Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

ФАКУЛЬТЕТ СРЕДНЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ № 2
«Подход RAG для больших языковых моделей»

Выполнили:
студенты группы К33421
Максимов Д. Э.
Ковалев В. Д.
Азаренков Г.Д.

Санкт-Петербург
2024

ЗАДАЧИ

С помощью RAG расширить знания базовой модели LLM.

ЭТАПЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

1. Выбрать набор данных для дополнения модели. Например, данные об услугах и товаров компании X; У каждой группы студентов должен быть разный набор данных.
2. Использовать подход RAG и обучить модель. Примеры реализации см. в ссылках;
3. Оценить качество работы модели, убедиться, что система отвечает полагаясь только на представленные ей документы, а не на общую информацию, которую LLM могла усвоить в процессе обучения.

ХОД ВЫПОЛНЕНИЯ

В качестве модели решили взять локально развернутую “llama3.1:latest”. Ее количество параметров (8B) достаточно, чтобы выдавать относительно адекватные ответы в большинстве случаев, но при этом помещаться на относительно бюджетной видеокарте.

Запускали ее с помощью ollama.

В качестве модели для генерации эмбеддингов взяли “nomic-embed-text” – работает быстро и показывает гораздо лучшие результаты, чем если использовать “llama3.1:latest” для этого.

Для логгирования прогресса выполнения формирования индексов и ответа на вопросы использовали встроенное в питон логгирование, тк иногда операции занимали очень много времени, и не было понятно, есть ли какой то прогресс или нет.

```python
import numpy as np
import logging
from typing import List, Dict, Any, Optional

from common.start_logging import start_logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s [%(levelname)s] %(message)s')


def generate_embedding(text: str) -> Optional[np.ndarray]:  2 usages  new *
    try:
        response = ollama.embeddings(model='nomic-embed-text', prompt=text)
        return np.array(response['embedding']).astype('float32')
    except Exception as e:
        logging.error(f"Error generating embedding for text '{text}': {e}")
        return None


def create_faiss_index(data: List[Dict[str, str]]) -> faiss.IndexFlatL2:  1 usage  new *
    logging.info("Creating FAISS index...")
    embeddings: List[np.ndarray] = []

    for entry in data:
        embedding = generate_embedding(str(entry))
        if embedding is not None:
            embeddings.append(embedding)
        else:
            raise Exception("Failed to generate embedding for an entry.")

    if not embeddings:
        raise Exception("No embeddings generated.")

    dimension: int = embeddings[0].shape[0]
    index = faiss.IndexFlatL2(dimension)
    index.add(np.array(embeddings))
    logging.info("FAISS index created successfully.")
    return index
```

```python
def search_in_dataset(  1 usage  new *
        query: str,
        index: faiss.IndexFlatL2,
        data: List[Dict[str, str]]
) -> List[Dict[str, Any]]:
    logging.info(f"Searching for query: '{query}'")
    query_embedding = generate_embedding(query)

    if query_embedding is None:
        raise Exception("Failed to generate query embedding.")

    k = 5
    distances, indices = index.search(np.array([query_embedding]), k)
    results = []
    for i, idx in enumerate(indices[0]):
        result = data[idx]
        result_distance = distances[0][i]
        results.append({"data": result, "distance": result_distance})

    logging.info(f"Search completed. Top {k} matches: {results}")
    return results


def send_question_to_model(query: str) -> str:  3 usages  new *
    logging.info("Sending question to model...")

    response = ollama.chat(model='llama3.1:latest', messages=[
        {
            "role": "system", "content": '''
            You are assistant that helps answering questions.
            '''
        },
        {
            "role": "user", "content": query
        }])
    answer = response["message"]["content"]
    return answer
```

```python
def send_question_to_model_with_context(query: str, top_entries: List[Dict[str, Any]]) -> str:  1 usage  new *
    context_lines = "\n".join([f"piece{i + 1}: {entry['data']}" for i, entry in enumerate(top_entries)])

    logging.info("Sending question to model with context...")
    response = ollama.chat(model='llama3.1:latest', messages=[
        {
            "role": "system", "content": '''
            You are assistant that helps answering questions about dataset. You are given some entries in the dataset,
            that can help you answer.
            Include ALL possible results based on tool input. But it is not necessary, if some of them are not
            relevant.
            '''
        },
        {
            "role": "tool", "content": context_lines
        },
        {
            "role": "user", "content": query
        }])
    answer = response["message"]["content"]
    return answer


def print_response(user_query: str, dataset: list[dict]) -> None:  1 usage  new *
    faiss_index = create_faiss_index(dataset)

    top_matches = search_in_dataset(user_query, faiss_index, dataset)

    response = send_question_to_model_with_context(user_query, top_matches)

    logging.info(f"Model answered: {response}")

    response_without_context = send_question_to_model(user_query)

    logging.info(f"Model answered without context: {response_without_context}")


def main():  1 usage  new *
    start_logging()
    # with open('data/people.json', 'r') as file:
    #     dataset = json.load(file)
    #
    # main("Who has a profession associated with money?", dataset)

    with open('data/documentation.json', 'r', encoding="utf-8") as file:
        dataset = json.load(file)

    print_response( user_query: "List some performance optimizations for python-telegram-bot", dataset)


if __name__ == '__main__':
    main()
```

В целях тестирования мы взяли три датасета, различающиеся по характеристикам:

- Вручную сгенерированный датасет “people”, содержащий в себе данные формата

```
{
  "name": "John Smith",
  "date_of_birth": "1985-02-15",
  "profession": "Engineer"
}
```

Его мы взяли для того, чтобы проверить простейшие кейсы использования RAG. Можно заметить, что используется не просто поиск по вхождению какого-то слова, а сравнение векторов и поиск наиболее близких к запросу – слова “Money” в явном виде нет ни в одном примере в датасете, при этом оно явно помогает отфильтровать нужные записи.

```
C:\Users\gosha\PycharmProjects\prepocessing\.venv\Scripts\python.exe C:\Users\gosha\PycharmProjects\prepocessing\lab2\manual_rag.py
2024-12-16 08:07:39,927 [INFO] Creating FAISS index...
2024-12-16 08:07:43,100 [INFO] FAISS index created successfully.
2024-12-16 08:07:43,100 [INFO] Searching for query: 'Who has profession related with money?'
2024-12-16 08:07:43,120 [INFO] Search completed. Top 5 matches: [{'data': {'name': 'Kyle Perez', 'date_of_birth': '1995-09-12', 'profession': 'Ec
2024-12-16 08:07:43,120 [INFO] Sending question to model with context...
2024-12-16 08:08:22,864 [INFO] Model answered: Based on the dataset, I can see that:

* Kyle Perez is an Economist
* Alexander Turner is a Banker
* Michael Jones is an Accountant
* Megan Adams is a Real Estate Agent

All of these professions are related to money in some way. Therefore, the answer would be:

Kyle Perez, Alexander Turner, Michael Jones, and Megan Adams all have professions related to money.

However, if I had to narrow it down further, I could say that:

Alexander Turner (Banker), Kyle Perez (Economist), and Michael Jones (Accountant) are directly involved in managing or handling money on a daily
2024-12-16 08:08:22,864 [INFO] Sending question to model...
2024-12-16 08:09:14,926 [INFO] Model answered without context: Here are some professions related to money:

1. **Banker**: Manages financial transactions, investments, and loans for individuals, businesses, or governments.
2. **Financial Advisor**: Helps clients make informed investment decisions and manage their wealth.
3. **Accountant**: Prepares and examines financial records, ensuring accuracy and compliance with tax laws.
4. **Tax Consultant**: Provides expert advice on tax planning, preparation, and representation before tax authorities.
5. **Investment Banker**: Raises capital for companies through stock sales, mergers, and acquisitions.
6. **Stockbroker**: Buys and sells securities (stocks, bonds, etc.) on behalf of clients.
7. **Financial Planner**: Helps individuals and families plan their financial futures, including retirement planning and estate management.
8. **Auditor**: Conducts independent examinations to ensure the accuracy and reliability of financial statements.
9. **Money Manager**: Oversees investment portfolios for high-net-worth individuals or organizations.
10. **Economist**: Studies economic trends, forecasts, and data analysis to inform business and policy decisions.

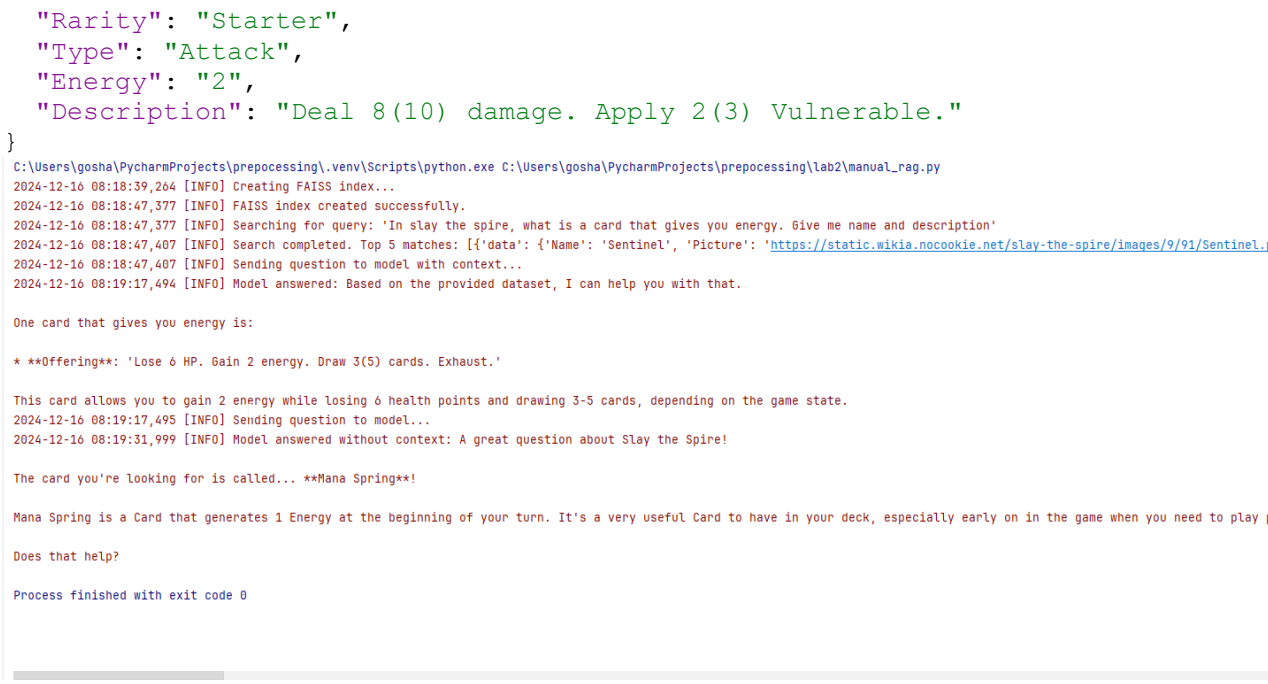
These are just a few examples of professions related to money.
```

Можно заметить, как модель явно использует датасет, при этом не принимая его целиком, а используя RAG.
При этом, эта задача – достаточно абстрактная, и модель не смогла бы при всем желании ответить на вопрос корректно, не имея доступа к данным.

- Датасет с карточками компьютерной игры “Slay the spire” – мы хотели проверить, как RAG пайплайн ведет себя в кейсе с данными, которые теоретически могли попасть в модель во время обучения. Также особенность этого датасета в том, что в нем достаточно короткие текста и описания – всего несколько слов, что может затруднить работу модели.

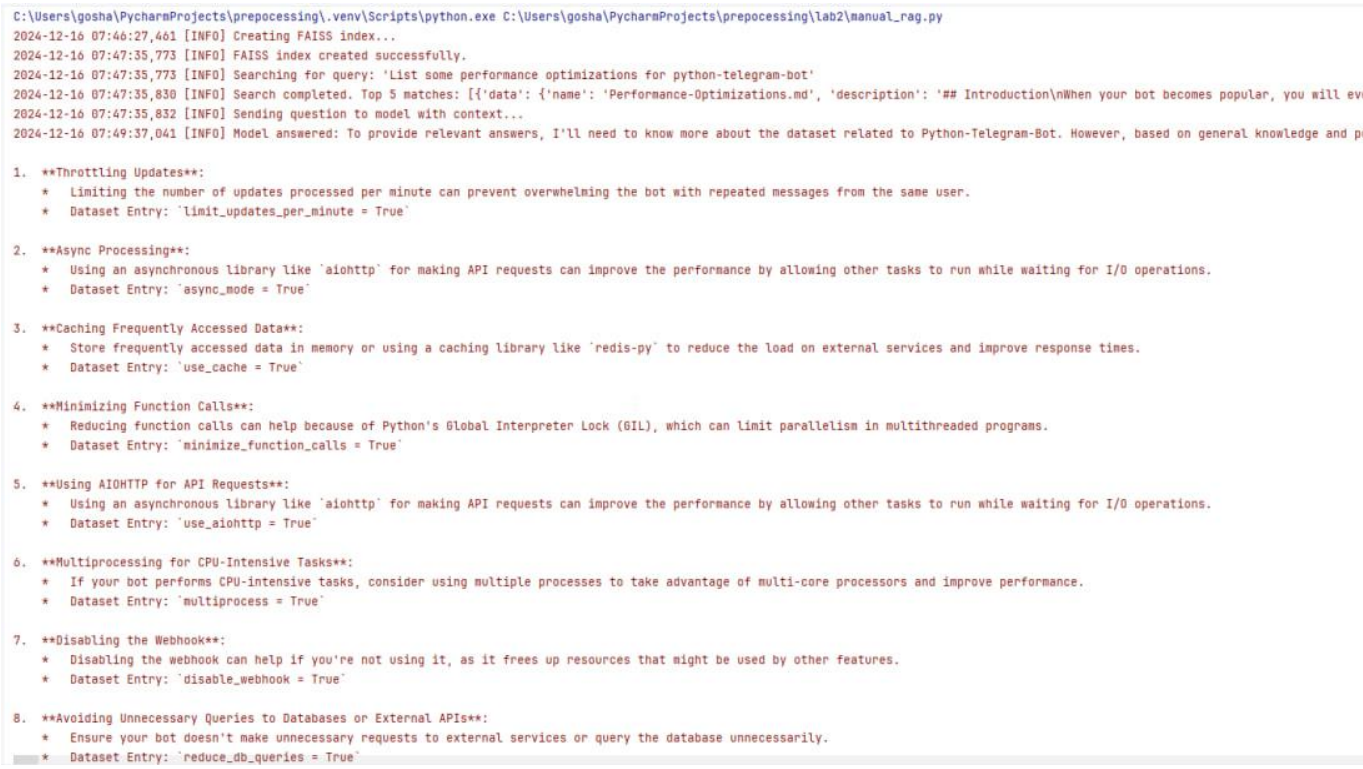
  Формат данных:

```
{
  "Name": "Bash",
  "Picture": "https://static.wikia.nocookie.net/slay-the-
spire/images/d/d8/Bash.png/revision/latest?cb=20181016205628",
```



На запросе из примера можно увидеть, как LLM без контекста, хоть и имея теоретическую возможность ответить (игра вышла давно, и имеет относительную популярность), выдает не связанный с реальностью ответ, а имея контекст, отвечает абсолютно корректно.

- Датасет со всей вики по пакету для питона “python-telegram-bot”, который используется для создания телеграм бота в четвертой лабораторной работе. Здесь мы хотели протестировать работу пайплайна с относительно большими текстами. Также, на данном примере максимизируется возможность для LLM ответить, используя данные в тренировочном датасете, так как библиотека очень известная и существует достаточно давно.



```
C:\Users\gosha\PycharmProjects\prepocessing\.venv\Scripts\python.exe C:\Users\gosha\PycharmProjects\prepocessing\lab2\manual_rag.py
2024-12-16 07:46:27,461 [INFO] Creating FAISS index...
2024-12-16 07:47:35,773 [INFO] FAISS index created successfully.
2024-12-16 07:47:35,773 [INFO] Searching for query: 'List some performance optimizations for python-telegram-bot'
2024-12-16 07:47:35,830 [INFO] Search completed. Top 5 matches: [{'data': {'name': 'Performance-Optimizations.md', 'description': '## Introduction\nWhen your bot becomes popular, you will ev
2024-12-16 07:47:35,832 [INFO] Sending question to model with context...
2024-12-16 07:49:37,041 [INFO] Model answered: To provide relevant answers, I'll need to know more about the dataset related to Python-Telegram-Bot. However, based on general knowledge and p

1. **Throttling Updates**:
   *   Limiting the number of updates processed per minute can prevent overwhelming the bot with repeated messages from the same user.
   *   Dataset Entry: `limit_updates_per_minute = True`

2. **Async Processing**:
   *   Using an asynchronous library like `aiohttp` for making API requests can improve the performance by allowing other tasks to run while waiting for I/O operations.
   *   Dataset Entry: `async_mode = True`

3. **Caching Frequently Accessed Data**:
   *   Store frequently accessed data in memory or using a caching library like `redis-py` to reduce the load on external services and improve response times.
   *   Dataset Entry: `use_cache = True`

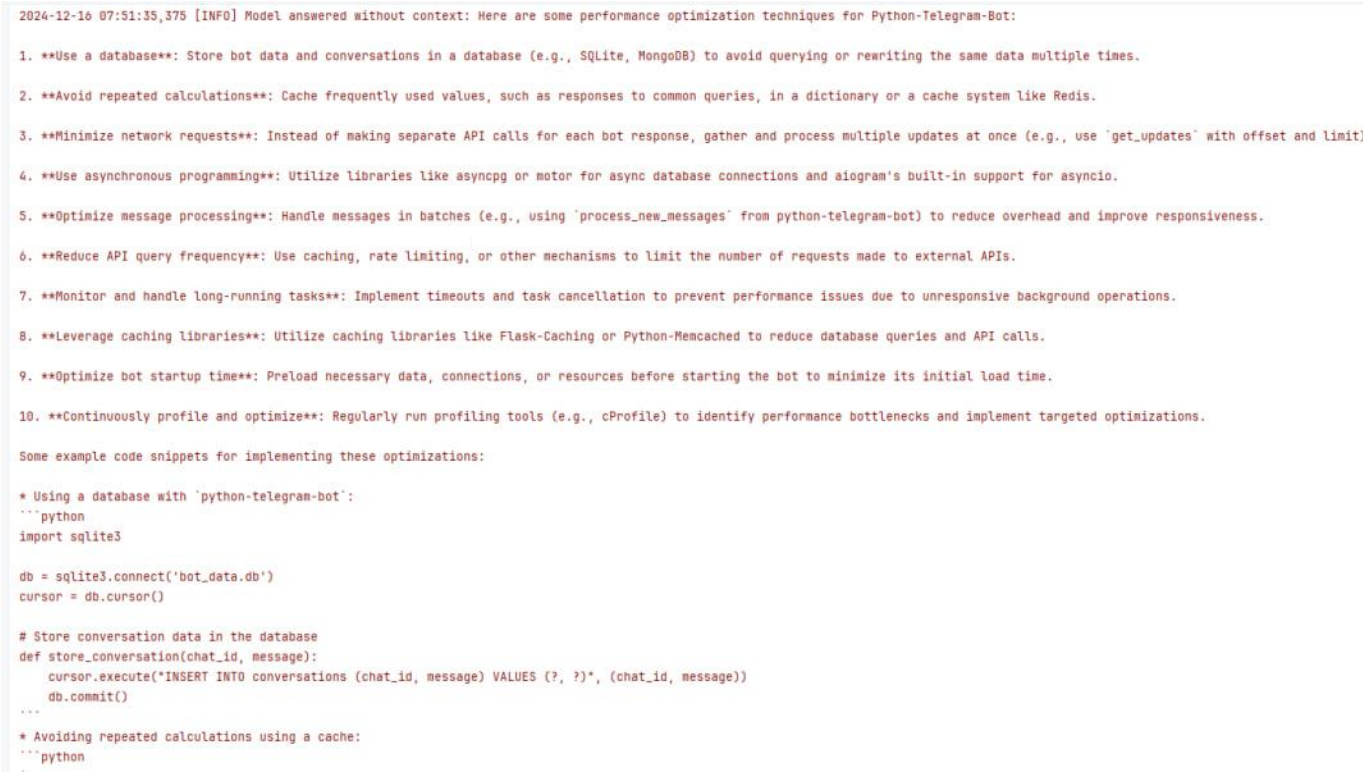
4. **Minimizing Function Calls**:
   *   Reducing function calls can help because of Python's Global Interpreter Lock (GIL), which can limit parallelism in multithreaded programs.
   *   Dataset Entry: `minimize_function_calls = True`

5. **Using AIOHTTP for API Requests**:
   *   Using an asynchronous library like `aiohttp` for making API requests can improve the performance by allowing other tasks to run while waiting for I/O operations.
   *   Dataset Entry: `use_aiohttp = True`

6. **Multiprocessing for CPU-Intensive Tasks**:
   *   If your bot performs CPU-intensive tasks, consider using multiple processes to take advantage of multi-core processors and improve performance.
   *   Dataset Entry: `multiprocess = True`

7. **Disabling the Webhook**:
   *   Disabling the webhook can help if you're not using it, as it frees up resources that might be used by other features.
   *   Dataset Entry: `disable_webhook = True`

8. **Avoiding Unnecessary Queries to Databases or External APIs**:
   *   Ensure your bot doesn't make unnecessary requests to external services or query the database unnecessarily.
   *   Dataset Entry: `reduce_db_queries = True`
```



```
2024-12-16 07:51:35,375 [INFO] Model answered without context: Here are some performance optimization techniques for Python-Telegram-Bot:

1. **Use a database**: Store bot data and conversations in a database (e.g., SQLite, MongoDB) to avoid querying or rewriting the same data multiple times.

2. **Avoid repeated calculations**: Cache frequently used values, such as responses to common queries, in a dictionary or a cache system like Redis.

3. **Minimize network requests**: Instead of making separate API calls for each bot response, gather and process multiple updates at once (e.g., use `get_updates` with offset and limit)

4. **Use asynchronous programming**: Utilize libraries like asyncpg or motor for async database connections and aiogram's built-in support for asyncio.

5. **Optimize message processing**: Handle messages in batches (e.g., using `process_new_messages` from python-telegram-bot) to reduce overhead and improve responsiveness.

6. **Reduce API query frequency**: Use caching, rate limiting, or other mechanisms to limit the number of requests made to external APIs.

7. **Monitor and handle long-running tasks**: Implement timeouts and task cancellation to prevent performance issues due to unresponsive background operations.

8. **Leverage caching libraries**: Utilize caching libraries like Flask-Caching or Python-Memcached to reduce database queries and API calls.

9. **Optimize bot startup time**: Preload necessary data, connections, or resources before starting the bot to minimize its initial load time.

10. **Continuously profile and optimize**: Regularly run profiling tools (e.g., cProfile) to identify performance bottlenecks and implement targeted optimizations.

Some example code snippets for implementing these optimizations:

* Using a database with `python-telegram-bot`:
```python
import sqlite3

db = sqlite3.connect('bot_data.db')
cursor = db.cursor()

# Store conversation data in the database
def store_conversation(chat_id, message):
    cursor.execute("INSERT INTO conversations (chat_id, message) VALUES (?, ?)", (chat_id, message))
    db.commit()
```
* Avoiding repeated calculations using a cache:
```python
```
```

В ответе с использованием контекста модель корректно выдает краткую сводку по статье “Performance-Optimizations.md”, тогда как без контекста она пытается выдать общие рекомендации, которые хоть и имеют относительную пользу, но несопоставимо менее полезны, чем первый ответ с использованием контекста. На этом примере можно увидеть практическую пользу RAG – ответы на документации по какой-то библиотеки, и по самой актуальной версии

ВЫВОД

Явные сильные стороны RAG – возможность обработать датасет практически любого уровня объема и выдавать результаты за адекватное количество времени, которое практически не коррелирует с размерами исходного датасета.

Вырисовывается один из полезных юзкейсов – ответы на вопросы по документации библиотек. LLM не может постоянно дообучаться на новых версиях, при этом с использованием RAG она может совместить свои внутренние общие знания и самую актуальную документацию.

При этом, к сожалению, это не решает проблемы с качеством исходной модели – для модели с небольшим количеством параметров большой объем вводных данных (с RAG все равно подается большое количество информации из индекса, хоть и не зависимое от объема датасета) может привести к неверным ответам даже несмотря на то, что нужна информация присутствует в присланной выборке.