Lab 5 - Embedding Adaptors

In [1]:
```
from helper_utils import load_chroma, word_wrap, project_embeddings
from chromadb.utils.embedding_functions import SentenceTransformerEmbeddingFun
import numpy as np
import umap
from tqdm import tqdm

import torch
```

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

chroma_collection = load_chroma(filename='microsoft_annual_report_2022.pdf', c
chroma_collection.count()
```

```
349
```

In [3]:
```
embeddings = chroma_collection.get(include=['embeddings'])['embeddings']
umap_transform = umap.UMAP(random_state=0, transform_seed=0).fit(embeddings)
projected_dataset_embeddings = project_embeddings(embeddings, umap_transform)
```

```
/usr/local/lib/python3.9/site-packages/umap/umap_.py:1943: UserWarning: n_job
s value -1 overridden to 1 by setting random_state. Use no seed for paralleli
sm.
  warn(f"n_jobs value {self.n_jobs} overridden to 1 by setting random_state.
Use no seed for parallelism.")
100%|██████████| 349/349 [07:29<00:00,  1.29s/it]
```

In [4]:
```
import os
import openai
from openai import OpenAI

from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv()) # read local .env file
openai.api_key = os.environ['OPENAI_API_KEY']

openai_client = OpenAI()
```

Creating a dataset

In [5]:
```
def generate_queries(model="gpt-3.5-turbo"):
    messages = [
        {
            "role": "system",
            "content": "You are a helpful expert financial research assistant.
            "Suggest 10 to 15 short questions that are important to ask when a
            "Do not output any compound questions (questions with multiple sen
            "Output each question on a separate line divided by a newline."
        },
    ]

    response = openai_client.chat.completions.create(
        model=model,
        messages=messages,
    )
    content = response.choices[0].message.content
    content = content.split("\n")
    return content
```

In [6]:
```
generated_queries = generate_queries()
for query in generated_queries:
    print(query)
```

```
huggingface/tokenizers: The current process just got forked, after parallelis
m has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(tru
e | false)

1. What is the company's revenue growth over the past year?
2. How has the company's profitability changed compared to the previous year?
3. What are the company's total assets and liabilities?
4. How does the company's debt level compare to its equity?
5. What is the company's cash flow from operating activities?
6. What are the company's major sources of revenue?
7. How much does the company spend on research and development?
8. What are the company's major operating costs and expenses?
9. How does the company's financial performance compare to industry peers?
10. What is the company's dividend policy?
11. Has the company experienced any significant changes in ownership or manag
ement?
12. What are the company's future growth prospects and strategies?
13. How does the company manage risk, particularly related to economic or ind
ustry-specific factors?
14. What are the company's financial ratios, such as return on assets and liq
uidity ratios?
15. Has the company faced any legal or regulatory issues that could impact th
e business?
```

In [7]:
```
results = chroma_collection.query(query_texts=generated_queries, n_results=10,
retrieved_documents = results['documents']
```

In [8]:
```
def evaluate_results(query, statement, model="gpt-3.5-turbo"):
    messages = [
    {
        "role": "system",
        "content": "You are a helpful expert financial research assistant. You
        "For the given query, evaluate whether the following satement is relev
        "Output only 'yes' or 'no'."
    },
    {
        "role": "user",
        "content": f"Query: {query}, Statement: {statement}"
    }
    ]

    response = openai_client.chat.completions.create(
        model=model,
        messages=messages,
        max_tokens=1
    )
    content = response.choices[0].message.content
    if content == "yes":
        return 1
    return -1
```

In [9]:
```
retrieved_embeddings = results['embeddings']
query_embeddings = embedding_function(generated_queries)
```

In [10]:
```
adapter_query_embeddings = []
adapter_doc_embeddings = []
adapter_labels = []
```

In [11]:
```
for q, query in enumerate(tqdm(generated_queries)):
    for d, document in enumerate(retrieved_documents[q]):
        adapter_query_embeddings.append(query_embeddings[q])
        adapter_doc_embeddings.append(retrieved_embeddings[q][d])
        adapter_labels.append(evaluate_results(query, document))
```

```
100%|██████████| 15/15 [01:07<00:00,  4.52s/it]
```

In [12]:
```
len(adapter_labels)
```

```
150
```

In [13]:
```
adapter_query_embeddings = torch.Tensor(np.array(adapter_query_embeddings))
adapter_doc_embeddings = torch.Tensor(np.array(adapter_doc_embeddings))
adapter_labels = torch.Tensor(np.expand_dims(np.array(adapter_labels),1))
```

In [14]:
```
dataset = torch.utils.data.TensorDataset(adapter_query_embeddings, adapter_doc
```

Setting up the model

In [15]:
```
def model(query_embedding, document_embedding, adaptor_matrix):
    updated_query_embedding = torch.matmul(adaptor_matrix, query_embedding)
    return torch.cosine_similarity(updated_query_embedding, document_embedding
```

In [16]:
```
def mse_loss(query_embedding, document_embedding, adaptor_matrix, label):
    return torch.nn.MSELoss()(model(query_embedding, document_embedding, adapt
```

In [17]:
```
# Initialize the adaptor matrix
mat_size = len(adapter_query_embeddings[0])
adapter_matrix = torch.randn(mat_size, mat_size, requires_grad=True)
```

In [18]:
```
min_loss = float('inf')
best_matrix = None

for epoch in tqdm(range(100)):
    for query_embedding, document_embedding, label in dataset:
        loss = mse_loss(query_embedding, document_embedding, adapter_matrix, l

        if loss < min_loss:
            min_loss = loss
            best_matrix = adapter_matrix.clone().detach().numpy()

        loss.backward()
        with torch.no_grad():
            adapter_matrix -= 0.01 * adapter_matrix.grad
            adapter_matrix.grad.zero_()
```

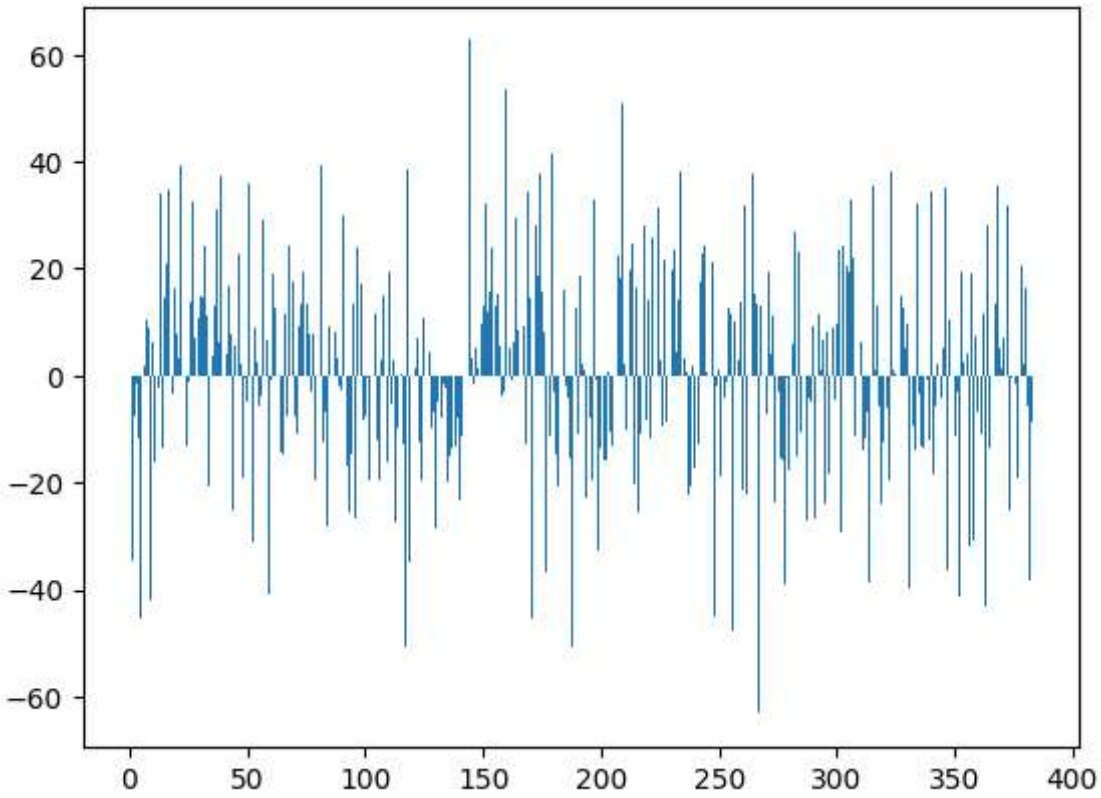
```
  0%|          | 0/100 [00:00<?, ?it/s]/usr/local/lib/python3.9/site-package
s/torch/nn/modules/loss.py:535: UserWarning: Using a target size (torch.Size
([1])) that is different to the input size (torch.Size([])). This will likely
lead to incorrect results due to broadcasting. Please ensure they have the sa
me size.
  return F.mse_loss(input, target, reduction=self.reduction)
100%|██████████| 100/100 [02:07<00:00,  1.28s/it]
```

In [19]:
```
print(f"Best loss: {min_loss.detach().numpy()}")
```

```
Best loss: 0.49309027194976807
```

In [20]:
```
test_vector = torch.ones((mat_size,1))
scaled_vector = np.matmul(best_matrix, test_vector).numpy()
```

In [21]:
```
import matplotlib.pyplot as plt
plt.bar(range(len(scaled_vector)), scaled_vector.flatten())
plt.show()
```

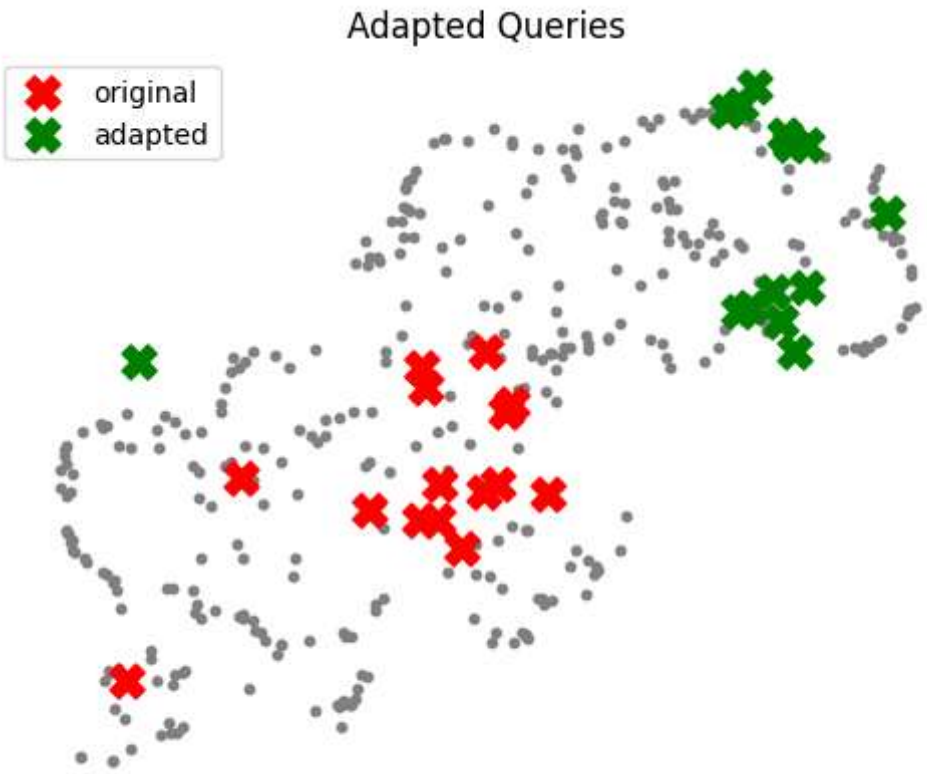


In [22]:
```
query_embeddings = embedding_function(generated_queries)
adapted_query_embeddings = np.matmul(best_matrix, np.array(query_embeddings).T

projected_query_embeddings = project_embeddings(query_embeddings, umap_transfo
projected_adapted_query_embeddings = project_embeddings(adapted_query_embeddin
```

```
100%|██████████| 15/15 [00:19<00:00,  1.32s/it]
100%|██████████| 15/15 [00:18<00:00,  1.26s/it]
```

In [23]:
```
# Plot the projected query and retrieved documents in the embedding space
plt.figure()
plt.scatter(projected_dataset_embeddings[:, 0], projected_dataset_embeddings[:
plt.scatter(projected_query_embeddings[:, 0], projected_query_embeddings[:, 1]
plt.scatter(projected_adapted_query_embeddings[:, 0], projected_adapted_query_

plt.gca().set_aspect('equal', 'datalim')
plt.title("Adapted Queries")
plt.axis('off')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fef784a7df0>
```



In [ ]:

In [ ]: