NLP Project 1
TripAdvisor Recommendation Challenge - Beating BM25

Léo RINGEISSEN & Santiago MARTIN

Google Colab Link

Table of Contents

Objective

The primary objective is to develop a recommendation system based on TripAdvisor reviews for hotels. The goal is to be able to query the model by feeding it the compilation of one hotel's reviews, and then getting a recommendation for the hotel most similar to the queried one. The completion of this objective contains several parts

First, we'll build this recommendation system out of a baseline BM25 model, with no tuning and without performing traditional NLP preprocessing techniques on the training and input data. And second, we'll test out different improvements to try to beat the initial baseline BM25 model, hence the challenge. This will be done by applying those previously mentioned preprocessing techniques and by testing other NLP models like TF-IDF, BERT, and T5 (small and baseline versions.

Recommendation system functionality

Training

We're training the models on a corpus, which contains the compilations of all the reviews left by users for each hotel. So, each element of the list is one long string that contains every review stitched together for one hotel. This corpus is what **WILL NOT** be preprocessed using traditional NLP techniques initially for the baseline BM25 model and **WILL** be for attempting to beat the baseline BM25.

Evaluation

To evaluate the models, we'll use the ratings of the hotels. Each hotel in our final preprocessed dataset contains the average score for all 7 of the following rating categories: "service", "cleanliness", "overall", "value", "location", "sleep quality", "rooms". To test the quality of the recommendations from a model, we perform several steps:

- Firstly, we query it 100 times with 100 hotels chosen at random (for each model we use the same random batch to ensure a fair comparison)
- Secondly, for each query we compute the Mean Squared Error of the difference between the 7 respective ratings of the queried hotel and the recommended hotel
- Finally, we compute the average the MSE over the 100 queries to obtain a final result to judge the quality of the model's recommendations

Data preparation

Baseline Preprocessing

This section details the preparations made on the original data which was applied for all models.

We initially disposed of an "offerings" dataset and a "reviews" dataset. "Offering" contained information for each hotel (name, location, ID, etc…), whereas "reviews" contained each review for all of the hotels, along with the ratings associated to that review and the ID of the hotel being reviewed.

The "offerings" dataset was not very useful since we're merely interested in the reviews and ratings and the "reviews" dataset already contains a unique hotel identifier column. As a result, we only utilize "reviews".

As mentioned previously, our final reviews table contained one line per hotel, each containing a compilation of the reviews for that hotel and an average aggregate of its ratings, to transform the initial reviews dataset, we had to perform several steps:

- Remove all unnecessary columns
- Extract ratings dictionary from ratings column based on our 7 categories of interest ("service", "cleanliness", "overall", "value", "location", "sleep quality", "rooms") and turn them into columns for each review
- Summarize the table by hotel using the unique hotel identifier in each review
    - For this we compiled each review for one hotel as one long string
    - And we aggregated the ratings into averages of each of the 7 categories

	id	name	service	cleanliness	overall	value	location	sleep_quality	rooms	text	num_reviews
0	72572	BEST WESTERN PLUS Pioneer Square Hotel	4.601010	4.636364	4.388889	4.323232	4.570707	4.333333	4.282828	I had to make fast visit to seattle and I foun...	198
1	72579	BEST WESTERN Loyal Inn	4.232000	4.240000	3.888000	4.152000	4.192000	3.768000	3.856000	Great service, rooms were clean, could use som...	125
2	72586	BEST WESTERN PLUS Executive Inn	4.250000	4.287879	4.045455	4.053030	4.537879	4.113636	3.992424	Beautiful views of the space needle - especial...	132
3	72598	Comfort Inn & Suites Seattle	3.243243	3.243243	2.918919	3.054054	3.027027	3.270270	3.189189	This hotel is in need of some serious updates....	37
4	73236	Days Inn San Antonio/Near Lackland AFB	4.277778	3.111111	3.388889	3.777778	4.111111	3.722222	3.222222	My experience at this days inn was perfect. th...	18

Preprocessed table, with 7 ratings and text containing all reviews

We then created a corpus, which is the what we'll feed the algorithms to train.

Initialization

```
corpus = grouped_df['text'].tolist()
tokenized_corpus = [doc.split(" ") for doc in corpus]
```

Querying on N documents

```
import random

n_queries = 100 # For testing, we will use 10 queries
random.seed(42)
query_ids = random.sample(range(len(corpus)), n_queries)
print(query_ids)
```

```
[2619, 456, 102, 3037, 1126, 1003, 914, 571, 3016, 419, 2771, 3033, 3654, 2233, 356, 2418, 1728, 130, 122
```

Creation of corpus

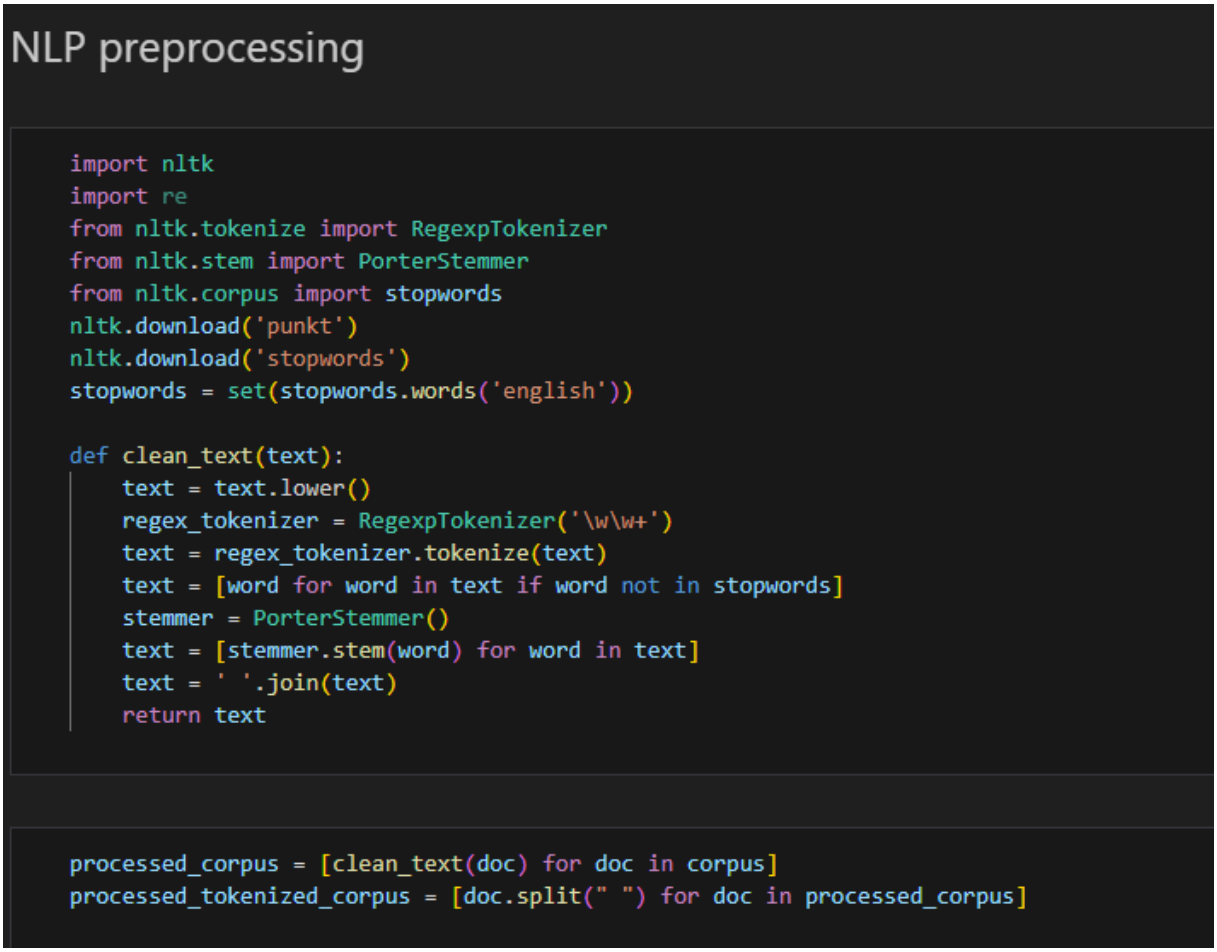
NLP preprocessing

This section details the data preprocessing techniques used to improve upon the initial baseline BM25 model.

The traditional NLP preprocessing techniques we reviewed in class are the following:

- Making the strings lowercase
- Tokenization
- Removing stop words
- Stemming or Lemmatization

These are the same techniques we applied to the compiled reviews of each hotel. In the end, we had two corpuses which we could use to train our models, a standard one containing the untouched compilation of reviews, and a preprocessed one. The preprocessed corpus wasn't just used for improving upon the BM25 itself, but also to test the different results with other models by running them on original and preprocessed data.



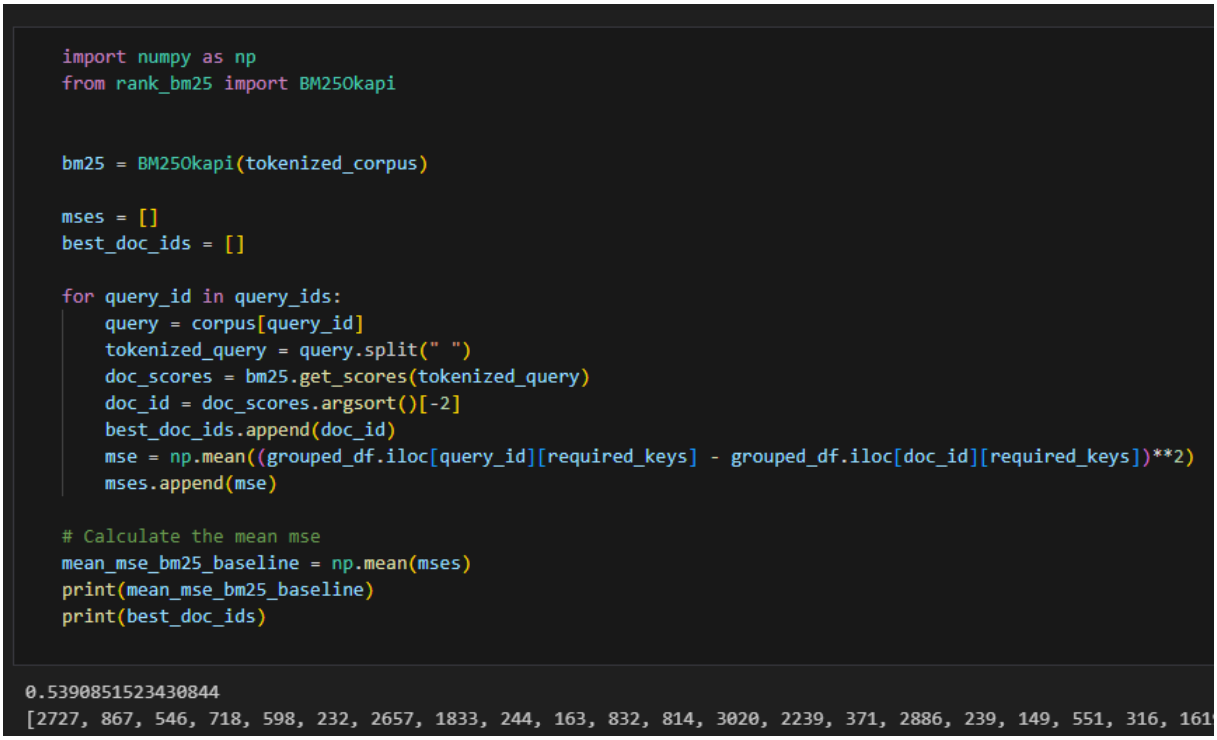
NLP preprocessing, with all methods grouped in one function

Models

Over the course of this work, we ran a mix of standard and preprocessed data through a variety of models: BM25, TF-IDF, BERT, T5-small, and T5-base. These are the results for each.

BM25 Baseline

The BM25 model is a very powerful model, and even without using the NLP preprocessed data it yielded some of the best results we saw from our entire project. Its final score after 100 queries was 0.54.



```
import numpy as np
from rank_bm25 import BM25Okapi


bm25 = BM25Okapi(tokenized_corpus)

mses = []
best_doc_ids = []

for query_id in query_ids:
    query = corpus[query_id]
    tokenized_query = query.split(" ")
    doc_scores = bm25.get_scores(tokenized_query)
    doc_id = doc_scores.argsort()[-2]
    best_doc_ids.append(doc_id)
    mse = np.mean((grouped_df.iloc[query_id][required_keys] - grouped_df.iloc[doc_id][required_keys])**2)
    mses.append(mse)

# Calculate the mean mse
mean_mse_bm25_baseline = np.mean(mses)
print(mean_mse_bm25_baseline)
print(best_doc_ids)
```

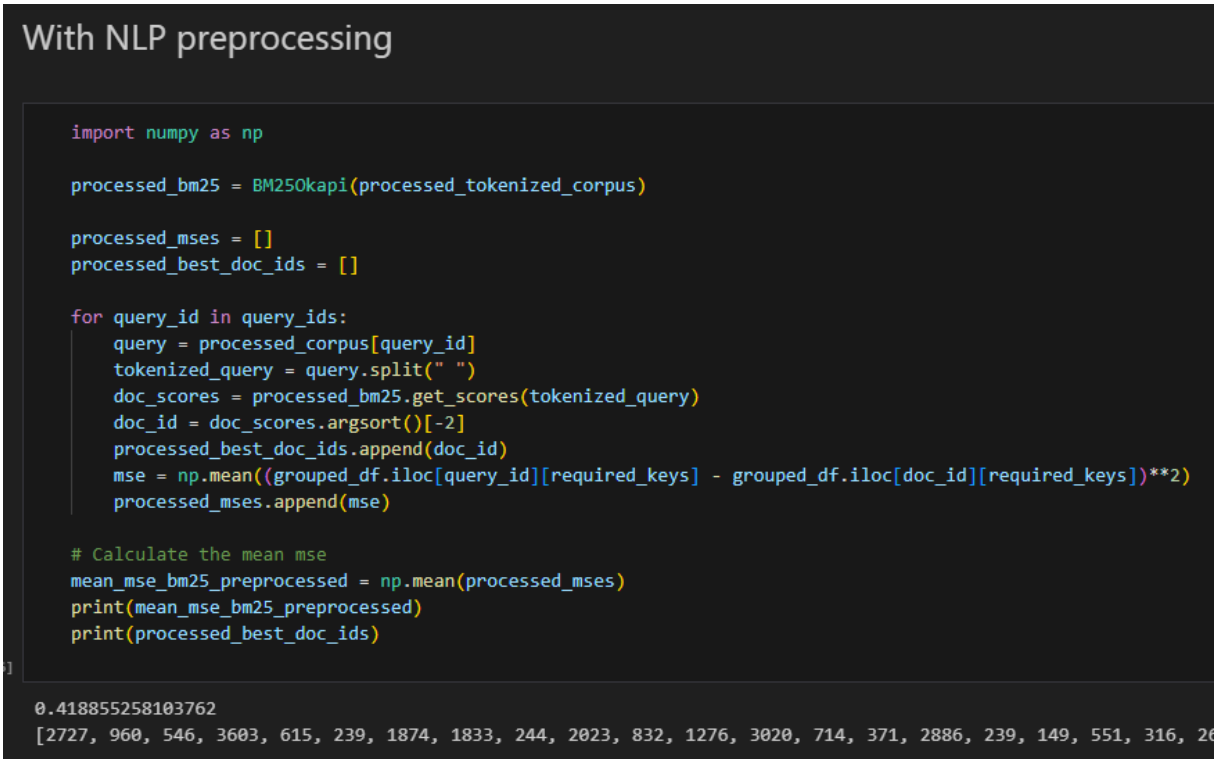
```
0.5390851523430844
[2727, 867, 546, 718, 598, 232, 2657, 1833, 244, 163, 832, 814, 3020, 2239, 371, 2886, 239, 149, 551, 316, 161
```

BM25 baseline

BM25 with preprocessed data

Logically, using NLP preprocessed data to train the BM25 model made it even better and this gave us the best result of the entire project. After 100 queries we obtained a score of 0.42.

Both algorithms are the exact same except we used processed_corpus instead of corpus.

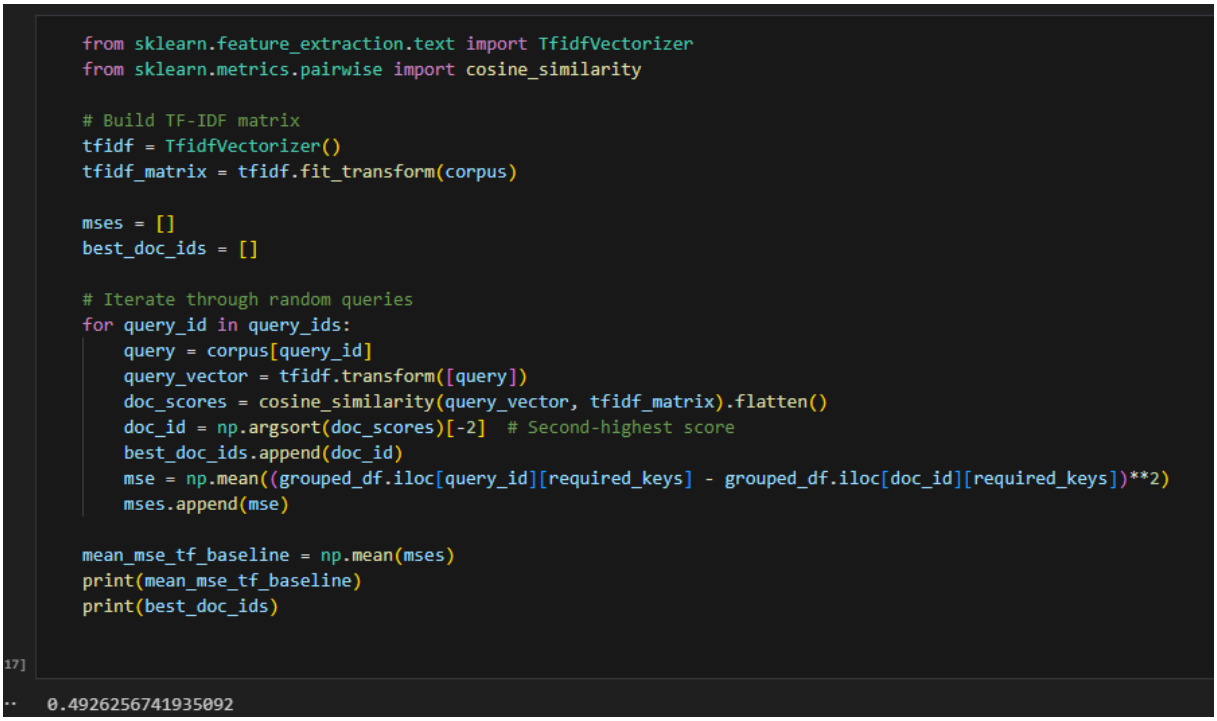


BM25 with preprocessed corpus

TF-IDF baseline

This model was surprisingly powerful. While not even using the NLP preprocessed data, it yielded a better result than the baseline BM25 model, which we were not expecting. Its final score was 0.49.

One of this model's advantages is also it's efficiency in terms of computing time, which was less than 10 minutes.



```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Build TF-IDF matrix
tfidf = TfidfVectorizer()
tfidf_matrix = tfidf.fit_transform(corpus)

mses = []
best_doc_ids = []

# Iterate through random queries
for query_id in query_ids:
    query = corpus[query_id]
    query_vector = tfidf.transform([query])
    doc_scores = cosine_similarity(query_vector, tfidf_matrix).flatten()
    doc_id = np.argsort(doc_scores)[-2]  # Second-highest score
    best_doc_ids.append(doc_id)
    mse = np.mean((grouped_df.iloc[query_id][required_keys] - grouped_df.iloc[doc_id][required_keys])**2)
    mses.append(mse)

mean_mse_tf_baseline = np.mean(mses)
print(mean_mse_tf_baseline)
print(best_doc_ids)
```

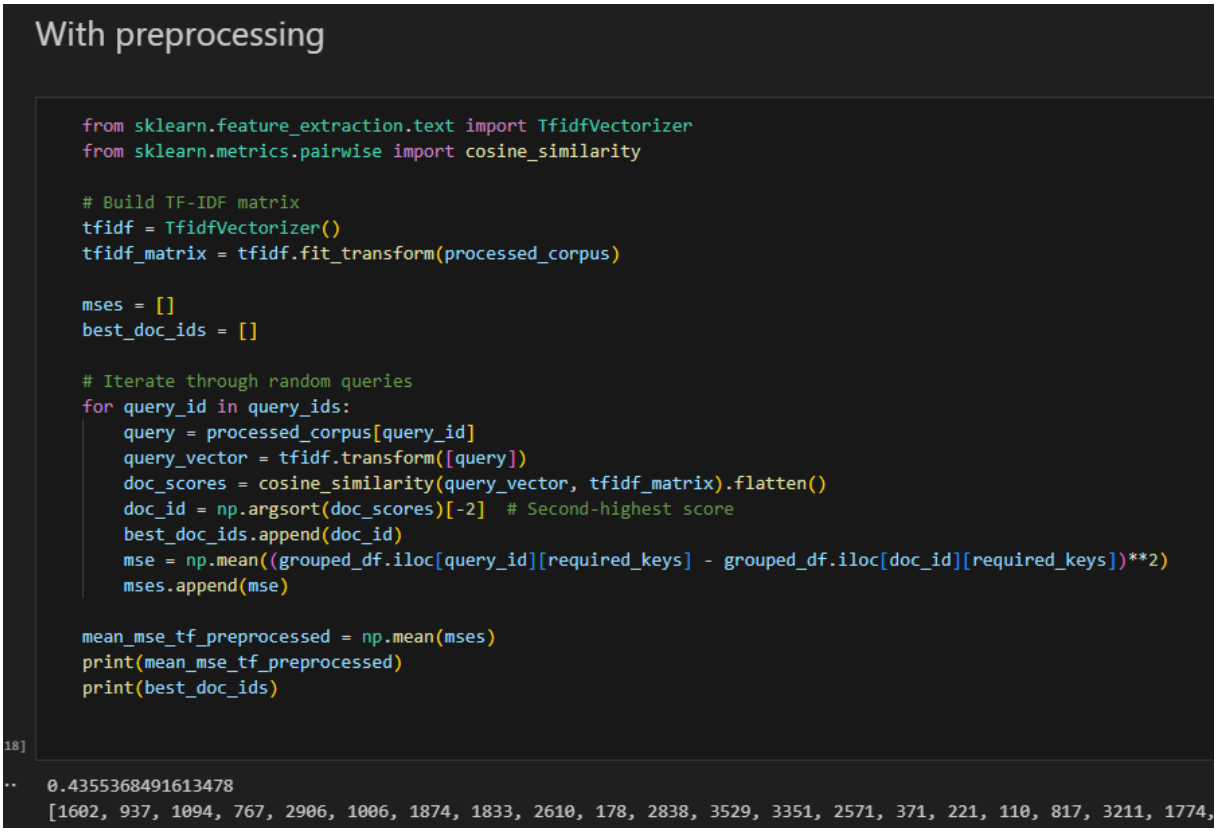
```
17]

··  0.4926256741935092
```

TF-IDF baseline

TF-IDF with preprocessed data

This already strong model only got slightly better upon training it with the preprocessed data, merely improving to 0.44, which is still the second-best score we obtained throughout the whole project.

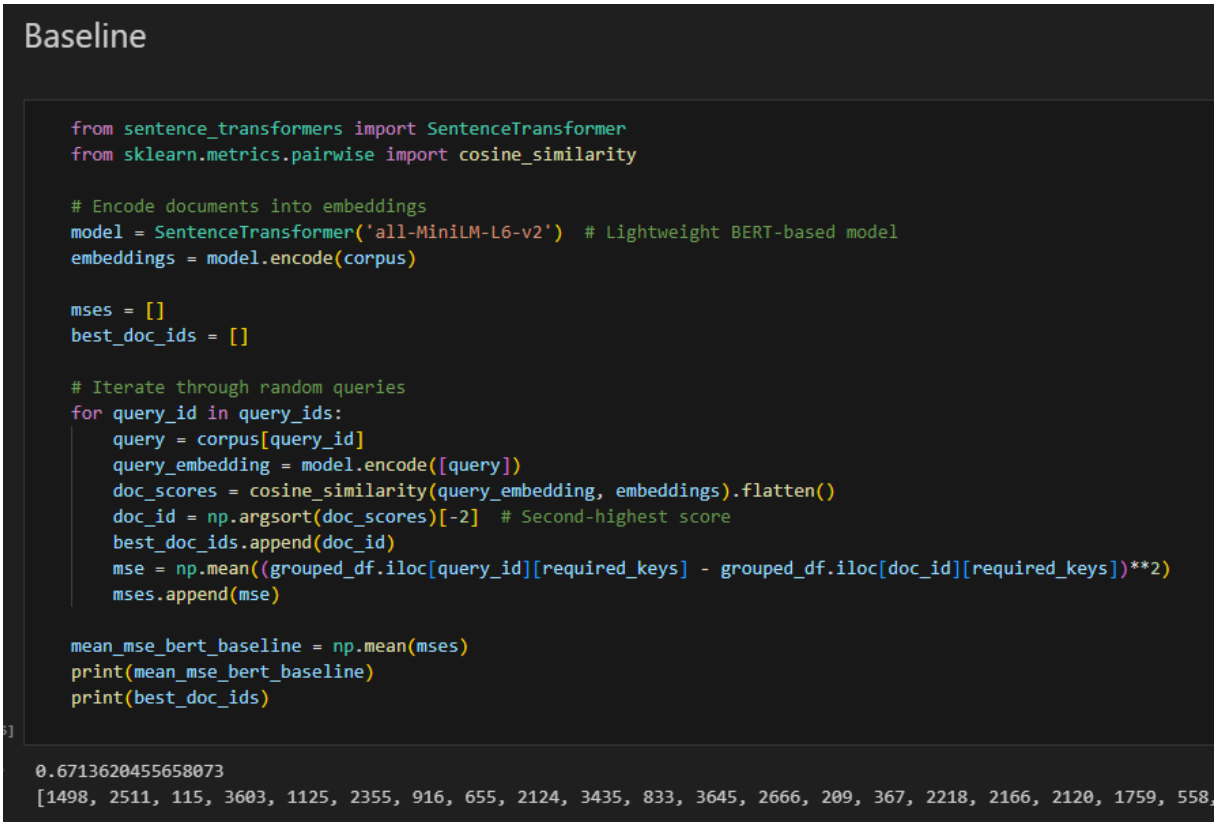


TF-IDF with preprocessed corpus

BERT baseline

The BERT baseline model yielded the worst results on this project, with a MSE of 0.67. This shows the bidirectional approach it uses might not be the best suited for this case.

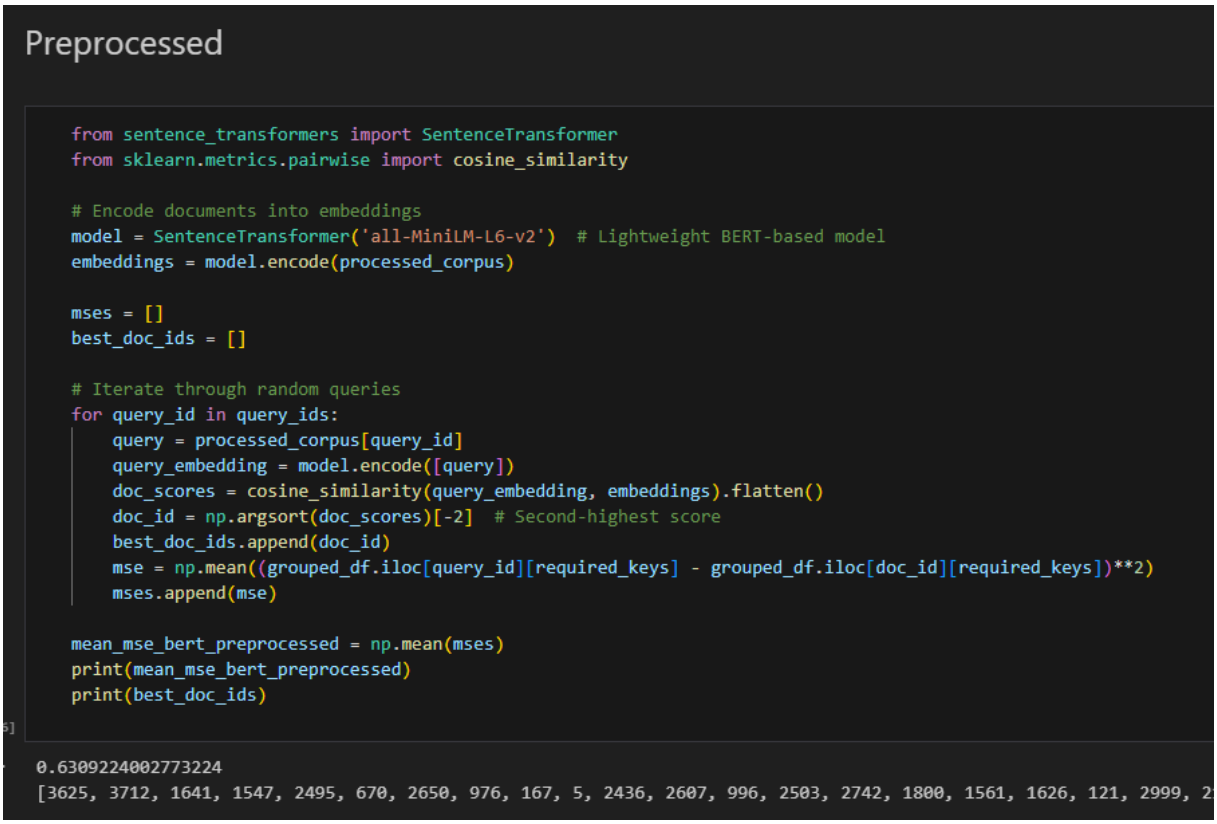
The only advantage of this approach is the computing time that was below 10 minutes.



BERT baseline

BERT with preprocessed data

In this case too, the preprocessing helped us get better results, which was still not enough given the MSE was 0.63, which is second worst to the baseline BERT mode. This highlights both the benefits of NLP preprocessing, and the difficulties the BERT model had to adapt to this problem.



BERT with preprocessed corpus

T5-small baseline

The T5 model ended up not being as powerful for this task as we had anticipated. It seemed like a well-suited candidate based on what we learned during TD3 of Machine Learning for NLP. The result of the baseline T5-small model yielded one of the worst results we saw at 0.63. We noticed that there are different variants of T5 (small, base, large, etc...), so opted to try with a larger size, while staying without our computational power limits.

```
from transformers import T5Tokenizer, T5Model
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
import random
import sentencepiece
import torch
```

```
WARNING:tensorflow:From c:\Users\ringi_3xz04z7\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\losse
```

```
from transformers import T5Tokenizer, T5Model
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
import random
import sentencepiece
import torch

# Load T5 model and tokenizer
model_name = "t5-small"  # Change to "t5-base" or "t5-large" for better results
tokenizer = T5Tokenizer.from_pretrained(model_name)
model = T5Model.from_pretrained(model_name)

# Ensure reproducibility
random.seed(42)
np.random.seed(42)

# Function to encode text into embeddings using T5
def encode_text(text, max_length=512):
    # Tokenize and prepare inputs
    inputs = tokenizer(text, return_tensors="pt", max_length=max_length, truncation=True, padding="max_length")
    # Pass inputs through T5 encoder to get embeddings
    with torch.no_grad():
        outputs = model.encoder(**inputs)
    # Mean pooling of the token embeddings
    return outputs.last_hidden_state.mean(dim=1).squeeze().numpy()

# Preprocess corpus to get embeddings
print("Encoding documents into embeddings...")
document_embeddings = np.array([encode_text(doc) for doc in corpus])
```

```
mses = []
best_doc_ids = []

# Iterate through random queries
print("Processing queries...")
for query_id in query_ids:
    query = corpus[query_id]
    query_embedding = encode_text(query)

    # Compute cosine similarities
    doc_scores = cosine_similarity([query_embedding], document_embeddings).flatten()

    # Get the second-highest score
    doc_id = np.argsort(doc_scores)[-2]  # Second-highest score
    best_doc_ids.append(doc_id)

    # Calculate MSE for evaluation
    mse = np.mean((grouped_df.iloc[query_id][required_keys] - grouped_df.iloc[doc_id][required_keys])**2)
    mses.append(mse)

# Calculate the mean MSE
mean_mse_t5s_baseline = np.mean(mses)
print(f"Mean MSE: {mean_mse_t5s_baseline}")
print("Best doc IDs:", best_doc_ids)
```

```
Processing queries...
Mean MSE: 0.6340350221023886
```

T5-small with preprocessed data

Before testing on a larger T5 model, we tried feeding the NLP preprocessed data to the T5-small model and the results were only slightly better, but still nowhere near our best result. The final score ended up at 0.58.

Preprocessed

```
# Load T5 model and tokenizer
model_name = "t5-small"  # Change to "t5-base" or "t5-large" for better results
tokenizer = T5Tokenizer.from_pretrained(model_name)
model = T5Model.from_pretrained(model_name)

# Ensure reproducibility
random.seed(42)
np.random.seed(42)

# Function to encode text into embeddings using T5
def encode_text(text, max_length=512):
    # Tokenize and prepare inputs
    inputs = tokenizer(text, return_tensors="pt", max_length=max_length, truncation=True, padding="max_length")
    # Pass inputs through T5 encoder to get embeddings
    with torch.no_grad():
        outputs = model.encoder(**inputs)
    # Mean pooling of the token embeddings
    return outputs.last_hidden_state.mean(dim=1).squeeze().numpy()

# Preprocess corpus to get embeddings
print("Encoding documents into embeddings...")
document_embeddings = np.array([encode_text(doc) for doc in processed_corpus])
```

```
Encoding documents into embeddings...
```

```
mses = []
best_doc_ids = []

# Iterate through random queries
print("Processing queries...")
for query_id in query_ids:
    query = processed_corpus[query_id]
    query_embedding = encode_text(query)

    # Compute cosine similarities
    doc_scores = cosine_similarity([query_embedding], document_embeddings).flatten()

    # Get the second-highest score
    doc_id = np.argsort(doc_scores)[-2]  # Second-highest score
    best_doc_ids.append(doc_id)

    # Calculate MSE for evaluation
    mse = np.mean((grouped_df.iloc[query_id][required_keys] - grouped_df.iloc[doc_id][required_keys])**2)
    mses.append(mse)

# Calculate the mean MSE
mean_mse_t5s_preprocessed = np.mean(mses)
print(f"Mean MSE: {mean_mse_t5s_preprocessed}")
print("Best doc IDs:", best_doc_ids)
```

```
Processing queries...
Mean MSE: 0.5792905873745835
Best doc IDs: [2242, 1123, 406, 2810, 2955, 2438, 1446, 2343, 167, 45, 1155, 2084, 3564, 663, 639, 3169, 1604, 1
```

T5-small with preprocessed corpus

T5-base baseline

We tried using the T5 base model, which is significantly heavier than the small one. It did perform better than the small version, but the result are still not that great compared to BMTF-IDF for example. The best result we got on the model with no preprocessed data was a 0.55 MSE.

T5 Base

```
[18]
from transformers import T5Tokenizer, T5Model
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
import random
import sentencepiece
import torch

# Load T5 model and tokenizer
model_name = "t5-base"  # Change to "t5-base" or "t5-large" for better results
tokenizer = T5Tokenizer.from_pretrained(model_name)
model = T5Model.from_pretrained(model_name)

# Ensure reproducibility
random.seed(42)
np.random.seed(42)

# Function to encode text into embeddings using T5
def encode_text(text, max_length=512):
    # Tokenize and prepare inputs
    inputs = tokenizer(text, return_tensors="pt", max_length=max_length, truncation=True, padding="max_length")
    # Pass inputs through T5 encoder to get embeddings
    with torch.no_grad():
        outputs = model.encoder(**inputs)
    # Mean pooling of the token embeddings
    return outputs.last_hidden_state.mean(dim=1).squeeze().numpy()

# Preprocess corpus to get embeddings
print("Encoding documents into embeddings...")
document_embeddings = np.array([encode_text(doc) for doc in corpus])
```

```
mses = []
best_doc_ids = []

# Iterate through random queries
print("Processing queries...")
for query_id in query_ids:
    query = processed_corpus[query_id]
    query_embedding = encode_text(query)

    # Compute cosine similarities
    doc_scores = cosine_similarity([query_embedding], document_embeddings).flatten()

    # Get the second-highest score
    doc_id = np.argsort(doc_scores)[-2]  # Second-highest score
    best_doc_ids.append(doc_id)

    # Calculate MSE for evaluation
    mse = np.mean((grouped_df.iloc[query_id][required_keys] - grouped_df.iloc[doc_id][required_keys])**2)
    mses.append(mse)

# Calculate the mean MSE
mean_mse_t5b_preprocessed = np.mean(mses)
print(f"Mean MSE: {mean_mse_t5b_preprocessed}")
print("Best doc IDs:", best_doc_ids)
]
```

```
Processing queries...
Mean MSE: 0.49376989779975416
Best doc IDs: [2242, 1299, 3096, 1293, 216, 1918, 1108, 958, 808, 1941, 2119, 3407, 382, 3153, 3452, 2254, 3021,
```

T5-base baseline

T5-base with preprocessed data

Finally, we saw improvement with the larger T5-base model, which took a fairly long time to run. It logically performed better than its clone that was merely trained on the standard data. In the end it yielded a result close to our best at 0.49.

```
from transformers import T5Tokenizer, T5Model
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
import random
import sentencepiece
import torch

# Load T5 model and tokenizer
model_name = "t5-base"  # Change to "t5-base" or "t5-large" for better results
tokenizer = T5Tokenizer.from_pretrained(model_name)
model = T5Model.from_pretrained(model_name)

# Ensure reproducibility
random.seed(42)
np.random.seed(42)

# Function to encode text into embeddings using T5
def encode_text(text, max_length=512):
    # Tokenize and prepare inputs
    inputs = tokenizer(text, return_tensors="pt", max_length=max_length, truncation=True, padding="max_length")
    # Pass inputs through T5 encoder to get embeddings
    with torch.no_grad():
        outputs = model.encoder(**inputs)
    # Mean pooling of the token embeddings
    return outputs.last_hidden_state.mean(dim=1).squeeze().numpy()

# Preprocess corpus to get embeddings
print("Encoding documents into embeddings...")
document_embeddings = np.array([encode_text(doc) for doc in processed_corpus])
```

[20]

```
Encoding documents into embeddings...
```

```
mses = []
best_doc_ids = []

# Iterate through random queries
print("Processing queries...")
for query_id in query_ids:
    query = processed_corpus[query_id]
    query_embedding = encode_text(query)

    # Compute cosine similarities
    doc_scores = cosine_similarity([query_embedding], document_embeddings).flatten()

    # Get the second-highest score
    doc_id = np.argsort(doc_scores)[-2]  # Second-highest score
    best_doc_ids.append(doc_id)

    # Calculate MSE for evaluation
    mse = np.mean((grouped_df.iloc[query_id][required_keys] - grouped_df.iloc[doc_id][required_keys])**2)
    mses.append(mse)

# Calculate the mean MSE
mean_mse_t5b_preprocessed = np.mean(mses)
print(f"Mean MSE: {mean_mse_t5b_preprocessed}")
print("Best doc IDs:", best_doc_ids)
```

1]

```
Processing queries...
Mean MSE: 0.49376989779975416
Best doc IDs: [2242, 1299, 3096, 1293, 216, 1918, 1108, 958, 808, 1941, 2119, 3407, 382, 3153, 3452, 2254, 3021,
```

T5-base with preprocessed corpus

Results summary

	BM25	TF-IDF	BERT	T5-small	T5-base
Standard data	0.54	0.49	0.67	0.63	0.55
Preprocessed data	0.42	0.44	0.63	0.58	0.49
Computing time	~1hr	<10min	<10min	~30min	~1hr30min

Conclusion

This project showed us that at the beginning of a project, it can be quite unclear what models are best adapted to the objective, and how you can best utilize them. For this reason, it was very interesting getting to experiment with a wide variety of models, with different levels of data processing. One thing we noticed is that other than the BM25 model, most of the models we tested didn't improve by such a large margin when trained on the NLP preprocessed data. After doing some research on the potential causes, we learned that for certain models, processes like stemming and the removal of stop words can actually reduce the quality of the model. This is because these models are built to understand the complex contexts of word placements, so simplifying the input data through stemming, or even limiting its context through the removal of stop words, won't necessarily be as beneficial as one might expect.

The big winner of this project is the BM25 model trained on preprocessed data. We indeed managed to beat the BM25 baseline, however it was only by improving the data we trained it with. Although some models like TF-IDF and T5-base showed promise and got close to our eventual best result.

Another important aspect to take into consideration when working with such models can be the computing time. In this project, the only objective was to optimize results regardless of computing time, but if we had to find the best balance between performance and efficiency, then TF-IDF would then be the best choice.