گزارش پروژه دوم پردازش زبان و گفتار

ملیکا محمدی فخار 99522086

ستاره باباجانی 99521109

احسان احمدپور 99521019

پیاده‌سازی تولید متن با استفاده از بازیابی (RAG) بر روی گزارش‌های سازمانی

RAG یا Retrieval-Augmented Generation یک رویکرد پیشرفته در زمینه هوش مصنوعی است که مدل‌های تولید متن را با سیستم‌های جستجوی اطلاعات ترکیب می‌کند تا پاسخ‌های دقیق‌تری ارائه دهد. این پروژه با هدف ارتقا پاسخ‌دهی به سوالات مرتبط با گزارش‌های سازمانی توسط RAG طراحی شده است. در اینجا مراحل پیاده‌سازی این پروژه را مشاهده می‌کنید:

نصب وابستگی‌ها

ابتدا نیاز داریم کتابخانه‌های مورد نیاز را نصب کنیم:

```
!pip install llama-index llama-index-llms-huggingface llama-index-embeddings-huggingface --quiet
!pip install llama-index-llms-huggingface-api --quiet
! pip install transformers evaluate accelerate --quiet
!pip install -U bitsandbytes --quiet
!pip install -q hazm
!pip install -q clean-text[gpl]
!pip install docx2txt
```

بارگذاری اسناد

```python
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader

document_1 = SimpleDirectoryReader(input_files=["/content/input_1.docx"]).load_data()
document_2 = SimpleDirectoryReader(input_files=["/content/input_2.docx"]).load_data()


def extract_text(documents):
    text = ""
    for doc in documents:
        text += doc.text + " "  # Assuming each document object has a 'text' attribute
    return text.strip()

def chunk_text(text, chunk_size=100):
    words = text.split()
    chunks = [' '.join(words[i:i + chunk_size]) for i in range(0, len(words), chunk_size)]
    return chunks

text_1 = extract_text(document_1)
text_2 = extract_text(document_2)
```

پیش پردازش متون

کتابخانه‌های مورد نیاز

```python
import hazm
from cleantext import clean
import re
```

کتابخانه‌های hazm، cleantext و re برای انجام مراحل مختلف پاک‌سازی متن استفاده می‌شوند. hazm برای نرمال‌سازی متن فارسی، cleantext برای انجام تمیزکاری‌های مختلف روی متن و re برای استفاده از عبارات منظم (Regular Expressions) به کار می‌روند.

تابع پاک‌سازی تگ‌های HTML

```python
def cleanhtml(raw_html):
    cleanr = re.compile('<.*?>')
    cleantext = re.sub(cleanr, '', raw_html)
    return cleantext
```

این تابع برای حذف تگ‌های HTML از یک متن خام استفاده می‌شود. ابتدا یک الگوی Regular Expression تعریف می‌شود که تمام تگ‌های HTML را شامل می‌شود و سپس این تگ‌ها با یک رشته خالی جایگزین می‌شوند.

تابع پاک‌سازی کلی متن

در ابتدا فاصله‌های اضافی از ابتدای و انتهای متن حذف می‌شود.

```
def cleaning(text):
    text = text.strip()
```

این قسمت از کد از کتابخانه cleantext استفاده می‌کند تا متن را به شکل زیر تمیز کند:

fix_unicode=True: کاراکترهای Unicode را تصحیح می‌کند.

to_ascii=False: متن را به ASCII تبدیل نمی‌کند.

lower=True: متن را به حروف کوچک تبدیل می‌کند.

no_line_breaks=True: شکست‌های خط را حذف می‌کند.

no_urls=True: URLها را حذف می‌کند.

no_emails=True: آدرس‌های ایمیل را حذف می‌کند.

no_phone_numbers=True: شماره‌های تلفن را حذف می‌کند.

no_numbers=False: اعداد را حذف نمی‌کند.

no_digits=False: ارقام را حذف نمی‌کند.

no_currency_symbols=True: نمادهای ارزی را حذف می‌کند.

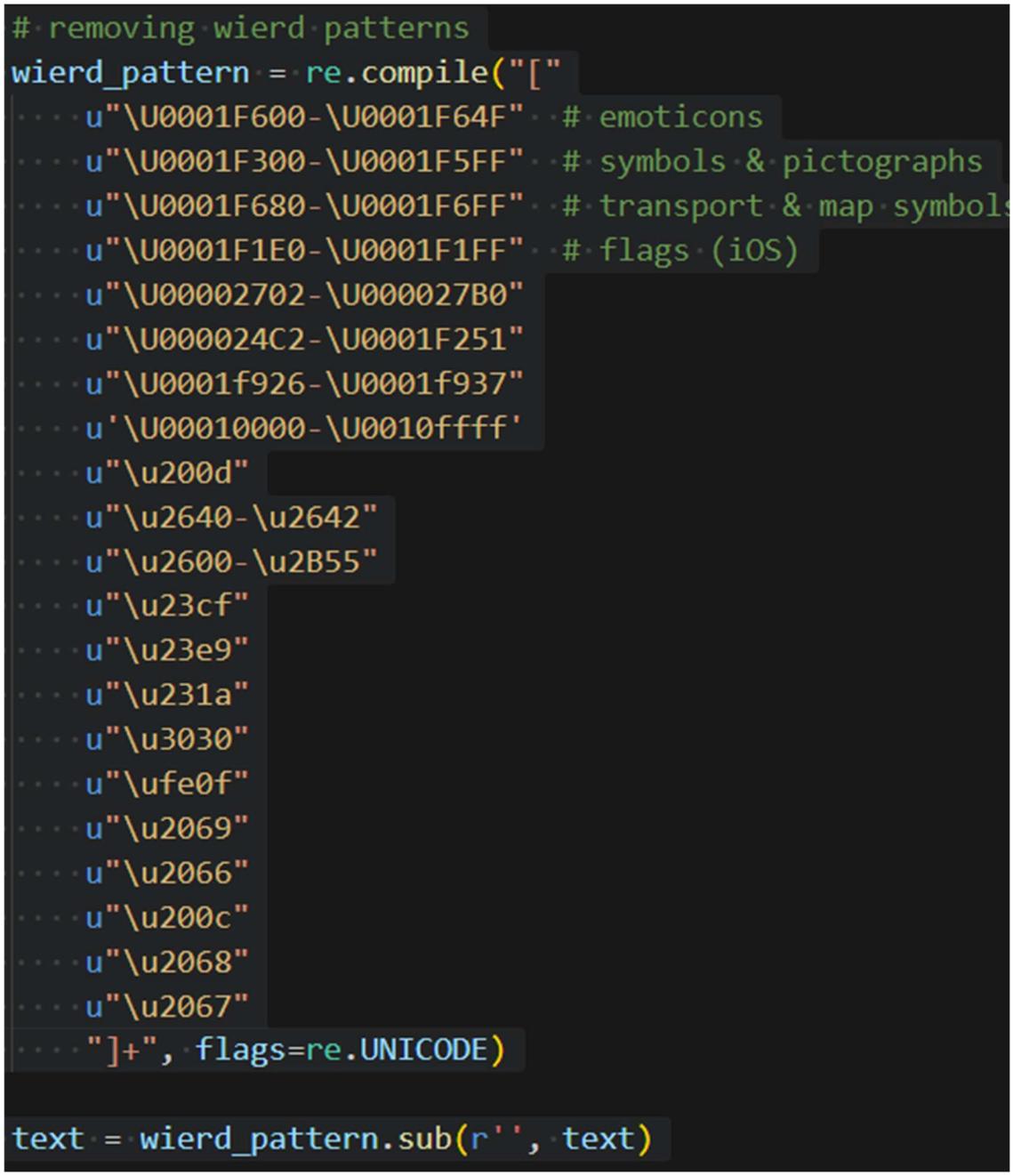
no_punct=False: نشانه‌گذاری‌ها را حذف نمی‌کند.

replace_with_*: جایگزین کردن موارد حذف شده با یک رشته خالی.

```
# regular cleaning
text = clean(text,
    fix_unicode=True,
    to_ascii=False,
    lower=True,
    no_line_breaks=True,
    no_urls=True,
    no_emails=True,
    no_phone_numbers=True,
    no_numbers=False,
    no_digits=False,
    no_currency_symbols=True,
    no_punct=False,
    replace_with_url="",
    replace_with_email="",
    replace_with_phone_number="",
    replace_with_number="",
    replace_with_digit="0",
    replace_with_currency_symbol="",
)
```

نرمال‌سازی متن با استفاده از hazm

```
# normalizing
normalizer = hazm.Normalizer()
text = normalizer.normalize(text)
```



```
# removing wierd patterns
wierd_pattern = re.compile("["
    u"\U0001F600-\U0001F64F"  # emoticons
    u"\U0001F300-\U0001F5FF"  # symbols & pictographs
    u"\U0001F680-\U0001F6FF"  # transport & map symbols
    u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
    u"\U00002702-\U000027B0"
    u"\U000024C2-\U0001F251"
    u"\U0001f926-\U0001f937"
    u'\U00010000-\U0010ffff'
    u"\u200d"
    u"\u2640-\u2642"
    u"\u2600-\u2B55"
    u"\u23cf"
    u"\u23e9"
    u"\u231a"
    u"\u3030"
    u"\ufe0f"
    u"\u2069"
    u"\u2066"
    u"\u200c"
    u"\u2068"
    u"\u2067"
    "]+", flags=re.UNICODE)

text = wierd_pattern.sub(r'', text)
```

این قسمت از کد از یک الگوی Regular Expression استفاده می‌کند تا کاراکترهای نامتعارف و عجیب و غریب مانند ایموجی‌ها و نمادها را حذف کند.

```
# removing extra spaces, hashtags
text = re.sub("#", "", text)
text = re.sub("\s+", " ", text)

return text
```

این قسمت از کد هشتگ‌ها را حذف می‌کند و فاصله‌های اضافی را با یک فاصله جایگزین می‌کند.

استفاده از تابع cleaning برای تمیزکاری متون

```
text_1 = cleaning(text_1)
text_2 = cleaning(text_2)



chunks_2 = chunk_text(text_2, chunk_size=1070)
chunks_2.append(text_1)
```

در این بخش از کد، ابتدا دو متن text_1 و text_2 با استفاده از تابع cleaning تمیز می‌شوند. سپس متن text_2 به بخش‌های کوچکتر تقسیم می‌شود و در نهایت text_1 به انتهای این بخش‌ها اضافه می‌شود.

راه اندازی مدل بازیابی

این کد برای تبدیل متون به بردارهای جملات (sentence embeddings) و ذخیره و بارگذاری این بردارها در Google Drive استفاده می‌شود. در ادامه، مراحل کد را با جزئیات توضیح می‌دهیم:

بارگذاری مدل Sentence Transformer

این خط کد مدل all-MiniLM-L6-v2 را از کتابخانه sentence_transformers بارگذاری می‌کند. این مدل یک مدل فشرده و سریع برای تبدیل جملات به بردارهای جملات است.

```
from transformers import AutoTokenizer, AutoModel
from sentence_transformers import SentenceTransformer
import torch

model = SentenceTransformer('all-MiniLM-L6-v2')
```

تبدیل متون به بردارهای جملات

chunks_2: لیستی از متون که قبلاً تمیز شده و به بخش‌های کوچکتر تقسیم شده است.

model.encode: این متون را به بردارهای جملات تبدیل می‌کند.

convert_to_tensor=True: مشخص می‌کند که خروجی به صورت یک tensor از کتابخانه PyTorch باشد.

```
embeddings = model.encode(chunks_2, convert_to_tensor=True)
```

ذخیره و بارگذاری بردارهای جملات

```
# Mount Google Drive
drive.mount('/content/drive')

save_path = '/content/drive/My Drive/chunk_embeddings.pt'

torch.save(embeddings, save_path)
```

تعریف تابع بازیابی

تعریف تابع find_best_matching_document

```python
def find_best_matching_document(query, model, embeddings, articles):
    query_embedding = model.encode(query, convert_to_tensor=True)
    similarity_scores = util.pytorch_cos_sim(query_embedding, embeddings)[0]
    best_match_index = int(similarity_scores.argmax())
    return articles[best_match_index]
```

این تابع به دنبال یافتن بهترین سند مشابه با پرسش ورودی (query) است.

ورودی‌ها:

Query:پرسش به صورت یک رشته متنی.

Model:مدل sentence_transformers که برای تبدیل متن به بردار جمله استفاده می‌شود.

Embeddings:بردارهای جملات مربوط به مجموعه مقالات یا اسناد.

Articles:مجموعه مقالات یا اسناد به صورت لیستی از رشته‌های متنی.

مراحل:

- تبدیل پرسش به بردار جمله
- محاسبه شباهت کسینوسی
- یافتن شاخص بهترین مطابقت
- استفاده از تابع find_best_matching_document

راه‌اندازی مدل تولید متن

بارگذاری مدل و توکنایزر GPT-2

```python
# GPT-2 Small
model_name = "gpt2"
tokenizer1 = AutoTokenizer.from_pretrained(model_name)
tokenizer1.pad_token = tokenizer1.eos_token
model1 = GPT2LMHeadModel.from_pretrained(model_name)
```

تعریف تابع QA_gpt2

```python
def QA_gpt2(query, article):
    prompt = f"بهترین پاسخ برای سوال روبه‌رو باتوجه به متن داده شده چیست؟ \n سوال: {query}\n\nمتن: {article}"
    inputs = tokenizer1.encode(prompt, return_tensors="pt", max_length=512, truncation=True)
    ids = model1.generate(inputs, max_length=1024, num_return_sequences=1)
    output = tokenizer1.decode(ids[0], skip_special_tokens=True)
    return output
```

این تابع برای تولید پاسخ به یک پرسش بر اساس یک متن داده شده استفاده می‌شود.

ورودی‌ها:

Query: پرسش به صورت یک رشته متنی.

article: متنی که بر اساس آن پاسخ داده می‌شود.

مراحل:

- ساخت پرامپت
- تبدیل پرامپت به توکن‌ها
- تولید پاسخ
- تبدیل توکن‌ها به متن

آزمایش با یک مثال

```
# Test with an example query
query = "چگونه میتوانم از صورت جلسات ماه 5 پرینت بگیرم؟"
best_matching_document = find_best_matching_document(query, model, embeddings, chunks_2)
summary_gpt2 = QA_gpt2(query, best_matching_document)

print(f"{summary_gpt2}")
```

یافتن بهترین سند مشابه، تولید پاسخ با GPT-2

```
for query in random_questions:
    best_matching_document = find_best_matching_document(query, model, embeddings, chunks_2)
    output_gpt2 = QA_gpt2(query, best_matching_document)
    print(f"\n{output_gpt2}")
```

مقدمه

در دنیای کسب و کارهای امروزی، بهره‌گیری از گزارش‌های سازمانی برای استخراج اطلاعات و پاسخ‌دهی به سوالات مختلف بسیار حائز اهمیت است. پروژه حاضر به بررسی و پیاده‌سازی تکنولوژی RAG (Retrieval-Augmented Generation) برای پاسخ‌دهی به سوالات مبتنی بر گزارش‌های سازمانی می‌پردازد. این تکنولوژی ترکیبی از مدل‌های بازیابی اطلاعات و تولید متن است که با استفاده از قدرت مدل‌های زبانی پیشرفته، می‌تواند پاسخ‌های دقیق و کاربردی ارائه دهد. هدف اصلی این پروژه بهبود کیفیت و دقت پاسخ‌ها و همچنین افزایش کارایی در استفاده از گزارش‌های سازمانی است.

پیش‌زمینه تکنولوژیکی

در این پروژه از ترکیب تکنولوژی‌های بازیابی اطلاعات و مدل‌های زبانی پیشرفته استفاده شده است. مدل‌های زبانی نظیر GPT-2 و Sentence-BERT برای تحلیل و پردازش زبان طبیعی به کار گرفته شده‌اند.

GPT-2: یک مدل زبانی تولیدی است که می‌تواند متن را بر اساس ورودی‌های داده شده تولید کند.

Sentence-BERT: یک مدل برای تولید بردارهای جملات که به منظور محاسبه شباهت معنایی بین جملات استفاده می‌شود.

این مدل‌ها با سیستم‌های بازیابی اطلاعات ترکیب شده‌اند تا بتوانند اسناد مرتبط با پرسش را بازیابی کنند و سپس بر اساس آن‌ها پاسخ‌های دقیق تولید نمایند.

روند پیاده‌سازی

مراحل پیاده‌سازی تکنولوژی RAG در این پروژه شامل موارد زیر است:

پیش‌پردازش داده‌ها: شامل پاک‌سازی و نرمال‌سازی متن‌ها برای آماده‌سازی ورودی مدل‌ها.

مدل‌سازی:

استفاده از Sentence-BERT برای تولید بردارهای جملات و محاسبه شباهت بین پرسش و بخش‌های مختلف گزارش‌ها.

استفاده از GPT-2 برای تولید پاسخ بر اساس متن بازیابی شده.

ادغام سیستم‌های بازیابی و تولید متن:

بازیابی اسناد مرتبط با پرسش از طریق مدل‌های بازیابی اطلاعات.

تولید پاسخ با استفاده از مدل GPT-2 بر اساس اسناد بازیابی شده.

ارزیابی و بهبود: ارزیابی عملکرد سیستم و بهبود مدل‌ها بر اساس بازخوردها و نتایج به دست آمده.

چالش‌ها و راه‌حل‌ها

محدودیت GPU در Google Colab: یکی از چالش‌های اصلی، محدودیت منابع پردازشی در Google Colab بود. برای غلبه بر این محدودیت، از

تکنیک‌های بهینه‌سازی مدل‌ها و کاهش حجم پردازش‌ها استفاده شد. همچنین، فرآیندها به بخش‌های کوچک‌تر تقسیم شده و به صورت موازی اجرا شدند.

نتایج و بررسی‌ها

نتایج نشان داد که استفاده از تکنولوژی RAG منجر به بهبود قابل توجهی در کیفیت پاسخ‌ها شد. ارزیابی‌ها با استفاده از معیارهای مختلف نشان داد که دقت و کارایی پاسخ‌ها نسبت به رویکردهای قبلی بهبود یافته است.

کیفیت پاسخ‌ها: با استفاده از معیارهایی نظیر دقت (precision) و بازیابی(recall) ، پاسخ‌های تولید شده ارزیابی شدند و نتایج نشان داد که کیفیت پاسخ‌ها به طور متوسط بهبود یافته است.

سرعت پردازش: بهینه‌سازی‌های انجام شده منجر به کاهش زمان پردازش نسبت به رویکردهای سنتی شد.

موانع و چالش‌ها

در طول پروژه، تیم با چندین چالش مواجه شد:

محدودیت‌های فنی: محدودیت GPU در Google Colab یکی از مشکلات اصلی بود که با استفاده از تکنیک‌های بهینه‌سازی و تقسیم فرآیندها به بخش‌های کوچک‌تر برطرف شد.

مسائل مربوط به داده‌ها: تنوع و کیفیت داده‌های ورودی یکی دیگر از چالش‌ها بود. برای حل این مشکل، داده‌ها به دقت پیش‌پردازش و نرمال‌سازی شدند.

پیچیدگی مدل‌ها: ترکیب مدل‌های بازیابی و تولید متن نیازمند هماهنگی دقیق بین آن‌ها بود که با طراحی مناسب و تست‌های مکرر به دست آمد.

آموخته‌ها و پیشنهادات

از اجرای این پروژه چندین درس مهم آموخته شد:

اهمیت پیش‌پردازش داده‌ها: کیفیت داده‌های ورودی تأثیر مستقیمی بر عملکرد مدل‌ها دارد.

بهینه‌سازی منابع: استفاده بهینه از منابع پردازشی و مدیریت محدودیت‌ها می‌تواند به بهبود کارایی سیستم کمک کند.

تست مداوم: ارزیابی مداوم مدل‌ها و سیستم‌ها برای اطمینان از عملکرد مناسب آن‌ها ضروری است.

پیشنهاد می‌شود تیم‌های آینده که قصد اجرای پروژه‌های مشابه را دارند:

بهینه‌سازی مستمر: مدل‌ها و فرآیندها را به طور مداوم بهینه‌سازی کنند.

استفاده از منابع پردازشی قوی‌تر: برای پروژه‌های بزرگ‌تر، استفاده از منابع پردازشی قوی‌تر و پایدارتر توصیه می‌شود.

توجه به کیفیت داده‌ها: پیش‌پردازش دقیق داده‌ها و تضمین کیفیت آن‌ها اهمیت زیادی دارد.

منابع

کتابخانه‌ها و ابزارها:

PyTorch

transformers

Sentence-BERT

مقالات و وب‌سایت‌ها:

مقاله‌های مرتبط با مدل‌های GPT-2 وSentence-BERT

مستندات رسمی کتابخانه‌های transformers وPyTorch