

תיעוד – פרויקט מחקרי, בינה מלאכותית במקורות תורניים

	תוכן
2	שלבי הפעולה שלנו
3	מבנה הפרויקט
3	ארכיטקטורה
4	תיאור הקוד
11	תכנון מול ביצוע
11	גילויים
11	בעיות
11	מבט להמשך
12	תוכן הגיט

שילבי הפעולה שלנו

בשלב הראשון: התחלנו בתהליך למידת הנושא, קראנו הרבה, וראינו סרטונים בנושא.

בשלב השני: ניסינו ליצור LLM בשימוש בRag על ספר בראשית בשפה האנגלית, של מכון ממרה. הצלחנו ליצור בוט שעונה על שאלות, קצת אי דיוקים לפעמים אבל עונה על הכל, השתמשנו במודל של cohere עם פונקציית embedding שלהם.

הקוד מדפיס את התשובה של הבוט ואת ה"citations" (איפה הבוט מצא את התשובה בחילוק הקבצים שאצלו, מיקום מדויק יותר מבdocuments), ואת ה"documents" (איפה ניתן למצוא את התשובה בקבצים שאנחנו שמנו), זאת במטרה לאפשר בהמשך שהמשתמש יכניס משפט לחיפוש והמודל יחפש אותו ודברים קשורים לו בצורה רוחבית ויחזיר לו מקורות לחפש בהם את התשובה.

בשלב השלישי: ניסינו את אותה המכונה על ספר בראשית בעברית, נתקלנו בהמון אי-דיוקים ושימוש במידע חיצוני לקבצים שלנו בכמות מסוימת של קבצים, ומקרים נוספים בהם הבוט אף "המציא" מקורות שהבאנו לו.

בשלב הרביעי: חזרנו עוד ללמידה וחיפוש, תוך הרבה דיבוג של התוכנית, הצלחנו לדייק במקצת את התשובות של המודל, כך שיצאו נכונות יותר, אך עדיין נתקלים בבעיה של המידע החיצוני שהבוט משתמש בו, ובחרטוטים.

בשלב החמישי: ניסינו לשלב את המודל של cohere שישתמש בפונקציית embedding - "Berel" (BERT Embeddings for Rabbinic-Encoded Language) של Dicta, גם שם נתקלנו באותה הבעיה, אך הגענו גם לתשובות מדויקות יותר בחלק מהמקרים.

בשלב השישי: בעיקר חקרנו את הקוד עצמו עם דיבוגים רבים ונסיונות רבים בסוגי שאלות שונים, והצלחנו לזהות כשהבוט משתמש במידע חיצוני, אך זה הדגיש את הבעיה הקודמת שעבור כמות קבצים זה כבר לא מחזיר בכלל תשובות, מכאן ניסינו לסדר. כאן נתקענו, עבור כמות קטנה זה כבר כמעט מדויק לגמרי (אך לפעמים מתקש לא להחזיר תשובה ואומר שזה לא נמצא בקבצים שלו) אך עבור כמות מסוימת של קבצים (תלוי איזה) זה כבר לא מחזיר תשובות מהמידע שלנו ורק ממידע חיצוני.

מבנה הפרויקט

ארכיטקטורה

יהיו לנו 3 מחלקות: Documents, Chatbot, App.

Documents

המחלקה תקבל רשימה של dictionaries שכל dictionary ייצג קובץ ויהיה בו את הכותרת של הקובץ ואת path שלו.

במחלקה זו ננהל את כל העבודה עם קבצי המידע לקוד, המחלקה תהיה אחראית על:

- טעינת הקבצים מהpaths שיש לנו.
- embedding לקבצים (כאן יהיה הבדל אם נשתמש בפונקציית embedding של dicta לבין פונקציית embedding של cohere)
- indexing לקבצים.
- Retrieve לקבצים.

Chatbot

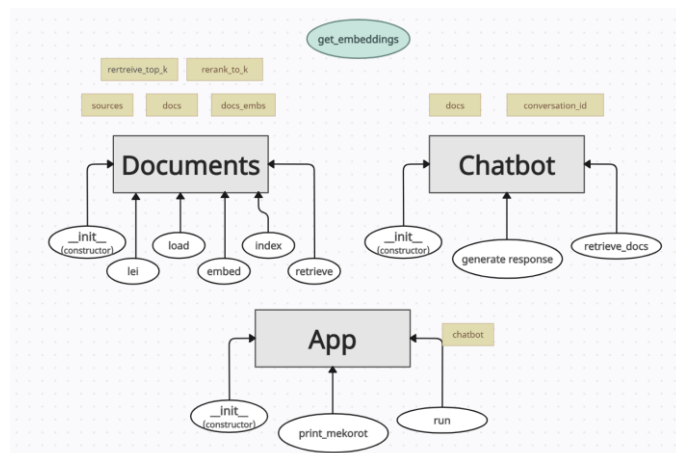
המחלקה הזאת תהיה אחראית על כל התקשורת מול המודל שלנו (אנחנו משתמשים במודל של cohere), גם המחלקה הזאת תשמור את הקבצים שלנו (לאחר שהDocuments עבדה איתם).

במחלקה יש פונקציה generate_response שבעזרתה נקבל תשובות מהמודל שלנו, וretrieve_docs שתהיה אחראית על מציאת הקבצים הרלוונטיים לתשובה.

App

במחלקה זו יש פונקציית run האחראית על כל התנהלות התקשורת מול המודל, בעזרת המחלקות האחרות וקריאה לפונקציות הרלוונטיות. הצאט יהיה בלולאה אינסופית של שאלה-תשובה למודל ואופציה לכתיבת exit ליציאה מהתוכנית.

תרשים – מבנה המחלקות:



במלבנים האפורים אלה המחלקות ולכל מחלקה יש את המלבנים הצהובים שלידה – השדות ששמורים במחלקה, ובעיגולים המחוברים אליה - הפונקציות של המחלקה. בנוסף, יש את הפונקציה `get_embeddings` שהיא מחוץ למחלקות, והיא ירוקה כי היא נמצאת רק בשימוש בDicta.

תיאור הקוד

• ספריות

```
from typing import List, Dict

import cohere
import hnswlib
import uuid
from unstructured.chunking.title import chunk_by_title
from unstructured.partition.text import partition_text
```

לפני הרצת הקוד יש לשים לב שהספריות הרלוונטיות מותקנות.

שימו לב! בשביל שספריית cohere תעבוד, יש לשים לב שמשתמשים בגרסה `cohere-4.54`.

• הכנות ראשוניות

ראשית, נטען את המודל:

```
co = cohere.Client('2ELFLZKqLyZi5bLI6wt1kDBMpoAT9ch44DHfycAm') # This is your trial API key
```

טעינת המודל של cohere עם API key שלנו.

בשביל שימוש בפונקציית embedding של Dicta נוסף גם:

```
model_name = 'dicta-1l/BEREL_2.0'
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained('dicta-1l/BEREL_2.0')
model.eval()
```

טעינת model של dicta לצורך שליפת embeddings בהמשך, בעזרת פונקציה (חינונית למחלקות):

```
def get_embeddings(texts):
    # Tokenize and convert to tensors
    inputs = tokenizer(texts, padding=True, truncation=True, return_tensors='pt')

    # Generate embeddings
    with torch.no_grad():
        outputs = model(**inputs)

    # Extract the last hidden state (token embeddings)
    embeddings = outputs.last_hidden_state

    # Average the token embeddings to get sentence embeddings
    sentence_embeddings = torch.mean(embeddings, dim=1).tolist()

    return sentence_embeddings
```

ולסיום ההכנות:

```
sources = []
for i in range(50):
    sources.append(
        {
            "title": "חנך - בראשית" + str(i + 1),
            "fileName": "bereshit\\bereshit " + str(i + 1) + ".txt"
        }
    )
```

ניצור מערך sources, אליו נטען את המקורות שאנחנו רוצים לחפש בהם תשובות, אשר כל מקור הוא dictionary של title עם הכותרת של הקובץ וfileName עם הpath לקובץ (אצלנו path רלטיבי).

• הכנת הקבצים

בעזרת המחלקה documents נכין את הקבצים לחיפוש בהם:

```
documents = Documents(sources)
```

ניצור מופע של המחלקה עם המערך של ה"קבצים" שלנו (זה עוד לא לגמרי הקבצים כי התוכנית יודעת כעת רק מה הpath אליהם ועוד לא הוציאה את התוכן שלהם).

```
class Documents:
    # loads the documents, the embed and index happens here.
    new *
    def __init__(self, sources: List[Dict[str, str]]):
        self.sources = sources
        self.docs = []
        self.docs_embs = []
        self.retrieve_top_k = 10
        self.rerank_top_k = 5
```

הבנאי ישמור את הקבצים, וייצור לו משתנים שיעבוד איתם בהמשך (נפגוש כל אחד במקומו הרלוונטי). ונקרא לפונקציה:

```
def lei(self, index_dim):
    self.load()
    self.embed()
    self.index(index_dim)
```

← documents.lei(1024)

שאחראית לקריאה לפונקציות (יחד עם המימד, בשימוש בcohere המימד יהיה 1024 ובשימוש dictab המימד יהיה 768:

- load – קריאת תוכן הקבצים

```
def load(self) -> None:
    print("Loading documents...")

    for source in self.sources:
        elements = partition_text(filename=source["fileName"], strategy="hi_res", include_page_breaks=True)
        chunks = chunk_by_title(elements)
        for chunk in chunks:
            self.docs.append(
                {
                    "title": source["title"],
                    "text": str(chunk),
                    "fileName": source["fileName"],
                }
            )
```

נעבור על הקבצים, נקרא את התוכן בעזרת path, ונוסיף dictionary עם הכותרת, התוכן והpath, למערך docs שיצרנו במחלקה בבנאי.

- embed – פונקציית הembedding

: Cohere

```
def embed(self) -> None:
    print("Embedding documents...")

    batch_size = 90
    self.docs_len = len(self.docs)

    for i in range(0, self.docs_len, batch_size):
        batch = self.docs[i: min(i + batch_size, self.docs_len)]
        texts = [item["text"] for item in batch]
        docs_embs_batch = co.embed(
            texts=texts,
            model="embed-multilingual-v3.0",
            input_type="search_document"
        ).embeddings
        self.docs_embs.extend(docs_embs_batch)
```

נבחר את גודל הbatch (כאן 90), ונעבור בלולאה בקפיצות של גודל הbatch עד האורך של docs, ניקח את התוכן ונעשה embed דרך coheren במודל embed-multilingual-v3.0 המתאים גם לעברית ונוסיף את זה למשתנה docs_embs שיש לנו במחלקה, שיהיה list של הקבצים שלנו אחרי שעברו "אימבוד".

: Dicta

כאן זה יהיה קצת שונה, ניקח את פונקציית הembedding מהמודל שצירפנו:

```
def get_embeddings(texts):
    # Tokenize and convert to tensors
    inputs = tokenizer(texts, padding=True, truncation=True, return_tensors='pt')

    # Generate embeddings
    with torch.no_grad():
        outputs = model(**inputs)

    # Extract the last hidden state (token embeddings)
    embeddings = outputs.last_hidden_state

    # Average the token embeddings to get sentence embeddings
    sentence_embeddings = torch.mean(embeddings, dim=1).tolist()

    return sentence_embeddings
```

ובעזרת הפונקציה הזאת, נוכל לעשות אימבוד לקבצים בצורה דומה לב cohere:

```
def embed(self) -> None:
    print("Embedding documents...")

    batch_size = 70
    self.docs_len = len(self.docs)

    for i in range(0, self.docs_len, batch_size):
        batch = self.docs[i: min(i + batch_size, self.docs_len)]
        texts = [item["text"] for item in batch]
        docs_embs_batch = get_embeddings(texts)
        self.docs_embs.extend(docs_embs_batch)
```

גם כאן נבחר גודל batch (כאן ראינו שהכי טוב 70) ושוב נרוץ בלולאה בקפיצות של גודל batch, נאמבד בעזרת פונקציית העזר שלנו, ונוסיף את זה לרשימת הקבצים המאומבדים שלנו.

- Index – אינדוקס

הפונקצייה הזאת תהיה זהה בין dicta וcohere עם מימדים שונים ביניהם leia המוזכרת לעיל (cohere המימד יהיה 1024 ובdicta המימד יהיה 768):

```
def index(self, dim) -> None:
    print("Indexing documents...")

    self.index = hnswlib.Index(space="ip", dim=dim)
    self.index.init_index(max_elements=self.docs_len, ef_construction=512, M=64)
    self.index.add_items(self.docs_embs, list(range(len(self.docs_embs))))

    print(f"Indexing complete with {self.index.get_current_count()} documents.")
```

הקוד שמוצג כאן מבצע אינדוקס של מסמכים עבור שליפה יעילה בעזרת ספריית hnswlib, הפונקציה index יוצרת אינדקס של וקטורי מסמכים עבור חיפוש קרובים במהירות גבוהה באמצעות אלגוריתם HNSW, עם וקטורים בממד 768, לפי dicta

• יצירת מופע Chatbot

```
chatbot = Chatbot(documents)
```

ניצור מופע של chatbot עם המופע של הקבצים שלנו (המחלקה האחראית על הניהול שלהם שיצרנו):

```
class Chatbot:
    new *
    def __init__(self, docs: Documents):
        self.docs = docs
        self.conversation_id = str(uuid.uuid4())
```

ונשמור לנו במופע את הקבצים, ואת id של השיחה לצורך התקשורת מול הבוט בהמשך.

בתוך Chatbot יש לנו שני פונקציות, נסביר כל אחת:

- Generate_response – יצירת תגובה לבקשת ושאלת המשתמש

```

response = co.chat(message=message, model="command-r-plus",
search_queries_only=True)

if response.search_queries:
    print("Retrieving information...")

    documents = self.retrieve_docs(response)
    response = co.chat(
        message=message,
        documents=documents,
        model="command-r-plus",
        conversation_id=self.conversation_id,
        stream=True,
    )

```

הקוד בונה אובייקט מסוג של chat עם הפרמטרים הבאים: בקשת/שאלת הלקוח, המודל שבו נעשה שימוש ו `search_queries_only=True` שמציין שהקריאה הזו היא רק לשליפת שאליות חיפוש ולא למענה מלא. במילים אחרות, המודל רק מחזיר את השאליות או הנושאים שאותם צריך לחפש במאגרי מידע נוספים כדי להשיב על הבקשה (שאצלינו זה ספר בראשית).

אם התשובה הראשונית של המודל (`response`) כוללת שאליות חיפוש (כלומר, יש צורך בשליפת מידע נוסף), הקוד בודק זאת באמצעות `response.search_queries`. אם יש שאליות חיפוש, הוא מדפיס הודעה שמעידה על כך שהולכים לשלוח מידע נוסף ממאגרי נתונים או ממקורות חיצוניים.

בפסיקה זו, הפונקציה `self.retrieve_docs` מופעלת, והיא לוקחת את השאליות ומבצעת חיפוש במאגרי מידע כדי לאסוף את המידע או המסמכים הרלוונטיים, לבסוף נשמור את המסמכים שנשלפו במשתנה `documents`.

לאחר שליפת המסמכים, הקריאה למודל השפה מתבצעת שוב עם אותם פרמטרים, אך כעת כוללים גם את המסמכים שנמצאו בשלב הקודם בשביל תשובה מדויקת יותר.

```

for event in response:
    yield event
if not response.documents:
    print("התנך על שאלה שאלה\nסליחה, לתנך קשורה הייתה לא השאלה")
    return
yield response

```

בקוד הנ"ל אנחנו עוברים בלולאה על האירועים בתגובה, אם התגובה מתקבלת בזרימה (`streaming`), התוכנה מחזירה כל אירוע בנפרד ברגע שהוא מגיע. לאחר מכן בודקים האם התשובה (`response`) "נבעה" ממסמכים שלנו (ספר בראשית), אם השאלה לא "נבעה" מהמסמכים הרלוונטיים (ספר בראשית), התוכנה מודיעה על כך למשתמש ועוצרת את הפעולה. לבסוף אם נמצאו מסמכים רלוונטיים בתשובה, אז נחזיר אותה, ולבסוף תהיה לנו תשובה מלאה.

```

else:
    print("התנך על שאלה שאלה\nסליחה, לתנך קשורה הייתה לא השאלה")
    return

```

אבל אם התשובה הראשונית של המודל (`response`) לא כוללת שאליות חיפוש (כלומר, אין צורך בשליפת מידע נוסף), סביר להניח שהשאלה לא קשורה לתנך ולכן נחזיר ללקוח את ההודעה הנ"ל ונסיים.

- `retrieve_docs` - שליפת מסמכים על סמך שאליות שמתקבלות מתוך האובייקט `response`

```

queries = []
for search_query in response.search_queries:
    queries.append(search_query["text"])

# Retrieve documents for each query
retrieved_docs = []
for query in queries:
    retrieved_docs.extend(self.docs.retrieve(query))

return retrieved_docs

```


הפונקציה מקבלת את ה response-שמכיל שאילתות חיפוש, ואז היא שולפת את הטקסט של כל שאילתה ומוסיפה אותו לרשימה queries, לאחר מכן היא מבצעת חיפוש על כל שאילתה באמצעות קריאה ל self.docs.retrieve(query) ומאגדת את כל המסמכים שנמצאו, ולבסוף היא מחזירה את כל המסמכים שנשלפו בפורמט של רשימה של מילונים.

• יצירת מופע App

```
app = App(chatbot)
```

יצירת ה"אפליקציה", המחלקה האחראית על ניהול התקשורת מול chatbot והלקוח שמכניס שאילתות בעזרת הפונקציות שנצטרך, ניצור מופע עם chatbot שיצרנו:

```
class App:
    new *
    def __init__(self, chatbot: Chatbot):
        self.chatbot = chatbot
```

שמירת מופע chatbot למחלקה.

וכעת, לאחר כל ההכנות, ניתן לעבור להרצת התוכנית עצמה בעזרת app שלנו:

```
app.run()
```



```
def run(self):
    while True:
        # Gets the user message
        message = input("User: ")

        # Typing "quit" ends the conversation
        if message.lower() == "quit":
            print("Ending chat.")
            break
```

מתחילים לולאה שתרוץ בלי הפסקה, המקבלת ראשית קלט מהמשתמש, אם הוכנס quit אז יוצאים מהלולאה לסיום התוכנית.

אחרת:

```
else:
    print(f"User: {message}")

    response = self.chatbot.generate_response(message)
```

מדפיסים את השאלה/בקשה של הלקוח למסך ולאחר מכן קוראים לפונקציה generate_response בתוך chatbot עם בקשת/שאלת המשתמש והתגובה הזו תישמר במשתנה response, שהוא זרם (stream) של אירועים.

```
def printText(texts):
    for text in texts:
        print(text, end="")
        time.sleep(0.05)
```

פונקציה זו מוגדרת כדי להדפיס את התשובה באופן הדרגתי, כאשר כל תו מודפס עם השהייה קצרה של 0.05 שניות בין כל תו. זה מדמה תהליך שבו הציטבוט מדפיס את התשובה בצורה איטית יותר, כאילו הוא "מקליד" את התגובה.

```

text = []
print("Chatbot:")
citations_flag = False
printed_text_flag = False

```

יוצרים רשימה ריקה בשם text כדי לאגור את הטקסט שנוצר בהמשך, מדפיסים "Chatbot:" כהכנה להצגת תגובת הציטוט. משתנים citations_flag ו printed_text_flag משמשים לבדיקת האם הודפס טקסט ראשי או האם ציטוטים כבר הוצגו (נבין יותר מאוחר למה).

```

for event in response:
    stream_type = type(event).__name__

```

לולאה זו עוברת על כל האירועים בתוך התגובה (זרם ה response), ולכל אירוע מוציאים את הסוג שלו באמצעות type(event).__name__.

```

if stream_type == "StreamTextGeneration":
    text.append(event.text)

```

אם סוג האירוע הוא StreamTextGeneration, משמע מדובר בטקסט שנוצר על ידי הציטוט. הטקסט מצורף לרשימה text שהגדרנו מקודם.

```

if stream_type == "StreamCitationGeneration":

    if not printed_text_flag:
        printText(text)
        printed_text_flag = True

    if not citations_flag:
        print("\n\nCITATIONS:")
        citations_flag = True

    print(event.citations[0])

```

אם האירוע הוא StreamCitationGeneration, כלומר מדובר בציטוטים שהציטוט מחזיר או תחילה נבדוק אם הטקסט לא הודפס עדיין, ואם לא – מודפסת התשובה המצטברת (הרשימה text). אם לא הודפסו ציטוטים קודם לכן, תודפס המילה "CITATIONS:" כהקדמה לציטוטים שיוצגו (נראות הפלט) ולאחר מכן הציטוט הראשון מתוך האירוע מודפס למסך.

```

if citations_flag:
    if stream_type == "StreamingChat":
        print("\n\nDOCUMENTS:")
        documents = [{'id': doc['id'],
                        'text': doc['text'][:50] + '...',
                        'title': doc['title'],
                        'fileName': doc['fileName']}
                      for doc in event.documents]
        self.printMekorot(documents)
        for doc in documents:
            print(doc)

```

אם הוצגו ציטוטים (סימן לכך שיש צורך להציג מסמכים רלוונטיים):

- בודקים אם סוג האירוע הוא StreamingChat, אם כן אז תודפס המילה "DOCUMENTS:" כהקדמה להצגת המסמכים.
- לאחר מכן יוצרים רשימה של מסמכים מקוצרים שבהם לכל מסמך מוצגים השדות: id, חלק מהטקסט הראשוני של המסמך (50 תווים), הכותרת, ושם הקובץ. כל מסמך מודפס לאחר מכן למשתמש.

ובנוסף, נדפיס את המקורות בעזרת הפונקציה printMekorot:

```

def print_mekorot(self, documents):
    str = ''
    for document in documents:
        str += document['title'] + ", "
    print("\n\nתוכל למצוא את התשובה ב: " + str)

```

ניצור מחרוזת ארוכה של כל titlen של documentn הרלוונטיים (הכותרת זה שם הקובץ שקבענו בהתחלה), ונדפיס user את כל המקורות (בקצרה יותר, רק את הכותרות) יחד עם "תוכל למצוא את התשובה ב:".

בסיום התהליך, מודפס קו מפריד של 100 מקפים כדי לסמן את סוף מחזור השיחה הנוכחי:

```
print(f"\n{'-' * 100}\n")
```

תכנון מול ביצוע

גילויים

לגבי המשך עבודה עם cohere:

- באופן כללי נראה שעל קבצי טקסט רגילים המודל שלנו עובד, נתקלנו בבעיות רק בעברית בתנך (שהיה לו ידע על זה כבר) והוא לא חיפש בקבצים שלנו מכמות מסוימת של קבצים.
- נראה שעם פונקציית embedding של dicta הוא קצת מדויק יותר.
- עדיף להשתמש בmodel:command-r-plusa בשאילתות.
- הצלחנו "לעלות" על הפעמים שהבוט משתמש בידע הכללי שלו ולהחזיר שלא מצאנו תשובה אצלנו.








בעיות

- הפרויקט מבוסס על טכנולוגיות ונושאים שלא הכרנו לפני כן ועשינו את הפרויקט בלי שום ידע מקדים, בהתחלה זה מאוד הקשה על הבסיס של הפרויקט כי פשוט לא הבנו כל-כך את הקוד עצמו, למרות העזרה של יונתן אנחנו עדין מרגישים שזה עיכב הרבה, בהמשך כבר יותר שלטנו בעניין אך זה עדיין הקשה מאוד שכל נושא היינו צריכים לגשת ללמידה שלו לפני שיכולנו להשתמש בו.
- מודל coheren יצר לנו בעיות עם המודל שלנו לתנך, זה כנראה נובע גם מזה שיש לו הרבה ידע כללי בנושא, וכן עם בעיות שנוצרו במעבר לעברית, שמכמות מסוימת של קבצים זה לא נתן להתבסס רק על rag שלנו.
- כפרויקט מחקרי, היה קשה להציב דדליינים ברורים ומטרות ברורות לתקופות זמן, ולא הייתה כל כך יכולת לצפות ליעדים, ובפרט בשנה כזו בעייתית שהעבודה נקטעה באמצע כמה פעמים.

מבט להמשך

- המבנה של הקוד מוכן, לאחר דיבוגים רבים, ותיקונים שהצלחנו לעשות, לא הצלחנו לסדר לגמרי את הבעייתיות של כמות הקבצים הגדולה יותר, מכאן, אפשר להמשיך לדבג בתקווה למצוא את הבעיה, אך חשבנו שאולי זה כבר הבעיה במודל שלא ניתן לסדר את זה וניתן לשקול החלפת מודל. החלפה כזו אמורה להיות פשוטה שכן מבנה הקוד כבר מוכן ויחסית גנרי ורק צריך להחליף בפונקציות הרלוונטיות למודל החדש.
- חשבנו על רעיון לעבוד עם קבוצות קטנות של קבצים, לבצע שאילתות עליהם, ומאלה להגיע לתשובה על כולם (מעין הפרד ומשול), לא מספיק עבדנו על זה, אך יש התחלה והכנה לגישה כזאת בקובץ nisayon הנמצא גם הוא בgithub.
- ניתן להתייעץ עם מומחים בהמשך ולעבוד בשיתוף איתם.

תוכן הגיט

	bereshit
	README.md
	berelEmbedding.py
	cohereEmbedding.py
	nisayon.py
	story.txt
	storyCohere.py

- התיקיה bereshit מכילה את הקבצים עם הפרקים של ספר בראשית.
- קובץ readme שמסביר בקצרה על אופן ההרצה.
- קבצי קוד berelEmbedding, cohereEmbedding, קבצי ההרצה שלנו עם שתי פונקציות embedding שלנו.
- nisayon : כאמור לעיל, הבסיס לבדיקת הרצה בחלוקה לכמויות קטנות של קבצים.
- story.txt, storyCohere.py ניסיון הרצה על סיפור בעברית וקובץ הרצה למודל העונה על שאלות עליו, נראה שעונה טוב על שאלות על הסיפור (שהרי בהמשך למה שהראנו קודם, כאן אין בעיות עם ידע כללי שיש לבוט על הסיפור, וכן הסיפור הוא באורך קצר בהרבה מספר בראשית).