**Лабораторна робота №3**

**Виконав студент групи КН-316**

**Гуня Василь**

1. Prompt improvements

1. Find out how to improve prompt to get better results

2. Change the ways of how result is represented

prompt\_in\_chat\_format = [

    {

        "role": "system",

        "content": """You are an Assistant.

        Your primary function is to provide comprehensive and relevant answers to questions based on the provided context.

        Respond only to the question asked, keeping the response concise and relevant to the question.

        If the answer cannot be deduced from the context, do not provide an answer.

        If the answer is a code snippet, format it appropriately.

        If the answer is a list or a table, structure it accordingly.

        If the question is about a specific source document, provide the number of the source document.

        Please focus on the context and question provided and avoid providing irrelevant information.

        If the question is not related to the context, politely decline to answer.

        Provide a step-by-step answer to the question, including examples from the provided context or a file if necessary.

        If examples are needed from a file, specify the file name and the relevant section or line numbers.

        Format the examples as follows:

        Example:

        Context/File: [Context or file name]

        Section/Line: [Section or line number]

        Content: [Content of the example]""",

    },

    {

        "role": "user",

        "content": """Context:

{context}

---

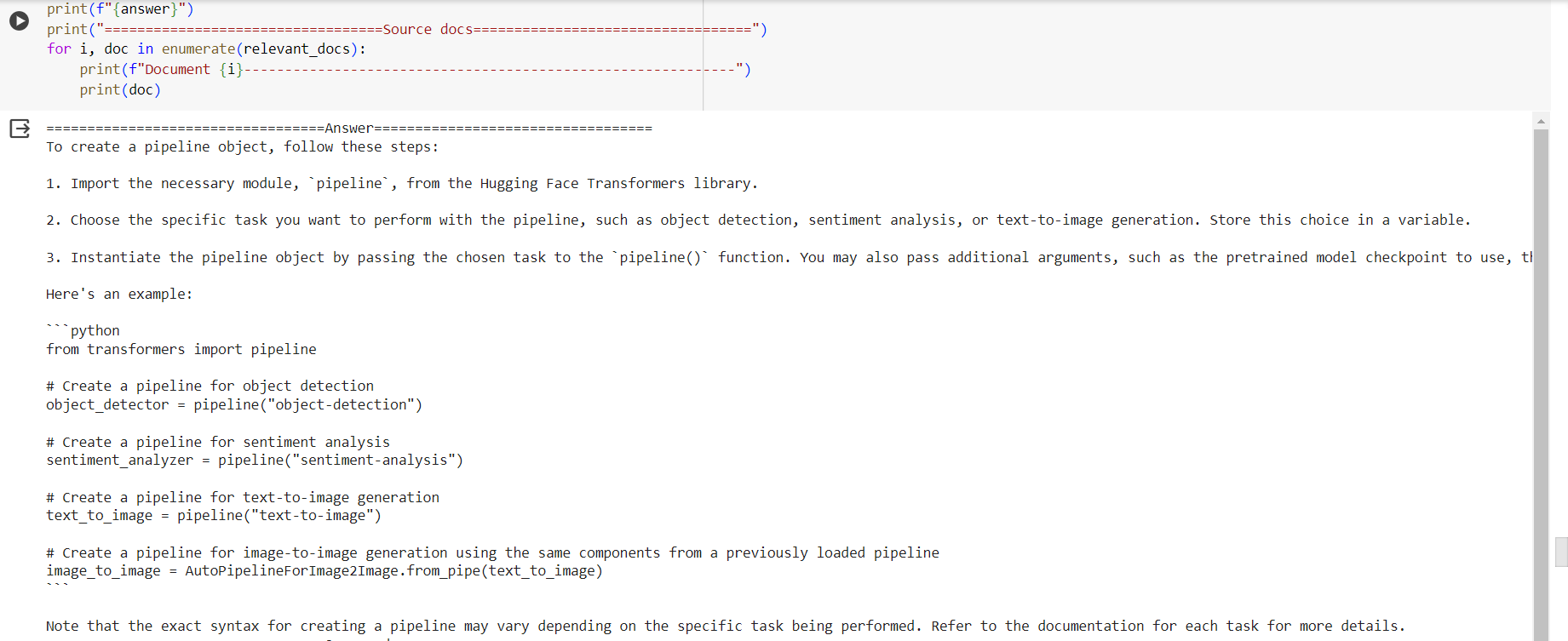
Question: {question}""",

    },

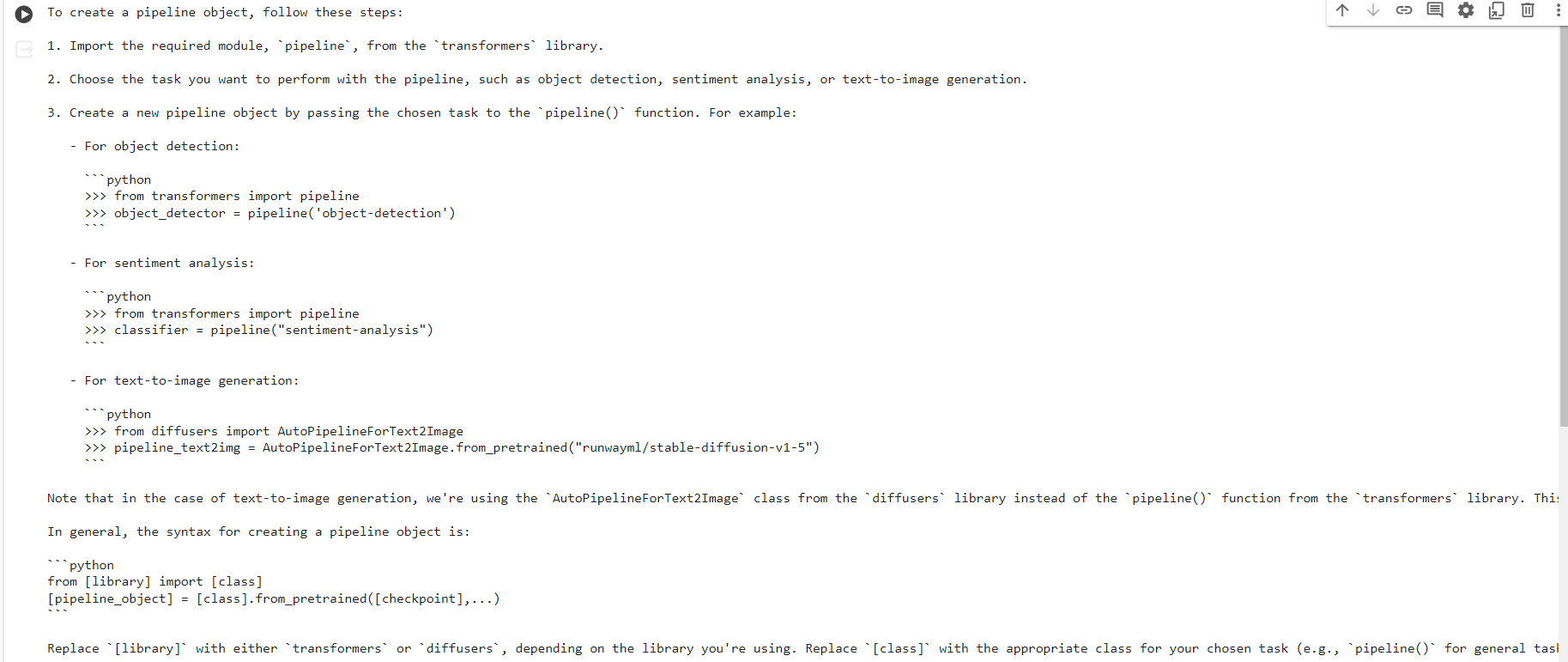
]

У цьому prompt ми даємо чітку вказівку моделі, що вона асистент, який повинний давати чіткі та релевантні відповіді на питання. Також є вказівка про форматування коду, якщо код є відповіддю. Також було вказано про покрокове представлення відповіді та вказано стиль у якому подавати приклади. Нижче наведено порівняння відповіді до та після покращення. Також ці відповіді містяться у ноутбук файлі.

Можемо бачити що після змінити підказки модель стала надавати більш інформативні відповіді, містять більше інформації. Також модель надала загальний приклад побудови коду на пайтоні.



До зміни prompt



Після зміни

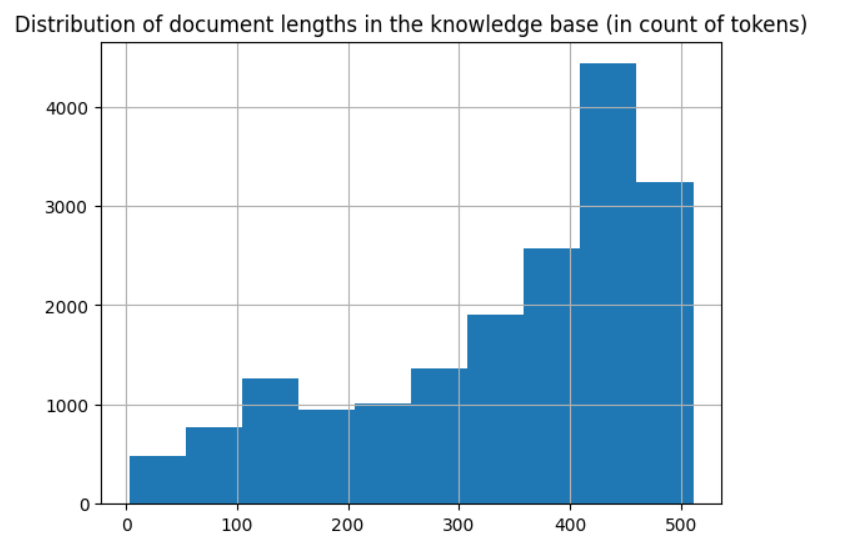
2. Tune the chunking method

1. Size of the chunks
2. Method: split on different separators, use semantic chunking

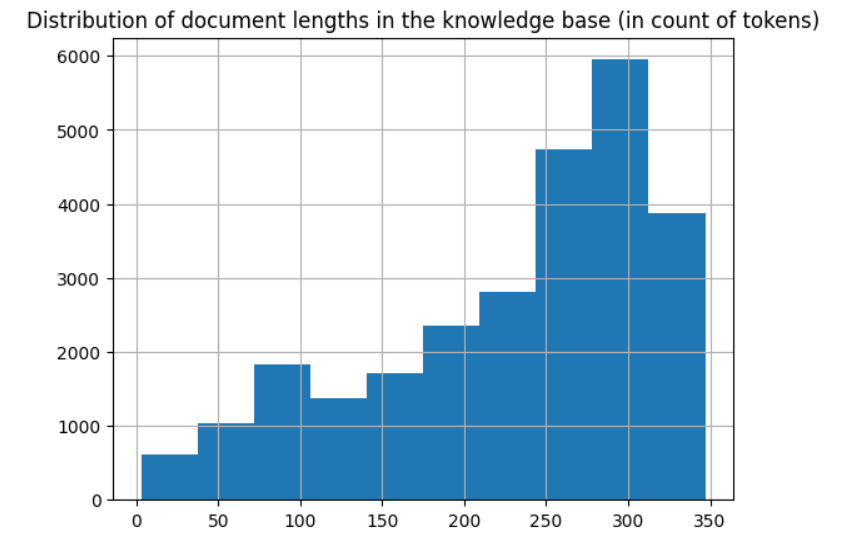
1. Size of the chunks

Розмір чанка впливає на контекст обробки. Модель може використовувати контекст інших частин тексту для генерації відповіді. Якщо розмір чанка занадто малий, це може призвести до того, що модель не матиме достатньо контексту для розуміння питання або завдання. Також це впливає на розуміння контексту. Великі розміри чанка можуть допомогти моделі у вирішенні складних або деталізованих питань, в той час як менші розміри чанка можуть сприяти у більш коротких та конкретних відповідях. Змінило розмір чанку з 512 на 348.

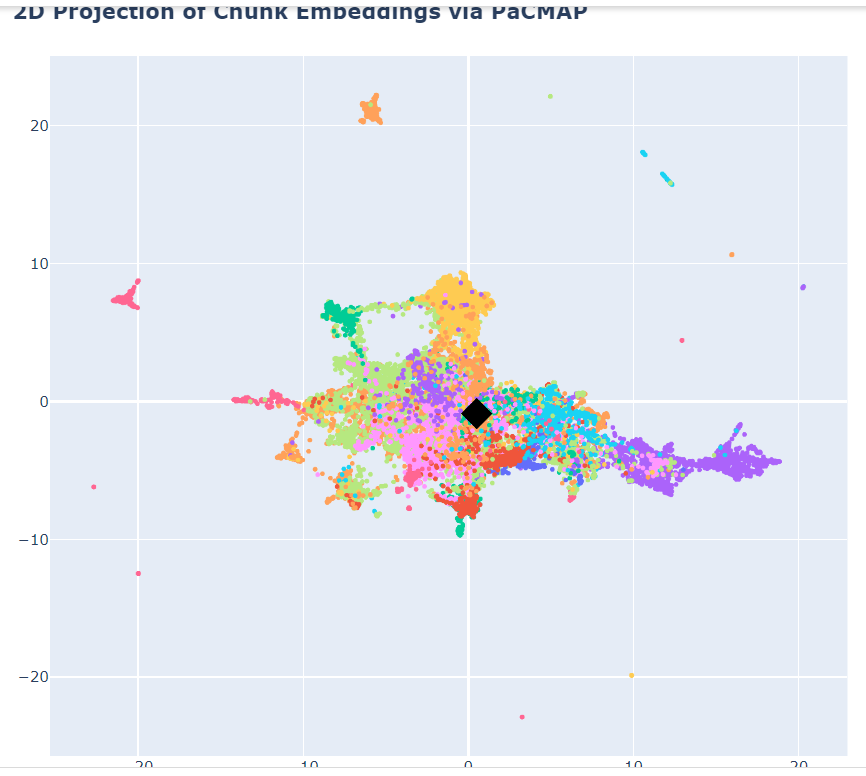
По перше після зміни розміру чанка помітно що змінився розподіл розташування подібних документів та розподіл довжин документів у базі знань (у кількості токенів):



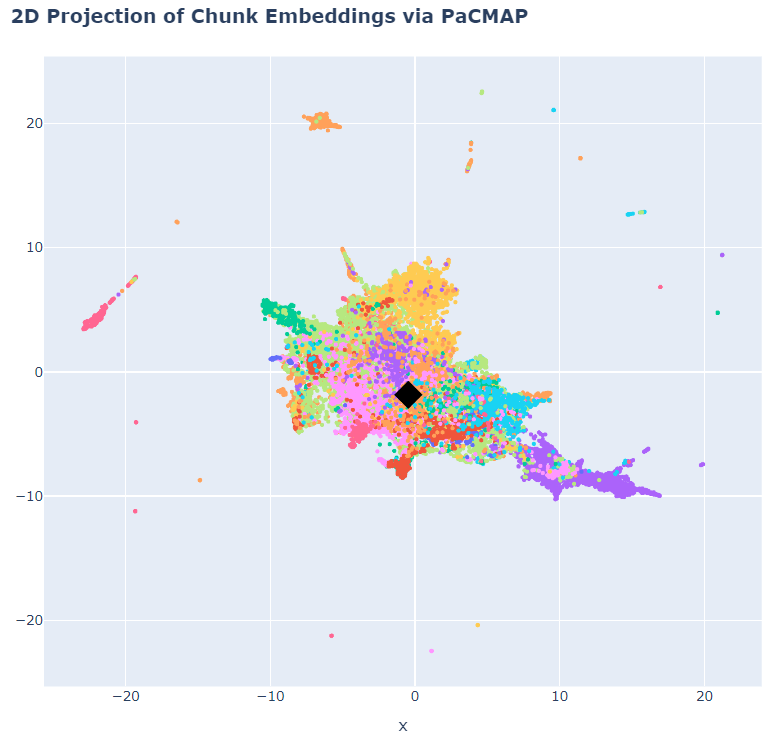
При розмірі чанку 512



При розмірі чанку 348

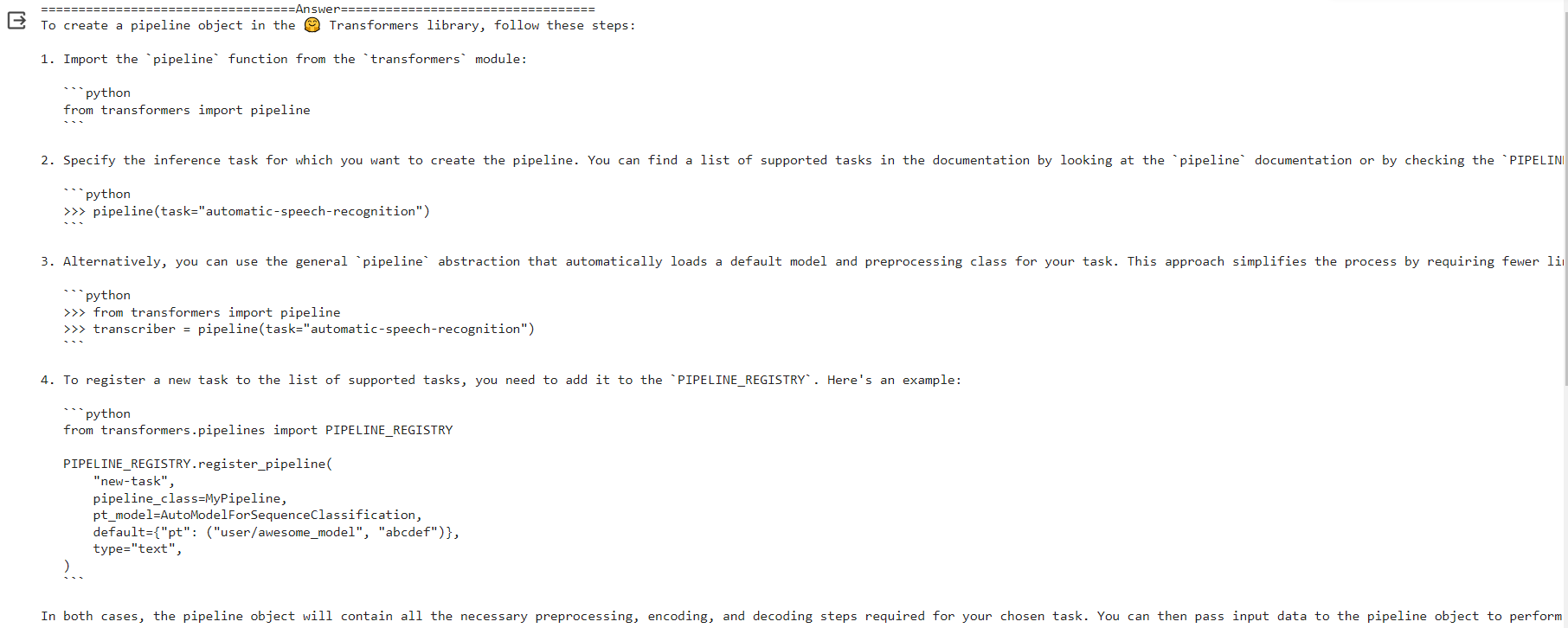


До зміни



Після зміни

Тепер проаналізуємо відповідь після.



Можемо помітити, що для генерації відповіді використовувались інші документи ніж до того. Загалом відповідь залишається далі покроковою та інформативною з прикладами коду на кожному кроці, але змінються контекст щодо різних видів пайплайні, також надає на останньому кроці надає приклад реєстрації нового пайплайну, якого до того не надавала, що теж є дуже корисним. Більшість моделей може працювати з різними розмірами чанку, але оптимальний розмір може варіюватися залежно від завдання, данних та обмежень ресурсів.

2. Method: split on different separators, use semantic chunking

Виконаємо семантичний поділ.

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from transformers import AutoTokenizer

import semchunk

import tiktoken

from semantic\_text\_splitter import TextSplitter

from tokenizers import Tokenizer

from langchain.docstore.document import Document

EMBEDDING\_MODEL\_NAME = "thenlper/gte-small"

def split\_documents(

    knowledge\_base: List[LangchainDocument],

    tokenizer\_name: Optional[str] = EMBEDDING\_MODEL\_NAME,

) -> List[LangchainDocument]:

    tokenizer = Tokenizer.from\_pretrained(EMBEDDING\_MODEL\_NAME)

    splitter = TextSplitter.from\_huggingface\_tokenizer(tokenizer)

    docs\_processed = []

    for doc in tqdm(knowledge\_base):

        page\_content = doc.page\_content

        curr = splitter.chunks(page\_content, chunk\_capacity=(200, 1000))

        for i in curr:

            docs\_processed.append(Document(page\_content=str(i), metadata=doc.metadata))

    # Remove duplicates

    unique\_texts = {}

    docs\_processed\_unique = []

    for doc in docs\_processed:

        if doc.page\_content not in unique\_texts:

            unique\_texts[doc.page\_content] = True

            docs\_processed\_unique.append(doc)

    return docs\_processed\_unique

docs\_processed = split\_documents(

    RAW\_KNOWLEDGE\_BASE,

    tokenizer\_name=EMBEDDING\_MODEL\_NAME,

)

# Let's visualize the chunk sizes we would have in tokens from a common model

from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from\_pretrained(EMBEDDING\_MODEL\_NAME)

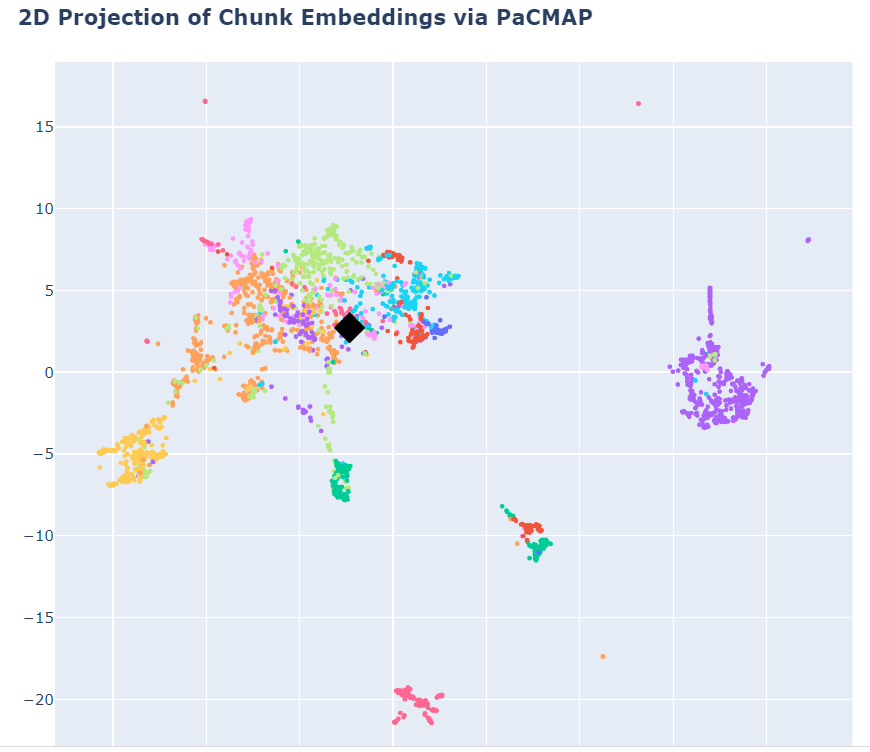
lengths = [len(tokenizer.encode(doc.page\_content)) for doc in tqdm(docs\_processed)]

fig = pd.Series(lengths).hist()

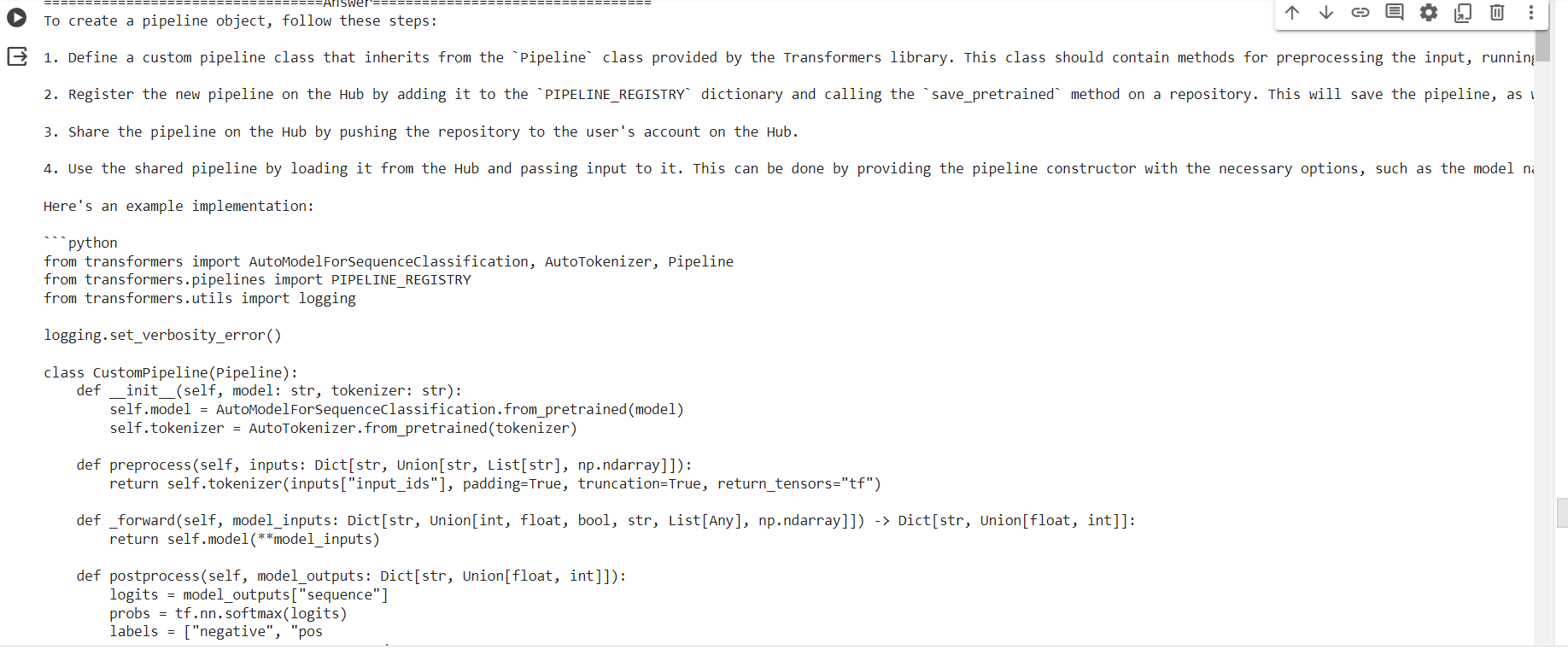
plt.title("Distribution of document lengths in the knowledge base (in count of tokens)")

plt.show()

Семантичний поділ тексту- це процес розділення тексту на малий набір значущих фрагментів або одиниць, які мають певний смисловий зв'язок. Цей підхід до аналізу тексту допомагає виокремити ключові ідеї, факти або сутності, що дозволяє зробити подальший аналіз більш ефективним та зрозумілим. Проаналізуємо розподіл сусідніх документів відповідно до запиту.



Можемо поміти що документи краще розподілились і утворили умовні кластери. Це поділ документів за їх сенсом. Розглянемо відповідь моделі на теж питання про пайплайн.



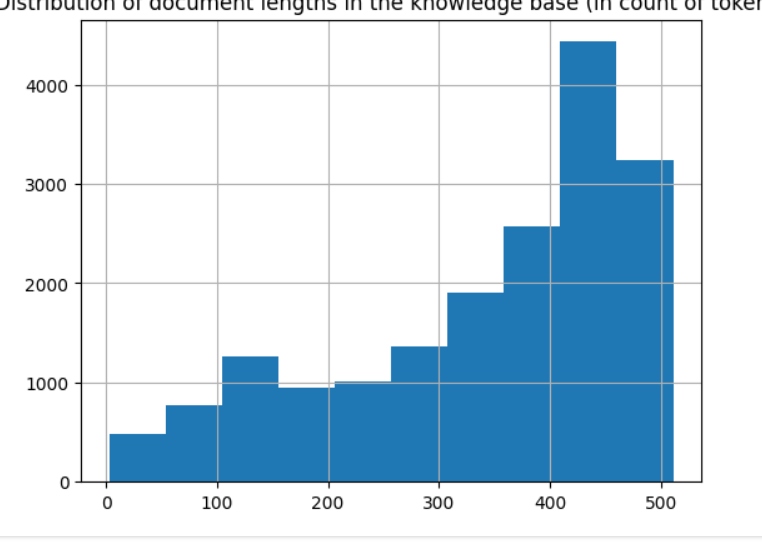
Можемо бачити що модель дала більш загальну відповідь ніж у попередніх випадках і не прив’язувалась до певних пайплайнів. Отже модель менше втрачає контекст, адже семантичний поділ ділить текст за сенсом тому модель краще відповідає і не прив’язується до певного контексту.

3. Try out different embedding model

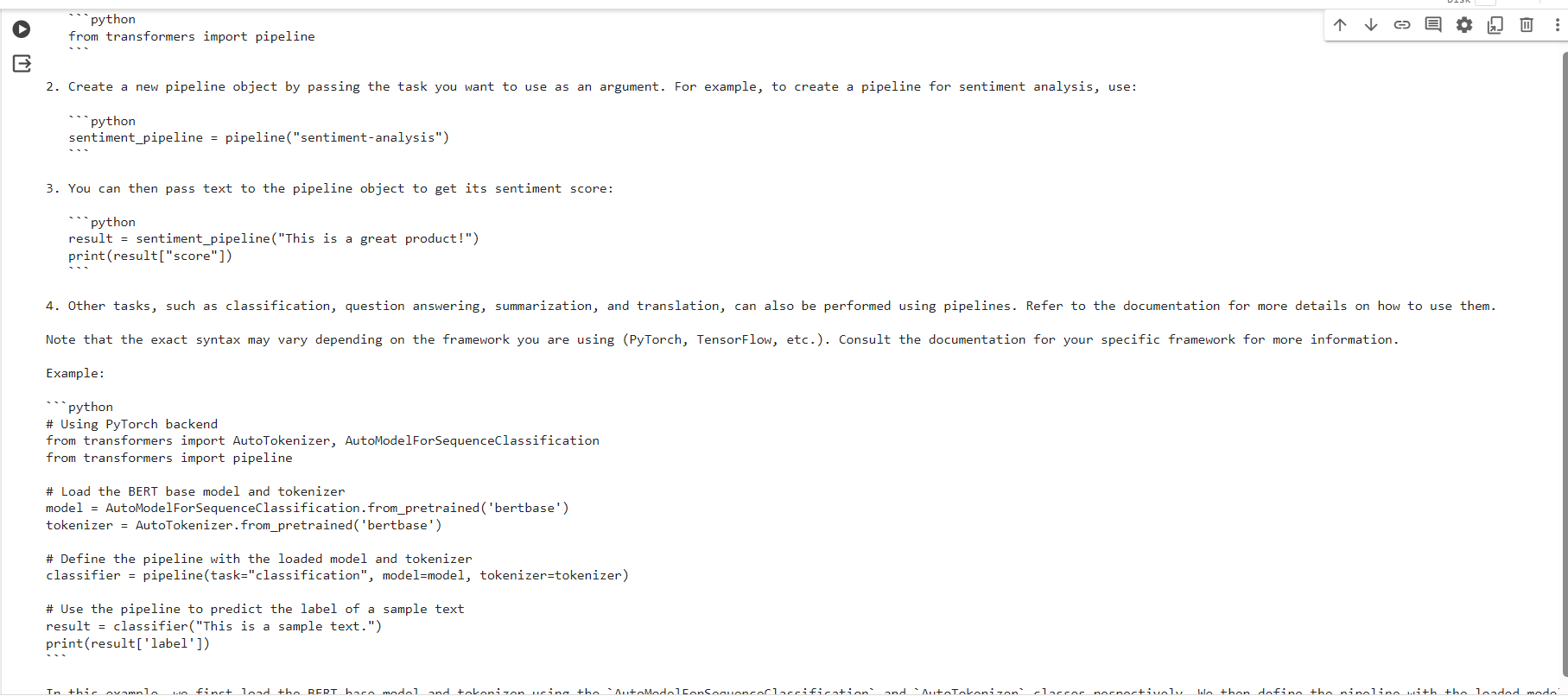
Спробуємо використати інші embedding моделі з сайту <https://huggingface.co/>.

Для початку я використав модель Mihaiii/gte-micro. Максимальна довжина послідовності в цієї моделі 512.

Проаналізуємо розподіл довжин документів у базі знань (у кількості токенів):

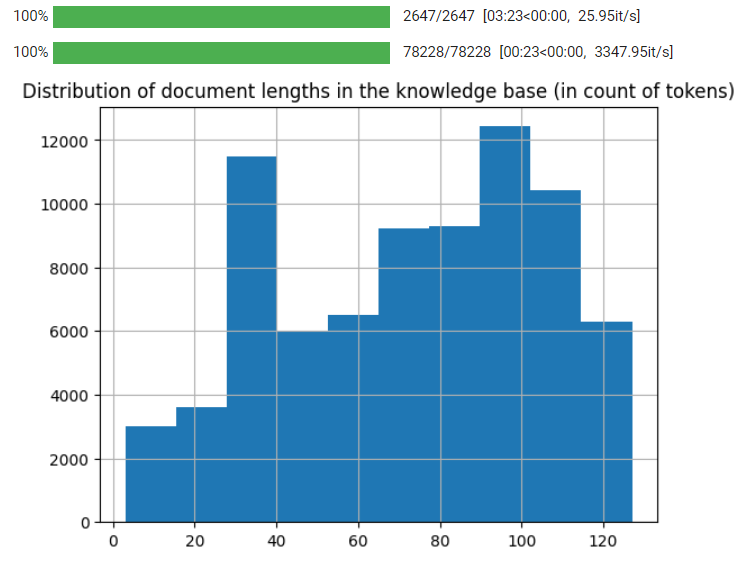


Тепер переглянемо отримані відповіді

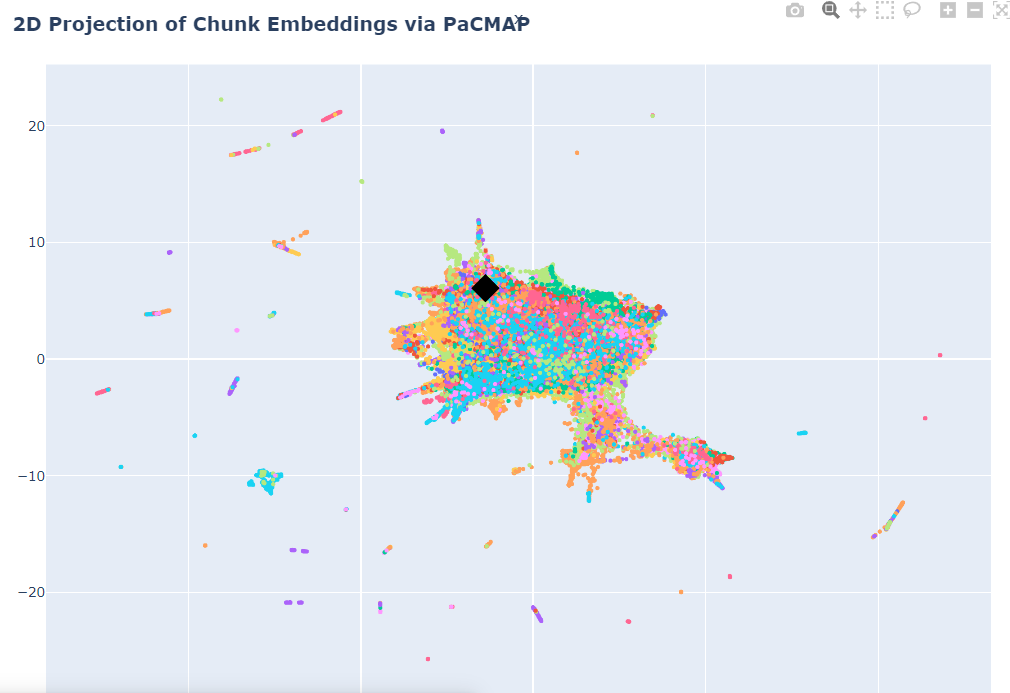


Можемо бачити що модель отримує документи в яких створюється пайплайн для семантичного аналізу. Але також варто помітити що модель дає загальний приклад приклад для інших задач але як приклад коду дає приклад роботи з NLP.

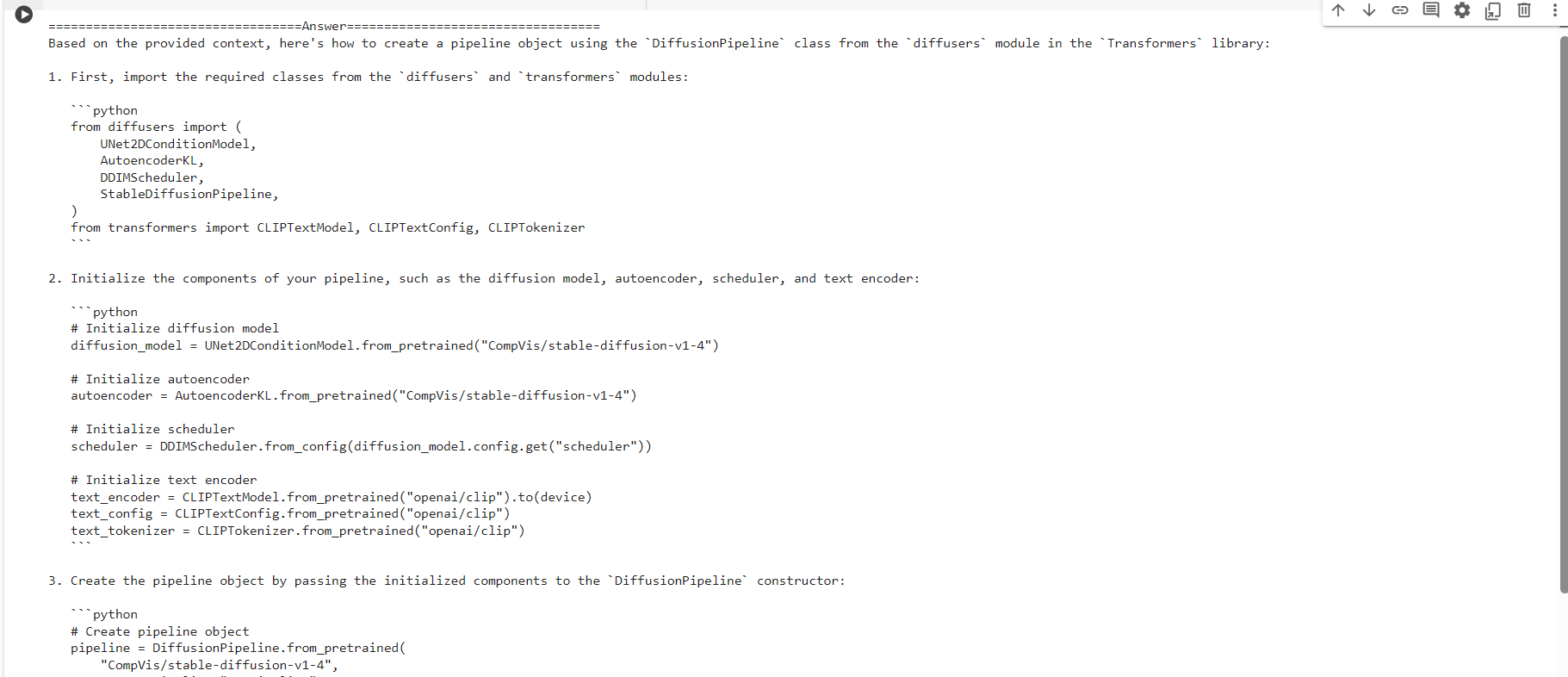
Також я використав модель sentence-transformers/bert-base-nli-stsb-mean-tokens. Максимальна довжина послідовності в цієї моделі 128. Проаналізуємо розподіл довжин документів у базі знань (у кількості токенів):



Також можемо бачити зміну розподілу документів



Переглянемо отриману відповідь.



Відповідь містить більше коду ніж пояснень, з одного боку це добре, адже ми отримуємо більше прикладів, але якщо користувач не дуже розуміється в темі то йому буде складніше розібратись з кодом. Це пов’язано на мою думку з максимальним розміром послідовності, який підтримує модуль, модель губить більш загальний контекст через маленький розмір.

6. Try different indexing algorithm used (here, FAISS)

Тут я використав Annoy. Основна ідея за алгоритмом Annoy полягає в побудові апроксимованих найближчих сусідів (ANN) через розбиття простору векторів на багато малих підпросторів і зберігання набору даних у дереві, де кожне листове піддерево представляє собою набір даних, що зберігається в пам'яті. Це дозволяє швидко знаходити найближчі вектори до даного запиту, оскільки пошук відбувається тільки в обмеженому підпросторі, а не у всьому просторі векторів. Одним з основних переваг Annoy є те, що вона досить ефективно працює з великими об'ємами даних та високими вимірами векторів, забезпечуючи при цьому швидкий доступ до найближчих сусідів. Вона також підтримує різні стратегії для розбиття дерева та пошуку, що дозволяє підібрати налаштування під конкретні потреби та обмеження.

from langchain.vectorstores import FAISS

from langchain.vectorstores import Annoy

from langchain\_community.embeddings import HuggingFaceEmbeddings

from langchain\_community.vectorstores.utils import DistanceStrategy

embedding\_model = HuggingFaceEmbeddings(

    model\_name=EMBEDDING\_MODEL\_NAME,

    multi\_process=True,

    model\_kwargs={"device": "cuda"},

    encode\_kwargs={"normalize\_embeddings": True},  # set True for cosine similarity

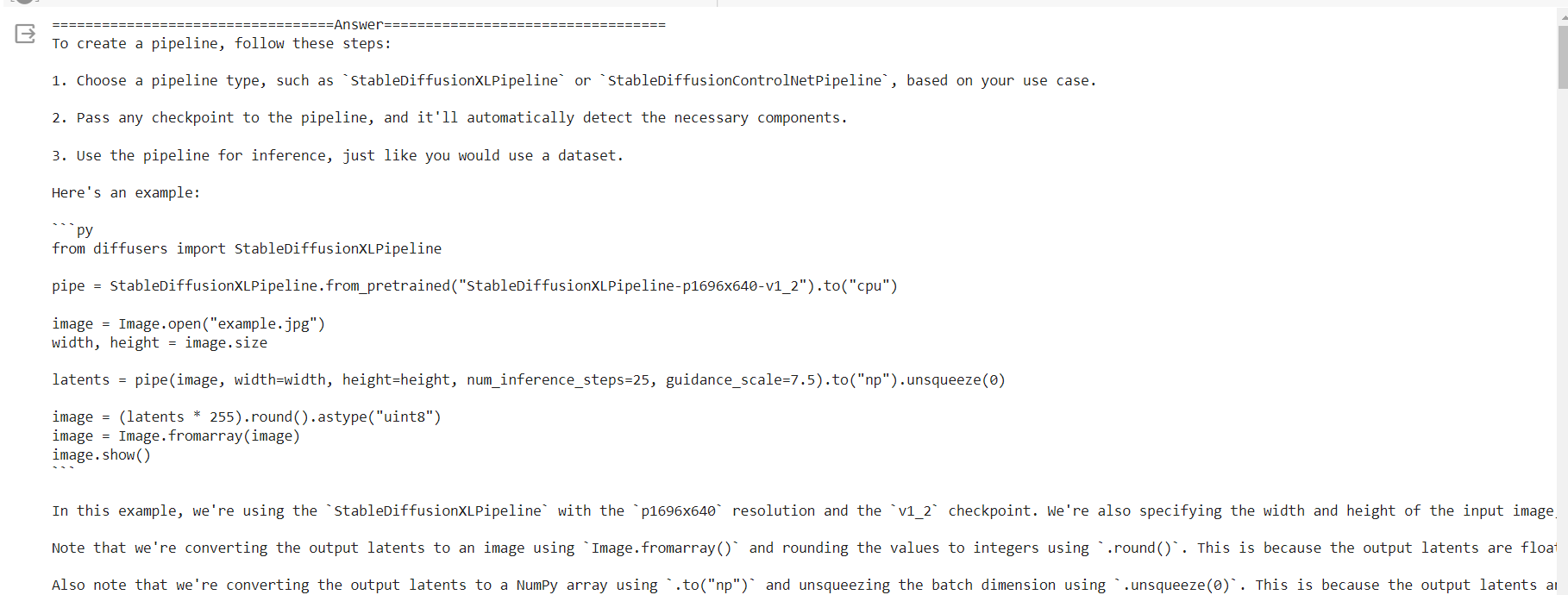
)

KNOWLEDGE\_VECTOR\_DATABASE = Annoy.from\_documents(

    docs\_processed, embedding\_model, distance\_strategy=DistanceStrategy.COSINE

)

По перше, хочу виділити що у моєму випадку при семантичних чанках та використання Annoy час створення бази зменшився в порівнянні з FAISS, адже як я писав вище цей алгоритм дуже добре працює з великими даними, а семантичні чанки утворюють доволі великі послідовності. Проаналізуємо отриману відповідь після зміни алгоритму створення векторизованої бази даних.



Можемо помітити що змінився приклад типу пайплайну через зміну алгоритму створення бази даних. Загалом відповідь інформативна, але не наведено якогось загального прикладу, проте доволі добре пояснено процес роботи пайтлайну в контексті CV.

7. Query expansion: reformulate the user query in slightly different ways to retrieve more documents

question = "What are the steps for creating a pipeline object??"

answer, relevant\_docs = answer\_with\_rag(

    question, READER\_LLM, KNOWLEDGE\_VECTOR\_DATABASE, reranker=RERANKER

)

Я змінив запит таким чином та отримав більше документів використаних у відповіді, мені вдалось отримати процес розгортання проекту на гітхабі після самого покрокового рішення створення пайплайну. Ось відповідь моделі.



Висновок

Оптимізація підказок: Чіткі та конкретні підказки стимулюють модель надавати більш інформативні відповіді, що відображають контекст та вимоги запиту. Покращення методів чанкування: Використання семантичного чанкування та оптимізованих розмірів чанків допомагає підвищити контекстуальну зрозумілість та релевантність отриманих відповідей. Експерименти з векторними моделями: Різні векторні моделі можуть мати різний вплив на якість відповідей, тому варто випробувати різні моделі та вибрати оптимальну для конкретного завдання. Використання різних алгоритмів індексації: Вибір підходящого алгоритму для індексації даних може покращити швидкодію та точність системи. Розширення запиту: Зміна формулювання запиту може допомогти отримати більше контекстуально відповідних документів для аналізу та відповідей. В цілому, експерименти показали, що кожна зміна в різних аспектах моделі може вплинути на якість отриманих відповідей, змінюючи їхню інформативність та контекстуальність. Варто вибирати оптимальні параметри та методи для конкретного завдання з метою отримання найкращих результатів.