'd like to summarize, you'll likely run into a token limit. Token limits won't always be a problem, but it is good to know how to handle them if you run into the issue.

The chain type "Map Reduce" is a method that helps with this. You first generate a summary of smaller chunks (that fit within the token limit) and then you get a summary of the summaries. Check out this video for more information on how chain types work

```
from langchain import OpenAI
from langchain.chains.summarize import load_summarize_chain
from langchain.text_splitter import RecursiveCharacterTextSplitter

paul_graham_essay = '../data/PaulGrahamEssays/startupideas.txt'

with open(paul_graham_essay, 'r') as file:
    essay = file.read()
```

Let's see how many tokens are in this essay

```
llm.get_num_tokens(essay)
```

```
9565
```

That's too many, let's split our text up into chunks so they fit into the prompt limit. I'm going a chunk size of 10,000 characters.

> You can think of tokens as pieces of words used for natural language processing. For English text, **1 token is approximately 4 characters** or 0.75 words. As a point of reference, the collected works of Shakespeare are about 900,000 words or 1.2M tokens.

This means the number of tokens we should expect is 10,000 / 4 = ~2,500 token chunks. But this will vary, each body of text/code will be different

```
text_splitter = RecursiveCharacterTextSplitter(separators=["\n\n", "\n"],
chunk_size=10000, chunk_overlap=500)

docs = text_splitter.create_documents([essay])

num_docs = len(docs)

num_tokens_first_doc = llm.get_num_tokens(docs[0].page_content)

print (f"Now we have {num_docs} documents and the first one has
{num_tokens_first_doc} tokens")
```

```
Now we have 5 documents and the first one has 2086 tokens
```

Great, assuming that number of tokens is consistent in the other docs we should be good to go. Let's use LangChain's load_summarize_chain to do the `map_reducing` for us. We first need to initialize our chain

```
summary_chain = load_summarize_chain(llm=llm, chain_type='map_reduce',
#                                     verbose=True # Set verbose=True if you
want to see the prompts being used
                                    )
```

Now actually run it

```
output = summary_chain.run(docs)

output

' This article provides strategies for coming up with startup ideas on
demand, such as looking in areas of expertise, talking to people about their
needs, and looking for waves and gaps in the market. It also discusses the
need for users to have sufficient activation energy to start using a product,
and how this varies depending on the product. It looks at the difficulty of
switching paths in life as one gets older, and how colleges can help students
start startups. Finally, it looks at the importance of focusing on users
rather than competitors, and how Steve Wozniak solved his own problems.'
```

This summary is a great start, but I'm more of a bullet point person. I want to get my final output in bullet point form.

In order to do this I'm going to use custom promopts (like we did above) to instruct the model on what I want.

The map_prompt is going to stay the same (just showing it for clarity), but I'll edit the combine_prompt.

```
map_prompt = """
Write a concise summary of the following:
"{text}"
CONCISE SUMMARY:
"""
map_prompt_template = PromptTemplate(template=map_prompt,
input_variables=["text"])

combine_prompt = """
Write a concise summary of the following text delimited by triple backquotes.
Return your response in bullet points which covers the key points of the
text.
```{text}```
BULLET POINT SUMMARY:
"""
combine_prompt_template = PromptTemplate(template=combine_prompt,
input_variables=["text"])

summary_chain = load_summarize_chain(llm=llm,
                                     chain_type='map_reduce',
                                     map_prompt=map_prompt_template,
                                     combine_prompt=combine_prompt_template,
#                                      verbose=True
                                    )

output = summary_chain.run(docs)

print (output)
```
```

```
- Y Combinator suggests that the best startup ideas come from looking for
problems, preferably ones that the founders have themselves.
- Good ideas should appeal to a small number of people who need it urgently.
- To find startup ideas, one should look for things that seem to be missing
and be prepared to question the status quo.
- College students should use their college experience to prepare themselves
for the future and build things with other students.
- Tricks for coming up with startup ideas on demand include looking in areas
of expertise, talking to people about their needs, and looking for waves and
gaps in the market.
- Sam Altman points out that taking the time to come up with an idea is a
better strategy than most founders are willing to put in the time for.
- Paul Buchheit suggests that trying to sell something bad can lead to better
ideas.
```

Level 4: Best Representation Vectors - Summarize an entire book

In the above method we pass the entire document (all 9.5K tokens of it) to the LLM. But what if you have more tokens than that?

What if you had a book you wanted to summarize? Let's load one up, we're going to load Into Thin Air about the 1996 Everest Disaster

```
from langchain.document_loaders import PyPDFLoader

# Load the book
loader = PyPDFLoader("../data/IntoThinAirBook.pdf")
pages = loader.load()

# Cut out the open and closing parts
pages = pages[26:277]

# Combine the pages, and replace the tabs with spaces
text = ""

for page in pages:
    text += page.page_content

text = text.replace('\t', ' ')

num_tokens = llm.get_num_tokens(text)

print (f"This book has {num_tokens} tokens in it")
```

```
This book has 139472 tokens in it
```

Wow, that's over 100K tokens, even GPT 32K wouldn't be able to handle that in one go. At 0.03 per 1K prompt tokens, this would cost us $4.17 just for the prompt alone.

So how do we do this without going through all the tokens? Pick random chunks? Pick equally spaced chunks?

I kicked off a twitter thread with a proposed solution to see if I was off base. I'm calling it the Best Representation Vectors method (not sure if a name already exists for it).

Goal: Chunk your book then get embeddings of the chunks. Pick a subset of chunks which represent a wholistic but diverse view of the book. Or another way, is there a way to pick the top 10 passages that describe the book the best?

Once we have our chunks that represent the book then we can summarize those chunks and hopefully get a pretty good summary.

Keep in mind there are tools that would likely do this for you, and with token limits increasing this won't be a problem for long. But if you want to do it from scratch this might help.

This is most definitely not the optimal answer, but it's my take on it for now! If the clustering experts wanna help improve it that would be awesome.

The BRV Steps:

1. Load your book into a single text file
2. Split your text into large-ish chunks
3. Embed your chunks to get vectors
4. Cluster the vectors to see which are similar to each other and likely talk about the same parts of the book
5. Pick embeddings that represent the cluster the most (method: closest to each cluster centroid)
6. Summarize the documents that these embeddings represent

Another way to phrase this process, "Which ~10 documents from this book represent most of the meaning? I want to build a summary off those."

Note: There will be a bit of information loss, but show me a summary of a whole book that doesn't have information loss ;)

```
# Loaders
from langchain.schema import Document

# Splitters
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Model
from langchain.chat_models import ChatOpenAI

# Embedding Support
from langchain.vectorstores import FAISS
```

```
from langchain.embeddings import OpenAIEmbeddings

# Summarizer we'll use for Map Reduce
from langchain.chains.summarize import load_summarize_chain

# Data Science
import numpy as np
from sklearn.cluster import KMeans
```

```
/Users/gregorykamradt/opt/anaconda3/lib/python3.9/site-
packages/deeplake/util/check_latest_version.py:32: UserWarning: A newer
version of deeplake (3.7.2) is available. It's recommended that you update to
the latest version using `pip install -U deeplake`.
  warnings.warn(
```

I'm going to initialize two models, gpt-3.5 and gpt4. I'll use gpt 3.5 for the first set of summaries to reduce cost and then gpt4 for the final pass which should hopefully increase the quality.

```
text_splitter = RecursiveCharacterTextSplitter(separators=["\n\n", "\n",
"\t"], chunk_size=10000, chunk_overlap=3000)

docs = text_splitter.create_documents([text])

num_documents = len(docs)

print (f"Now our book is split up into {num_documents} documents")
```

```
Now our book is split up into 78 documents
```

Let's get our embeddings of those 78 documents

```
embeddings = OpenAIEmbeddings(openai_api_key=openai_api_key)

vectors = embeddings.embed_documents([x.page_content for x in docs])
```

Now let's cluster our embeddings. There are a ton of clustering algorithms you can chose from. Please try a few out to see what works best for you!

```
# Assuming 'embeddings' is a list or array of 1536-dimensional embeddings

# Choose the number of clusters, this can be adjusted based on the book's
content.
# I played around and found ~10 was the best.
# Usually if you have 10 passages from a book you can tell what it's about
num_clusters = 11

# Perform K-means clustering
kmeans = KMeans(n_clusters=num_clusters, random_state=42).fit(vectors)
```

Here are the clusters that were found. It's interesting to see the progression of clusters throughout the book. This is expected because as the plot changes you'd expect different clusters to emerge due to different semantic meaning

`kmeans.labels_`

This is sweet, but whenever you have a clustering exercise, it's hard *not* to graph them. Make sure you add colors.

We also need to do dimensionality reduction to reduce the vectors from 1536 dimensions to 2 (this is sloppy data science but we are working towards the 80% solution)

```
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

# Taking out the warnings
import warnings
from warnings import simplefilter

# Filter out FutureWarnings
simplefilter(action='ignore', category=FutureWarning)

# Perform t-SNE and reduce to 2 dimensions
tsne = TSNE(n_components=2, random_state=42)
reduced_data_tsne = tsne.fit_transform(vectors)

# Plot the reduced data
plt.scatter(reduced_data_tsne[:, 0], reduced_data_tsne[:, 1],
c=kmeans.labels_)
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.title('Book Embeddings Clustered')
plt.show()
```

Awesome, not perfect, but pretty good directionally. Now we need to get the vectors which are closest to the cluster centroids (the center).

The function below is a quick way to do that (w/ help from ChatGPT)

```
# Find the closest embeddings to the centroids

# Create an empty list that will hold your closest points
closest_indices = []

# Loop through the number of clusters you have
for i in range(num_clusters):

    # Get the list of distances from that particular cluster center
    distances = np.linalg.norm(vectors - kmeans.cluster_centers_[i], axis=1)
```

```
    # Find the list position of the closest one (using argmin to find the
smallest distance)
    closest_index = np.argmin(distances)

    # Append that position to your closest indices list
    closest_indices.append(closest_index)
```

Now sort them (so the chunks are processed in order)

```
selected_indices = sorted(closest_indices)
selected_indices
```

It's intersting to see which chunks pop up at most descriptive. How does your distribution look?

Let's create our custom prompts. I'm going to use gpt4 (which has a bigger token limit) for the combine step so I'm asking for long summaries in the map step to reduce the information loss.

```
llm3 = ChatOpenAI(temperature=0,
                  openai_api_key=openai_api_key,
                  max_tokens=1000,
                  model='gpt-3.5-turbo'
                 )

map_prompt = """
You will be given a single passage of a book. This section will be enclosed
in triple backticks (```)
Your goal is to give a summary of this section so that a reader will have a
full understanding of what happened.
Your response should be at least three paragraphs and fully encompass what
was said in the passage.

```{text}```
FULL SUMMARY:
"""
map_prompt_template = PromptTemplate(template=map_prompt,
input_variables=["text"])
```

I kept getting a timeout errors so I'm actually going to do this map reduce manually

```
map_chain = load_summarize_chain(llm=llm3,
                                 chain_type="stuff",
                                 prompt=map_prompt_template)
```

Then go get your docs which the top vectors represented.

```
selected_docs = [docs[doc] for doc in selected_indices]
```
```

Let's loop through our selected docs and get a good summary for each chunk. We'll store the summary in a list.

```
# Make an empty list to hold your summaries
summary_list = []

# Loop through a range of the lenght of your selected docs
for i, doc in enumerate(selected_docs):

    # Go get a summary of the chunk
    chunk_summary = map_chain.run([doc])

    # Append that summary to your list
    summary_list.append(chunk_summary)

    print (f"Summary #{i} (chunk #{selected_indices[i]}) - Preview:
{chunk_summary[:250]} \n")
```

Great, now that we have our list of summaries, let's get a summary of the summaries

```
summaries = "\n".join(summary_list)

# Convert it back to a document
summaries = Document(page_content=summaries)

print (f"Your total summary has {llm.get_num_tokens(summaries.page_content)}
tokens")
```

```
llm4 = ChatOpenAI(temperature=0,
                  openai_api_key=openai_api_key,
                  max_tokens=3000,
                  model='gpt-4',
                  request_timeout=120
                 )

combine_prompt = """
You will be given a series of summaries from a book. The summaries will be
enclosed in triple backticks (```)
Your goal is to give a verbose summary of what happened in the story.
The reader should be able to grasp what happened in the book.

```{text}```
VERBOSE SUMMARY:
"""
combine_prompt_template = PromptTemplate(template=combine_prompt,
input_variables=["text"])

reduce_chain = load_summarize_chain(llm=llm4,
                                    chain_type="stuff",
                                    prompt=combine_prompt_template,
```
```

```
#                                 verbose=True # Set this to true if you want to
see the inner workings
                                    )
```

Run! Note this will take a while

```
output = reduce_chain.run([summaries])

print (output)
```

Wow that was a long process, but you get the gist, hopefully we'll see some library abstractions in the coming months that do this automatically for us! Let me know what you think on Twitter

Level 5: Agents - Summarize an unknown amount of text

What if you have an unknown amount of text you need to summarize? This may be a verticalize use case (like law or medical) where more research is required as you uncover the first pieces of information.

We're going to use agents below, this is still a very actively developed area and should be handled with care. Future agents will be able to handle a lot more complicated tasks.

```
from langchain import OpenAI
from langchain.chat_models import ChatOpenAI
from langchain.agents import initialize_agent, Tool
from langchain.utilities import WikipediaAPIWrapper

llm = ChatOpenAI(temperature=0, model_name='gpt-4',
openai_api_key=openai_api_key)
```

We're going to use the Wiki search tool and research multiple topics

```
wikipedia = WikipediaAPIWrapper()
```

Let's define our toolkit, in this case it's just one tool

```
tools = [
    Tool(
        name="Wikipedia",
        func=wikipedia.run,
        description="Useful for when you need to get information from
wikipedia about a single topic"
    ),
]
```

Init our agent

```
agent_executor = initialize_agent(tools, llm, agent='zero-shot-react-
description', verbose=True)
```

Then let's ask a question that will need multiple documents

```
output = agent_executor.run("Can you please provide a quick summary of
Napoleon Bonaparte? \
                            Then do a separate search and tell me what the
commonalities are with Serena Williams")

print (output)
```

Awesome, good luck summarizing!