Lab 4 - Cross-encoder re-ranking

In [1]:
```
from helper_utils import load_chroma, word_wrap, project_embedding
from chromadb.utils.embedding_functions import SentenceTransformerE
import numpy as np
```

In [2]:
```
embedding_function = SentenceTransformerEmbeddingFunction()

chroma_collection = load_chroma(filename='microsoft_annual_report_2
chroma_collection.count()
```

.gitattributes: 100% 1.18k/1.18k [00:00<00:00, 93.2kB/s]

1_Pooling/config.json: 100% 190/190 [00:00<00:00, 21.0kB/s]

README.md: 100% 10.6k/10.6k [00:00<00:00, 1.34MB/s]

config.json: 100% 612/612 [00:00<00:00, 76.0kB/s]

config_sentence_transformers.json: 100% 116/116 [00:00<00:00, 13.6kB/s]

data_config.json: 100% 39.3k/39.3k [00:00<00:00, 4.54MB/s]

pytorch_model.bin: 100% 90.9M/90.9M [00:01<00:00, 73.2MB/s]

sentence_bert_config.json: 100% 53.0/53.0 [00:00<00:00, 5.97kB/s]

special_tokens_map.json: 100% 112/112 [00:00<00:00, 13.9kB/s]

tokenizer.json: 100% 466k/466k [00:00<00:00, 2.60MB/s]

tokenizer_config.json: 100% 350/350 [00:00<00:00, 43.1kB/s]

train_script.py: 100% 13.2k/13.2k [00:00<00:00, 1.72MB/s]

vocab.txt: 100% 232k/232k [00:00<00:00, 1.97MB/s] modules.json: 100% 349 349/349 [00:00<00:00, 33.1kB/s]

Re-ranking the long tail

Now as we have long tailed documents that are retrieved as relavent documents. We will use Cross Encode from Sentence Encoder to rank each document according to query:

What is Cross Encoder and How it performs this Ranking thing ? So Basically Sentence Encoder have two types of encoders:

```
    1: Bi-Encoder:
       This type of Encoder takes the two Queries seprately and pass them
from seprate encoder and then perfoms cosine similarity to give the relev
ancy among both.

    2: Cross Encoder:
       This type of encoder process inputs (we are considering 2) togethe
r as single unit. This allows the model to directly compare and contrast
the inputs and understand their relation in better way, also in the end i
t returns the score from 0 to 1 that shows the similarity among both :)
```

-Now in short we use Cross Encoder as Ranking thing by giving it Query and Retrived Documents one by one and in the end we will get score of each document representing how much query is related to that specified document.

In [3]:
```
query = "What has been the investment in research and development?'
results = chroma_collection.query(query_texts=query, n_results=10,

retrieved_documents = results['documents'][0]

for document in results['documents'][0]:
    print(word_wrap(document))
    print('')
```

```
• operating expenses increased $ 1. 5 billion or 14 % driven by
investments in gaming, search and news advertising, and windows
marketing. operating expenses research and development ( in million
s,
except percentages ) 2022 2021 percentage change research and
development $ 24, 512 $ 20, 716 18 % as a percent of revenue 12 % 12
%
0ppt research and development expenses include payroll, employee
benefits, stock - based compensation expense, and other headcount -
related expenses associated with product development. research and
development expenses also include third - party development and
programming costs, localization costs incurred to translate software
for international markets, and the amortization of purchased softwar
e
code and services content. research and development expenses increas
ed
$ 3. 8 billion or 18 % driven by investments in cloud engineering,
gaming, and linkedin. sales and marketing
```

In [4]:
```
from sentence_transformers import CrossEncoder
cross_encoder = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-6-v2
```

config.json: 100% 794/794 [00:00<00:00, 91.5kB/s]

pytorch_model.bin: 100% 90.9M/90.9M [00:01<00:00, 73.6MB/s]

tokenizer_config.json: 100% 316/316 [00:00<00:00, 39.0kB/s]

vocab.txt: 100% 232k/232k [00:00<00:00, 3.90MB/s]

special_tokens_map.json: 100% 112/112 [00:00<00:00, 13.9kB/s]

In [5]:
```
pairs = [[query, doc] for doc in retrieved_documents]
scores = cross_encoder.predict(pairs)
print("Scores:")
for score in scores:
    print(score)
```

```
Scores:
0.9869341
2.6445777
-0.26802987
-10.731592
-7.706605
-5.6469994
-4.297035
-10.933233
-7.038428
-7.324694
```

In [6]:
```
print("New Ordering:")
for o in np.argsort(scores)[::-1]:
    print(o+1)
```

```
New Ordering:
2
1
3
7
6
9
10
5
4
8
```

Re-ranking with Query Expansion

Lastly why not to combine the Query Expansion technique in which we were generating extra queries using LLM that were related to our main query, and Re-ranking to optimize a bit more. But in this scenario we will get 10 documents for each query and then we remove duplicated documents (that were retrieved by every query) and then we can apply re-ranking to get the most relevant documents(may be top 5 or so).

In [7]:
```
original_query = "What were the most important factors that contril
generated_queries = [
    "What were the major drivers of revenue growth?",
    "Were there any new product launches that contributed to the i
    "Did any changes in pricing or promotions impact the revenue g
    "What were the key market trends that facilitated the increase
    "Did any acquisitions or partnerships contribute to the revenu
]
```

In [8]:
```
queries = [original_query] + generated_queries

results = chroma_collection.query(query_texts=queries, n_results=1(
retrieved_documents = results['documents']
```

In [9]:
```
# Deduplicate the retrieved documents
unique_documents = set()
for documents in retrieved_documents:
    for document in documents:
        unique_documents.add(document)

unique_documents = list(unique_documents)
```

In [10]:
```
pairs = []
for doc in unique_documents:
    pairs.append([original_query, doc])
```

In [11]:
```
scores = cross_encoder.predict(pairs)
```

In [12]:
```
print("Scores:")
for score in scores:
    print(score)
```

```
Scores:
-9.768024
-10.0839405
-5.1418324
-4.3417664
-10.042843
-9.80788
-8.505109
-5.27475
-7.754099
-9.357721
-9.918428
-7.917177
-7.490655
-10.000139
-4.818485
-1.1369953
-3.7948635
-11.0792675
-4.6518917
-10.148884
-10.711212
-6.9020915
-3.7681558
```

In [13]:
```
print("New Ordering:")
for o in np.argsort(scores)[::-1]:
    print(o)
```

```
New Ordering:
15
22
16
3
18
14
2
7
21
12
8
11
6
9
0
5
10
13
4
1
19
20
17
```

In [ ]:

In [ ]: