Problem 1 - Approximate Nearest Neighbors (25 points)

Given a dataset of vectors in a high-dimensional space, implement and analyze an Approximate Nearest Neighbors (ANN) solution using the Hierarchical Navigable Small World (HNSW) approach.

Note #1: Use the following test parameters:

- Number of vectors: 100
- Dimension: 2
- M-nearest neighbors: 2
- Test with query vector [0.5, 0.5]

Required Libraries: numpy, networkx, matplotlib

Note #2: Submit your code with clear documentation and visualizations of the graph structure and search process.

(10 points) Task (a):

Implement a function `construct_HNSW(vectors, m_neighbors)` that builds a hierarchical graph structure where:

- `vectors` is a numpy array of shape (n_vectors, dimension)
- `m_neighbors` is the number of nearest neighbors to connect in each layer
- Return a list of networkx graphs representing each layer

In [207…

```
#Import required libraries
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import random
import time

#Define number of vectors, dimension, m_neighbors and generate a random dataset (specs mentioned above and in the PDF)
num_vectors = 100
dim = 2
m_neighbors = 2
vectors = np.random.rand(num_vectors, dim)
query_vec = [0.5, 0.5]
```

HNSW Construction

In [209…

```python
#Define the function construct_HNSW

def find_nearest_neighbors(layer, node_id, k):
    # Get distances of all nodes in the layer from node node_id
    node_pos = layer.nodes[node_id]['pos']
    distances = [(np.linalg.norm(node_pos - layer.nodes[n]['pos']), n) for n in layer.nodes if n != node_id]
    # Sort the distances in increasing order
    distances.sort(key=lambda x: x[0])

    # Return ids for the top k nodes closest to node_id
    return [neighbor for _, neighbor in distances[:k]]

#Implement the function with the dataset
def construct_HNSW(vectors, m_neighbors):
    # Initialize the number of layers in the HNSW network
    num_layers = 7

    # Create a list of graphs, one for each layer
    graphs = [nx.Graph() for _ in range(num_layers)]

    # Add all the vectors as nodes to layer 0
    for i, vector in enumerate(vectors):
        graphs[0].add_node(i, pos=vector)

    # Assign a max_layer attribute to each node in the bottom layer such that around 1/e of nodes will be at layer <= 1
    mL = 1.0 / np.log(m_neighbors)
    for i in range(len(vectors)):
        u = random.random()
        max_layer = int(np.floor(-np.log(u) * mL))
        graphs[0].nodes[i]['max_layer'] = min(max_layer, num_layers - 1)

    # Add nodes and edges in all layers
    for i in range(len(vectors)):
        max_layer = graphs[0].nodes[i]['max_layer']

        # Add the node to all layers up to its max_layer
        for layer_ind in range(1, max_layer + 1):
            graphs[layer_ind].add_node(i, pos=vectors[i], max_layer=max_layer)

        # Create edges in each layer for this node
        for layer_ind in range(max_layer + 1):
            if i in graphs[layer_ind]:
                # Find m nearest neighbors within the current layer
                neighbors = find_nearest_neighbors(graphs[layer_ind], i, m_neighbors)
```

```
                # Add an edge between node i and m nearest neighbors within the current layer
                for neighbor in neighbors:
                    graphs[layer_ind].add_edge(i, neighbor)
    return graphs

graph_layers = construct_HNSW(vectors, m_neighbors)
```

(8 points) Task (b):

Implement a function `search_HNSW(graph_layers, query)` that performs approximate nearest neighbor search. Your function should:

- Accept the graph layers from `construct HNSW` and a query vector
- Return the nearest neighbor found and the search path taken
- Use the layer-wise search strategy discussed in class

In [211…

```
#Your code here of the implementation

# helper function to calculate euclidean distance between two vectors
def euclidean_dist(a, b):
    return np.linalg.norm(a - b)

def search_HNSW(graph_layers, query):
    num_layers = len(graph_layers)
    lastLayerInd = num_layers - 1
    lastLayer = graph_layers[lastLayerInd]
    # select a random node from last layer as starting point
    while len(list(lastLayer.nodes())) <= 0:
        lastLayerInd -= 1
        lastLayer = graph_layers[lastLayerInd]
    start_id = random.choice(list(lastLayer.nodes()))
    # initialize the path with just the starting point node
    path = [(num_layers - 1, vectors[start_id])]

    # iterate through each layer from top to bottom
    for layer in range(lastLayerInd, -1, -1):
        current_id = start_id
        changed = True
        # keep moving to closer neighbors until current_id is the closest node to query in the layer
        while changed:
            changed = False
            currDist = euclidean_dist(vectors[current_id], query)
            # iterate through neighbors of current
            for neighbor in graph_layers[layer].neighbors(current_id):
                # if any neighbor is closer to query move from current to neighbor and append current node to the path
                if euclidean_dist(vectors[neighbor], query) < currDist:
```

```
                    current_id = neighbor
                    path.append((layer, vectors[current_id]))
                    changed = True
        # set the start node to closest node to query in this layer before starting nearest neighbors search in next la
        start_id = current_id

    # return resultant path to closest node and the closest node's vector
    return path, vectors[start_id]

(search_path, nearest_neighbor) = search_HNSW(graph_layers, query)
#print("Path: " + str(search_path))
#print("Nearest Neighbor: " + str(nearest_neighbor))
```

```
Path: [(6, array([0.64582232, 0.43633695])), (5, array([0.55568291, 0.48706499])), (5, array([0.53743626, 0.6041269
])), (5, array([0.55568291, 0.48706499]))]
Nearest Neighbor: [0.55568291 0.48706499]
```

(7 points) Task (c):

Evaluate your implementation by:

- Comparing results against brute force search for a dataset of 100 vectors in 2D space
- Measuring and reporting search time for both methods
- Visualizing one example search path through the layers
- Calculating and reporting the accuracy of your approximate solution

Brute Force

In [215…

```
# Implement brute force search
def nearest_neighbor_brute_force(vectors, query):
    closest_dist = float("inf")
    closest_id = -1

    # Loop through every single vector in vectors and compare each vector to query
    for i, vec in enumerate(vectors):
        dist = euclidean_dist(vec, query)
        # Update closest_dist and closest_id if current vector distance to query < closest_dist
        if dist < closest_dist:
            closest_dist = dist
            closest_id = i
    # Return vector that is closest to query
    return vectors[closest_id]
ans = nearest_neighbor_brute_force(vectors, query)
#print(ans)
```

```
[0.55568291 0.48706499]
```

Measure and compare search times in these two cases

In [217…

```python
#YOUR CODE/OUTPUTS HERE

# Measure search time for brute force nearest neighbor search and display true nearest neighbor answer
start = time.time()
true_ans = nearest_neighbor_brute_force(vectors, query)
end = time.time()
total_time = end - start
print("Brute Force Nearest Neighbor Search Total Time: " + str(total_time) + " Seconds")
print("Answer: " + str(true_ans))
print()

# Construct HNSW network
graph_layers = construct_HNSW(vectors, m_neighbors)

# Measure search time for hnsw nearest neighbor search and display approximate nearest neighbor answer
start = time.time()
(search_path, approximate_nearest_neighbor) = search_HNSW(graph_layers, query)
end = time.time()
total_time = end - start
print("Hierarchical Navigable Small World Approximate Nearest Neighbor Search Total Time: " + str(total_time) + " Secor
print("Answer: " + str(approximate_nearest_neighbor))
```

```
Brute Force Nearest Neighbor Search Total Time: 0.0038361549377441406 Seconds
Answer: [0.55568291 0.48706499]

Hierarchical Navigable Small World Approximate Nearest Neighbor Search Total Time: 0.0005290508270263672 Seconds
Answer: [0.55568291 0.48706499]
```

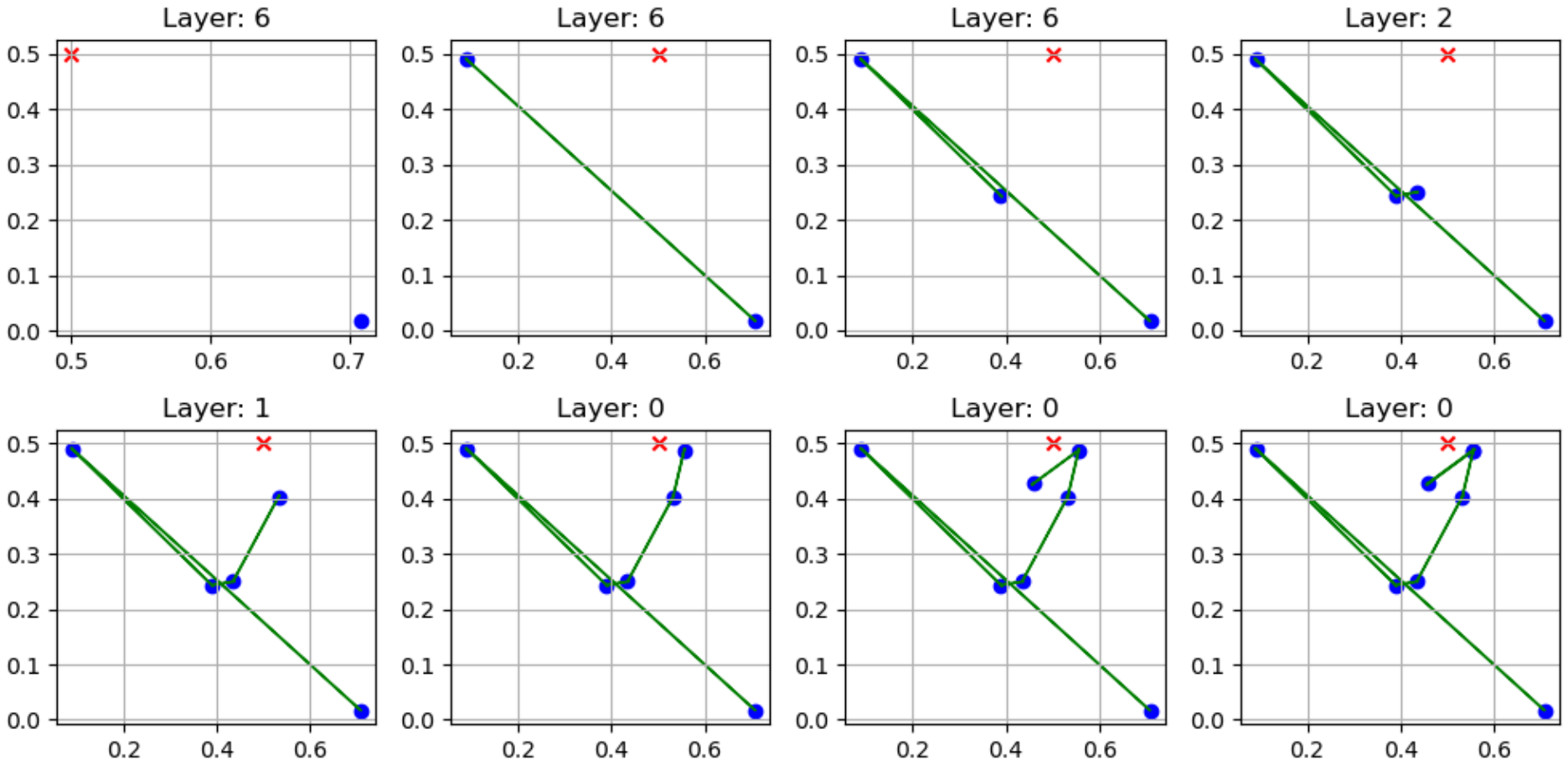
Visualize one example search path

In [219…

```python
# YOUR CODE HERE
graph_layers = construct_HNSW(vectors, m_neighbors)
(path, nearest_neighbor) = search_HNSW(graph_layers, query)
num_layers = len(graph_layers)
fig, axs = plt.subplots(nrows=2, ncols=4, figsize=(10, 5))
axs = axs.flatten()

for i, ax in enumerate(axs):
    if i >= len(path):
        break
    layer = path[i][0]
    ax.set_title("Layer: " + str(layer))
    ax.scatter(*query, color='red', marker='x', label='Query')
```

```
    coords = []
    for j in range(i + 1):
        vec = path[j][1]
        coords.append(vec)
        ax.scatter(*vec, color='blue')
    for j in range(len(coords) - 1):
        ax.arrow(coords[j][0], coords[j][1], coords[j+1][0] - coords[j][0], coords[j+1][1] - coords[j][1], color='green

    ax.grid(True)

print("Query is represented as a red x")
print("Vector data points are represented by blue dots")
print("Visualization For Example Search Path: " + str(path))
plt.tight_layout()
plt.show()
```

```
Query is represented as a red x
Vector data points are represented by blue dots
Visualization For Example Search Path: [(6, array([0.70885531, 0.01645161])), (6, array([0.09030545, 0.48911854])), (6,
array([0.38913044, 0.24339891])), (2, array([0.43440611, 0.24974109])), (1, array([0.53222012, 0.40315244])), (0, array
([0.55568291, 0.48706499])), (0, array([0.45815364, 0.42661584])), (0, array([0.55568291, 0.48706499]))]
```



Calculate and report accuracy of approximate search case

In [221…

```
# Your code here
def evaluate_accuracy(graph_layers, vectors, queries):
    correct = 0
    for q in queries:
        _, approx = search_HNSW(graph_layers, q)
        true = nearest_neighbor_brute_force(vectors, q)
        if np.array_equal(approx, true):
            correct += 1
    return correct / len(queries)

graph_layers = construct_HNSW(vectors, m_neighbors)
queries = []
for _ in range(500000):
    random_array = np.random.rand(1, 2)
    queries.append(random_array)
```

```
# Evaluated the accuracy of Hierarchical Navigable Small World Approximate Nearest Neighbor Search on 500000 random que
print("Resultant Accuracy of Hierarchical Navigable Small World Approximate Nearest Neighbor Search on 500000 random qu
```

```
Resultant Accuracy of Hierarchical Navigable Small World Approximate Nearest Neighbor Search on 500000 random queries:
0.656258
```

Problem 1 Bonus:

- (+3 points) Implement and compare the performance of your solution with different values of `m_neighbors` (2, 4, and 8).
- (+2 points) Test your algorithm on a real dataset embedding (like Wikipedia) and report your results.

In [ ]:

Problem 2 - Semantic Search for RAG Systems (30 points)

Tasks:

1. **(12 points) Document Processing and Vector Store Setup:**

   - Load text documents using appropriate document loaders (e.g., PyPDFLoader for PDFs)
   - Split documents into chunks using `chunk_size` and `chunk_overlap` parameters with `RecursiveCharacterTextSplitter`
   - Generate embeddings for each chunk using a pre-trained embedding model
   - Store the embeddings in a vector database (e.g., Chroma) with proper metadata
   - Demonstrate the setup with sample documents and verify chunk creation

2. **(10 points) Implement Semantic Search with Multiple Methods:**

   - Implement similarity search using `vector_store.similarity_search()` method
   - Implement Maximum Marginal Relevance (MMR) search using `vector_store.max_marginal_relevance_search()` method
   - **MMR Definition:** A retrieval strategy that balances relevance with diversity to reduce redundant results.
     MMR iteratively selects documents that are both relevant to the query **and** different from already-selected documents.
     Use the formula:

$$MMR = \lambda \times Sim(doc, query) - (1 - \lambda) \times \max(Sim(doc, selected))$$

   where (\lambda = 0.5)

   - Demonstrate both search methods with the same queries and compare results
   - Include search with similarity scores using `similarity_search_with_score()`

3. **(8 points) Evaluate your implementation by:**

   - Testing on a provided dataset of 20 short articles about different topics
   - Comparing retrieval quality between different chunk sizes (500, 1000, 1500 characters)
   - Measuring and reporting search latency for different values of k (1, 5, 10)
   - Analyzing the effect of different search types on result diversity (compare similarity vs MMR results for the same query)
   - Demonstrating scenarios where MMR provides better coverage than similarity search
   - Creating visualizations of embedding similarities using dimensionality reduction

Bonus:

- **(+4 points):** Implement a hybrid search combining semantic search with keyword-based BM25 scoring
- **(+3 points):** Create a simple web interface demonstrating your search engine
- **(+3 points):** Implement and compare different embedding models (e.g., OpenAI, Cohere, local models)

In [3]:
```
# (run once if needed)
!pip install -q "langchain>=0.2.10" "langchain-openai>=0.2.10" "langchain-community>=0.2.10" "langchain-huggingface>=0.
                "langchain-text-splitters>=0.2.0" "chromadb>=0.5.5" \
                "sentence-transformers>=2.2.2" "pypdf>=4.2.0"
```

```
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 67.3/67.3 kB 2.0 MB/s eta 0:00:00
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 76.0/76.0 kB 3.2 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.5/2.5 MB 28.2 MB/s eta 0:00:0000:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 20.8/20.8 MB 69.4 MB/s eta 0:00:00:00:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 278.2/278.2 kB 11.1 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.0/2.0 MB 46.6 MB/s eta 0:00:00:00:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 449.8/449.8 kB 18.0 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 103.1/103.1 kB 4.8 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 17.4/17.4 MB 63.6 MB/s eta 0:00:00:00:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.0/1.0 MB 32.4 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 72.5/72.5 kB 2.7 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 132.3/132.3 kB 4.6 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 65.9/65.9 kB 2.5 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 208.0/208.0 kB 7.7 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 105.4/105.4 kB 5.0 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 71.4/71.4 kB 3.1 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 566.1/566.1 kB 19.5 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 363.4/363.4 MB 4.1 MB/s eta 0:00:00:00:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.8/13.8 MB 75.5 MB/s eta 0:00:00:00:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.6/24.6 MB 60.9 MB/s eta 0:00:00:00:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 883.7/883.7 kB 28.2 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 664.8/664.8 MB 2.2 MB/s eta 0:00:00:00:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 211.5/211.5 MB 7.0 MB/s eta 0:00:00:00:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.3/56.3 MB 26.9 MB/s eta 0:00:00:00:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 127.9/127.9 MB 11.1 MB/s eta 0:00:0000:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 207.5/207.5 MB 4.8 MB/s eta 0:00:00:00:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.1/21.1 MB 67.9 MB/s eta 0:00:00:00:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 456.6/456.6 kB 18.4 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 323.2/323.2 kB 11.1 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 128.4/128.4 kB 5.0 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.8/3.8 MB 60.1 MB/s eta 0:00:00:00:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 456.1/456.1 kB 14.7 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 46.0/46.0 kB 1.5 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 86.8/86.8 kB 3.7 MB/s eta 0:00:00
  Building wheel for pypika (pyproject.toml) ... done
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behavio
ur is the source of the following dependency conflicts.
bigframes 2.12.0 requires google-cloud-bigquery-storage<3.0.0,>=2.30.0, which is not installed.
datasets 4.1.1 requires pyarrow>=21.0.0, but you have pyarrow 19.0.1 which is incompatible.
google-api-core 1.34.1 requires protobuf!=3.20.0,!=3.20.1,!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<4.0.0d
ev,>=3.19.5, but you have protobuf 6.33.0 which is incompatible.
google-cloud-translate 3.12.1 requires protobuf!=3.20.0,!=3.20.1,!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,
<5.0.0dev,>=3.19.5, but you have protobuf 6.33.0 which is incompatible.
google-cloud-bigtable 2.32.0 requires google-api-core[grpc]<3.0.0,>=2.17.0, but you have google-api-core 1.34.1 which i
s incompatible.
```

```
google-colab 1.0.0 requires google-auth==2.38.0, but you have google-auth 2.40.3 which is incompatible.
google-colab 1.0.0 requires notebook==6.5.7, but you have notebook 6.5.4 which is incompatible.
google-colab 1.0.0 requires pandas==2.2.2, but you have pandas 2.2.3 which is incompatible.
google-colab 1.0.0 requires requests==2.32.3, but you have requests 2.32.5 which is incompatible.
google-colab 1.0.0 requires tornado==6.4.2, but you have tornado 6.5.2 which is incompatible.
bigframes 2.12.0 requires google-cloud-bigquery[bqstorage,pandas]>=3.31.0, but you have google-cloud-bigquery 3.25.0 wh
ich is incompatible.
bigframes 2.12.0 requires rich<14,>=12.4.4, but you have rich 14.1.0 which is incompatible.
libcugraph-cu12 25.6.0 requires libraft-cu12==25.6.*, but you have libraft-cu12 25.2.0 which is incompatible.
gradio 5.38.1 requires pydantic<2.12,>=2.0, but you have pydantic 2.12.0a1 which is incompatible.
google-ai-generativelanguage 0.6.15 requires protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=
3.20.2, but you have protobuf 6.33.0 which is incompatible.
pydrive2 1.21.3 requires cryptography<44, but you have cryptography 46.0.1 which is incompatible.
pydrive2 1.21.3 requires pyOpenSSL<=24.2.1,>=19.1.0, but you have pyopenssl 25.3.0 which is incompatible.
pandas-gbq 0.29.2 requires google-api-core<3.0.0,>=2.10.2, but you have google-api-core 1.34.1 which is incompatible.
google-cloud-storage 2.19.0 requires google-api-core<3.0.0dev,>=2.15.0, but you have google-api-core 1.34.1 which is in
compatible.
tensorflow 2.18.0 requires protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.3, but you ha
ve protobuf 6.33.0 which is incompatible.
pylibcugraph-cu12 25.6.0 requires pylibraft-cu12==25.6.*, but you have pylibraft-cu12 25.2.0 which is incompatible.
pylibcugraph-cu12 25.6.0 requires rmm-cu12==25.6.*, but you have rmm-cu12 25.2.0 which is incompatible.
jupyter-kernel-gateway 2.5.2 requires jupyter-client<8.0,>=5.2.0, but you have jupyter-client 8.6.3 which is incompatib
le.
dataproc-spark-connect 0.8.3 requires google-api-core>=2.19, but you have google-api-core 1.34.1 which is incompatible.
gcsfs 2025.3.0 requires fsspec==2025.3.0, but you have fsspec 2025.9.0 which is incompatible.
```

In [4]:
```python
import langchain
print(f"LangChain version: {langchain.__version__}")
```

```
LangChain version: 0.3.27
```

In [5]:
```python
# Import necessary libraries
import os
from dotenv import load_dotenv
from langchain_openai import OpenAI
from langchain.agents import load_tools, initialize_agent, AgentType
from langchain.tools import Tool
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate

# Load environment variables, including both OPENAI_API_KEY and SERPAPI_API_KEY
os.environ['OPENAI_API_KEY'] = 'INSERT OPENAI_API_KEY'
os.environ['SERPAPI_API_KEY'] = 'INSERT SERPAPI_API_KEY'
load_dotenv()

# Initialize the OpenAI language model
llm = OpenAI(temperature=0)
```

In [46]:
```python
# --- imports & paths ---
import os, glob
from langchain_community.document_loaders import PyPDFLoader
from  langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.vectorstores import Chroma

DATA_DIR = "../input/documents"      # put your 8–10 docs here (PDF/TXT/MD)
PERSIST_DIR = "./chroma"    # where Chroma DB will be saved
COLLECTION = "rag_demo"
CHUNK_SIZE = 1000
CHUNK_OVERLAP = 200
```

In [13]:
```python
# Collection of 8–10 Wikipedia articles covering different topics (technology, science, sports, politics, history)
ls ../input/documents
```

```
History_of_sport.pdf       History.pdf           Politics.pdf  Sport.pdf
History_of_technology.pdf  Machine_learning.pdf  Science.pdf   Technology.pdf
```

```
huggingface/tokenizers: The current process just got forked, after parallelism has already been used. Disabling paralle
lism 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=(true | false)
```

In [7]:
```python
# Load text documents using appropriate document loaders (e.g., PyPDFLoader for PDFs)
paths = sorted(
    glob.glob(os.path.join(DATA_DIR, "**/*.pdf"), recursive=True)
  + glob.glob(os.path.join(DATA_DIR, "**/*.txt"), recursive=True)
  + glob.glob(os.path.join(DATA_DIR, "**/*.md"),  recursive=True)
)
docs = []
for p in paths:
    loader = PyPDFLoader(p) if p.lower().endswith(".pdf") else TextLoader(p, encoding="utf-8")
    docs.extend(loader.load())
print(f"Loaded {len(docs)} docs from {len(paths)} files")
```

```
Loaded 248 docs from 8 files
```

In [8]:
```python
# Split documents into chunks using RecursiveCharacterTextSplitter with chunk size and chunk overlap parameters
splitter = RecursiveCharacterTextSplitter(
    chunk_size=CHUNK_SIZE,
    chunk_overlap=CHUNK_OVERLAP,
    add_start_index=True,
)
chunks = splitter.split_documents(docs)
print(f"Created {len(chunks)} chunks")
```

```
Created 1016 chunks
```

In [9]:
```python
# Initialize pre-trained embedding model
emb = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
```

```
2025-10-30 15:57:00.016318: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:477] Unable to register cuFFT fac
tory: Attempting to register factory for plugin cuFFT when one has already been registered
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
E0000 00:00:1761839820.326532      37 cuda_dnn.cc:8310] Unable to register cuDNN factory: Attempting to register factor
y for plugin cuDNN when one has already been registered
E0000 00:00:1761839820.429965      37 cuda_blas.cc:1418] Unable to register cuBLAS factory: Attempting to register fact
ory for plugin cuBLAS when one has already been registered
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
AttributeError: 'MessageFactory' object has no attribute 'GetPrototype'
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
AttributeError: 'MessageFactory' object has no attribute 'GetPrototype'
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
AttributeError: 'MessageFactory' object has no attribute 'GetPrototype'
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
AttributeError: 'MessageFactory' object has no attribute 'GetPrototype'
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
AttributeError: 'MessageFactory' object has no attribute 'GetPrototype'
modules.json:   0%|          | 0.00/349 [00:00<?, ?B/s]
config_sentence_transformers.json:   0%|          | 0.00/116 [00:00<?, ?B/s]
README.md: 0.00B [00:00, ?B/s]
sentence_bert_config.json:   0%|          | 0.00/53.0 [00:00<?, ?B/s]
config.json:   0%|          | 0.00/612 [00:00<?, ?B/s]
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schema.py:2225: UnsupportedFieldAttributeWarning:
The 'repr' attribute with value False was provided to the `Field()` function, which has no effect in the context it was
used. 'repr' is field-specific metadata, and can only be attached to a model field using `Annotated` metadata or by ass
ignment. This may have happened because an `Annotated` type alias using the `type` statement was used, or if the `Field
()` function was attached to a single member of a union type.
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schema.py:2225: UnsupportedFieldAttributeWarning:
The 'frozen' attribute with value True was provided to the `Field()` function, which has no effect in the context it wa
s used. 'frozen' is field-specific metadata, and can only be attached to a model field using `Annotated` metadata or by
assignment. This may have happened because an `Annotated` type alias using the `type` statement was used, or if the `Fi
eld()` function was attached to a single member of a union type.
  warnings.warn(
model.safetensors:   0%|          | 0.00/90.9M [00:00<?, ?B/s]
tokenizer_config.json:   0%|          | 0.00/350 [00:00<?, ?B/s]
```

```
vocab.txt: 0.00B [00:00, ?B/s]
tokenizer.json: 0.00B [00:00, ?B/s]
special_tokens_map.json:   0%|          | 0.00/112 [00:00<?, ?B/s]
config.json:   0%|          | 0.00/190 [00:00<?, ?B/s]
```

In [47]:
```python
# Generate embeddings for each chunk using a pre-trained embedding model
# Store the embeddings in a vector database (e.g., Chroma) with proper metadata to enable cosine distance calculations
vs = Chroma.from_documents(
    documents=chunks,
    embedding=emb,
    persist_directory=PERSIST_DIR,
    collection_name=COLLECTION,
    collection_metadata={"hnsw:space": "cosine"}
)
vs.persist()
print("Persisted at:", os.path.abspath(PERSIST_DIR))
print("Vector count:", vs._collection.count())
```

```
Persisted at: /kaggle/working/chroma
Vector count: 1016
```

In [48]:
```python
# Demonstrate the setup with sample documents and verify chunk creation
sample = vs._collection.get(include=["metadatas"], limit=3)
print("Sample metadatas:", sample.get("metadatas", []))
```

```
Sample metadatas: [{'producer': 'Skia/PDF m141', 'page_label': '1', 'moddate': '2025-10-30T15:47:43+00:00', 'total_page
s': 47, 'creationdate': '2025-10-30T15:47:43+00:00', 'page': 0, 'source': '../input/documents/History.pdf', 'start_inde
x': 0, 'title': 'History - Wikipedia', 'creator': 'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Geck
o) HeadlessChrome/141.0.0.0 Safari/537.36'}, {'page_label': '1', 'creationdate': '2025-10-30T15:47:43+00:00', 'start_in
dex': 894, 'creator': 'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/141.0.0.0
Safari/537.36', 'total_pages': 47, 'source': '../input/documents/History.pdf', 'moddate': '2025-10-30T15:47:43+00:00',
'producer': 'Skia/PDF m141', 'page': 0, 'title': 'History - Wikipedia'}, {'total_pages': 47, 'producer': 'Skia/PDF m14
1', 'page_label': '1', 'moddate': '2025-10-30T15:47:43+00:00', 'page': 0, 'creator': 'Mozilla/5.0 (X11; Linux x86_64) A
ppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/141.0.0.0 Safari/537.36', 'start_index': 1710, 'source': '../inpu
t/documents/History.pdf', 'title': 'History - Wikipedia', 'creationdate': '2025-10-30T15:47:43+00:00'}]
```

In [49]:
```python
total_vectors = vs._collection.count()

# Implement similarity search using vector_store.similarity_search_with_score() method
def vector_store_similarity_search(query, k):
    # Handle edge cases where k exceeds available documents
    k = min(k, total_vectors)
    results = vs.similarity_search_with_score(query=query, k=k)
    return results

# Implement Maximum Marginal Relevance (MMR) search using vector_store.max_marginal_relevance_search() method
def vector_store_max_marginal_relevance_search(query, k):
    # Handle edge cases where k exceeds available documents
```

```
    k = min(k, total_vectors)
    results = vs.max_marginal_relevance_search(query=query, k=k)
    return results
```

In [58]:

```
# Demonstrate both search methods with the same queries and compare results
test_queries = ["artificial intelligence applications", "climate change effects", "Olympic games history", "economic pc

# Loop through queries
for query in test_queries:
    print("Query: " + query)
    # Get Similarity Search results
    similarity_search_results = vector_store_similarity_search(query, k=3)
    print("Similarity Search with Scores Results")
    for (document, score) in similarity_search_results:
        # Raw score = cosine distance
        # Cosine similarity = 1 - cosine distance
        cosine_similarity = 1 - score
        print("Document: " + document.metadata["title"] + ", Page: " + str(document.metadata["page"]) + ", Cosine Simil

    # Get MMR Search results
    mmr_search_results = vector_store_max_marginal_relevance_search(query, k=3)
    print("Max Marginal Relevance Search Results")
    for document in mmr_search_results:
        print("Document: " + document.metadata["title"] + ", Page: " + str(document.metadata["page"]))
    print()
    print()

"""
For the first query("artificial intelligence applications"), both search methods returned the same results(the exact sa
"""
```

```
Query: artificial intelligence applications
Similarity Search with Scores Results
Document: Machine learning - Wikipedia, Page: 11, Cosine Similarity Score: 0.5009536743164062
Document: Machine learning - Wikipedia, Page: 14, Cosine Similarity Score: 0.4993807077407837
Document: Machine learning - Wikipedia, Page: 18, Cosine Similarity Score: 0.4594351053237915
Max Marginal Relevance Search Results
Document: Machine learning - Wikipedia, Page: 11
Document: Machine learning - Wikipedia, Page: 14
Document: Machine learning - Wikipedia, Page: 18


Query: climate change effects
Similarity Search with Scores Results
Document: Technology - Wikipedia, Page: 20, Cosine Similarity Score: 0.5481756925582886
Document: Technology - Wikipedia, Page: 20, Cosine Similarity Score: 0.5200040340423584
Document: Science - Wikipedia, Page: 37, Cosine Similarity Score: 0.4506065845489502
Max Marginal Relevance Search Results
Document: Technology - Wikipedia, Page: 20
Document: Machine learning - Wikipedia, Page: 35
Document: Technology - Wikipedia, Page: 5


Query: Olympic games history
Similarity Search with Scores Results
Document: History of sport - Wikipedia, Page: 1, Cosine Similarity Score: 0.611660361289978
Document: History of sport - Wikipedia, Page: 14, Cosine Similarity Score: 0.5991773009300232
Document: History of sport - Wikipedia, Page: 1, Cosine Similarity Score: 0.5907113552093506
Max Marginal Relevance Search Results
Document: History of sport - Wikipedia, Page: 1
Document: Sport - Wikipedia, Page: 15
Document: Sport - Wikipedia, Page: 8


Query: economic policy impacts
Similarity Search with Scores Results
Document: History - Wikipedia, Page: 8, Cosine Similarity Score: 0.41303467750549316
Document: Technology - Wikipedia, Page: 5, Cosine Similarity Score: 0.4066343307495117
Document: Technology - Wikipedia, Page: 25, Cosine Similarity Score: 0.40126627683639526
Max Marginal Relevance Search Results
Document: History - Wikipedia, Page: 8
Document: Technology - Wikipedia, Page: 5
Document: Science - Wikipedia, Page: 14
```

Out[58]:
```
'\nFor the first query("artificial intelligence applications"), both search methods returned the same results(the exac
t same pages from the machine learning wikipedia article in the exact same order as well). For the second query("clima
te change effects"), there is more of a difference between the results for the search methods as the results differ mo
re in terms of the pages and overall documents retrieved. It also looks like the max marginal relevance search method
produces more diverse results than the regular similarity search method. For the third query("Olympic games history"),
the results show the same trend that was seen in the previous query. Also, the max marginal relevance search method st
ill produces similar results in terms of diversity as what was seen in the previous query. For the fouth query("econom
ic policy impacts"), the results for the search methods are more similar than what was seen for queries two and three
in terms of the pages and overall documents retrieved. The search methods also seem to basically agree on the most rel
evant chunk(in terms of the overall document and page retrieved) for each query.\n'
```

Comparison of Results

For the first query("artificial intelligence applications"), both search methods returned the same results(the exact same pages from the machine learning wikipedia article in the exact same order as well). For the second query("climate change effects"), there is more of a difference between the results for the search methods as the results differ more in terms of the pages and overall documents retrieved. It also looks like the max marginal relevance search method produces more diverse results than the regular similarity search method. For the third query("Olympic games history"), the results show the same trend that was seen in the previous query. Also, the max marginal relevance search method still produces similar results in terms of diversity as what was seen in the previous query. For the fouth query("economic policy impacts"), the results for the search methods are more similar than what was seen for queries two and three in terms of the pages and overall documents retrieved. The search methods also seem to basically agree on the most relevant chunk(in terms of the overall document and page retrieved) for each query.

In [59]:
```python
# Comparing retrieval quality between different chunk sizes (500, 1000, 1500 characters) with k(# docs retrieved in eac
chunk_sizes = [500, 1000, 1500]
# Loop through chunk_sizes
for chunk_size in chunk_sizes:
    # Get chunks for this chunk_size
    splitter = RecursiveCharacterTextSplitter(
        chunk_size=chunk_size,
        chunk_overlap=CHUNK_OVERLAP,
        add_start_index=True
    )
    chunks = splitter.split_documents(docs)

    # Generate embeddings for each chunk using a pre-trained embedding model
    # Store the embeddings in a vector database (e.g., Chroma) with proper metadata to enable cosine distance calculati
    vs = Chroma.from_documents(
        documents=chunks,
        embedding=emb,
        persist_directory=PERSIST_DIR + "chunk_size_" + str(chunk_size),
        collection_name=COLLECTION,
        collection_metadata={"hnsw:space": "cosine"}
    )
```

```
    vs.persist()
    print("Chunk Size: " + str(chunk_size))

    # Loop through queries
    for query in test_queries:
        print("Query: " + query)
        # Get Similarity Search results
        similarity_search_results = vector_store_similarity_search(query, k=1)
        print("Similarity Search with Scores Results")
        for (document, score) in similarity_search_results:
            # Raw score = cosine distance
            # Cosine similarity = 1 - cosine distance
            cosine_similarity = 1 - score
            print("Document: " + document.metadata["title"] + ", Page: " + str(document.metadata["page"]) + ", Cosine S

        # Get MMR Search results
        mmr_search_results = vector_store_max_marginal_relevance_search(query, k=1)
        print("Max Marginal Relevance Search Results")
        for document in mmr_search_results:
            print("Document: " + document.metadata["title"] + ", Page: " + str(document.metadata["page"]))
    print("================================================================")
```

```
Chunk Size: 500
Query: artificial intelligence applications
Similarity Search with Scores Results
Document: Machine learning - Wikipedia, Page: 18, Cosine Similarity Score: 0.5682734251022339
Max Marginal Relevance Search Results
Document: Machine learning - Wikipedia, Page: 18
Query: climate change effects
Similarity Search with Scores Results
Document: Science - Wikipedia, Page: 38, Cosine Similarity Score: 0.5841324925422668
Max Marginal Relevance Search Results
Document: Science - Wikipedia, Page: 38
Query: Olympic games history
Similarity Search with Scores Results
Document: History of sport - Wikipedia, Page: 1, Cosine Similarity Score: 0.6728670597076416
Max Marginal Relevance Search Results
Document: History of sport - Wikipedia, Page: 1
Query: economic policy impacts
Similarity Search with Scores Results
Document: Science - Wikipedia, Page: 29, Cosine Similarity Score: 0.4437234401702881
Max Marginal Relevance Search Results
Document: Science - Wikipedia, Page: 29
================================================================
Chunk Size: 1000
Query: artificial intelligence applications
Similarity Search with Scores Results
Document: Machine learning - Wikipedia, Page: 11, Cosine Similarity Score: 0.5009536743164062
Max Marginal Relevance Search Results
Document: Machine learning - Wikipedia, Page: 11
Query: climate change effects
Similarity Search with Scores Results
Document: Technology - Wikipedia, Page: 20, Cosine Similarity Score: 0.5481756925582886
Max Marginal Relevance Search Results
Document: Technology - Wikipedia, Page: 20
Query: Olympic games history
Similarity Search with Scores Results
Document: History of sport - Wikipedia, Page: 1, Cosine Similarity Score: 0.611660361289978
Max Marginal Relevance Search Results
Document: History of sport - Wikipedia, Page: 1
Query: economic policy impacts
Similarity Search with Scores Results
Document: History - Wikipedia, Page: 8, Cosine Similarity Score: 0.41303467750549316
Max Marginal Relevance Search Results
Document: History - Wikipedia, Page: 8
================================================================
Chunk Size: 1500
Query: artificial intelligence applications
Similarity Search with Scores Results
Document: Machine learning - Wikipedia, Page: 14, Cosine Similarity Score: 0.4993807077407837
```

```
Max Marginal Relevance Search Results
Document: Machine learning - Wikipedia, Page: 14
Query: climate change effects
Similarity Search with Scores Results
Document: Technology - Wikipedia, Page: 20, Cosine Similarity Score: 0.5154675841331482
Max Marginal Relevance Search Results
Document: Technology - Wikipedia, Page: 20
Query: Olympic games history
Similarity Search with Scores Results
Document: History of sport - Wikipedia, Page: 11, Cosine Similarity Score: 0.6260300874710083
Max Marginal Relevance Search Results
Document: History of sport - Wikipedia, Page: 11
Query: economic policy impacts
Similarity Search with Scores Results
Document: Technology - Wikipedia, Page: 25, Cosine Similarity Score: 0.3764430284500122
Max Marginal Relevance Search Results
Document: Technology - Wikipedia, Page: 25
============================================================
```

Comparing retrieval quality between different chunk sizes (500, 1000, 1500 characters) with k(# docs retrieved in each similarity search) = 1

For chunk size = 500, both search methods produced the same exact results(with respect to page number and overall document) for each query. The retrieval quality of both search methods was good for the first three queries as both search methods retrieved a chunk from the document that was most related to the search query. However, both search methods retrieved a chunk from the Science Wikipedia article for the last search query("economic policy impacts") which is kind of odd considering that a Wikipedia article on Politics exists in the data. I would assume that chunks of the Politics Wikipedia article would be the most relevant chunks to receive for this query but I guess that wasn't the case for both search methods. For chunk size = 1000, both search methods again produced the same exact results(with respect to page number and overall document) for each query. The results for chunk size = 1000 were also different from the results for chunk size = 500 for all the queries. For the second query("climate change effects"), both search methods retrieved a chunk from the Technology Wikipedia article which is pretty odd considering that most people would assume that the Science Wikipedia article is probably more relevant. Also, both search methods retrieved a chunk from the History Wikipedia article for the last query("economic policy impacts") which makes more sense than retrieving a chunk from the Science Wikipedia article which is what both search methods retrieved for the last query when chunk size = 500. For chunk size = 1500, both search methods again produced the same exact results(with respect to page number and overall document) for each query. For the second query("climate change effects"), both search methods retrieved a chunk from the Technology Wikipedia article which is pretty odd considering that most people would assume that the Science Wikipedia article is probably more relevant. Also, both search methods retrieved a chunk from the Technology Wikipedia article for the last query("economic policy impacts") which makes more sense than retrieving a chunk from the Science Wikipedia article which is what both search methods retrieved for the last query when chunk size = 500 but it's probably not as relevant as retrieving a chunk from the History Wikipedia article which is what both search methods retrieved for the last query when chunk size = 1000. Also, for the most part, the cosine similarity scores produced by the regular vector similarity search method tends to decrease across all queries as the chunk size increases which makes sense since if you

compare the queries to chunks that contain more words the similarity should naturally decrease even if the chunks retrieved are more relevant to the query.

In [69]:
```python
# Measuring and reporting search latency for different values of k = (1, 5, 10) where k = # docs retrieved in each simi
# keeping chunk size constant as CHUNK_SIZE = 1000 across all k
import time
k_vals = [1, 5, 10]
# Loop through k_vals
for k_val in k_vals:
    # Get chunks for this chunk_size
    splitter = RecursiveCharacterTextSplitter(
        chunk_size=CHUNK_SIZE,
        chunk_overlap=CHUNK_OVERLAP,
        add_start_index=True
    )
    chunks = splitter.split_documents(docs)

    # Generate embeddings for each chunk using a pre-trained embedding model
    # Store the embeddings in a vector database (e.g., Chroma) with proper metadata to enable cosine distance calculati
    vs = Chroma.from_documents(
        documents=chunks,
        embedding=emb,
        persist_directory=PERSIST_DIR + "_k_" + str(k_val),
        collection_name=COLLECTION,
        collection_metadata={"hnsw:space": "cosine"}
    )
    vs.persist()
    print("K Value: " + str(k_val))

    # Loop through queries
    for query in test_queries:
        print("Query: " + query)
        # Get Similarity Search results and track total time of similarity search
        start_time = time.time()
        similarity_search_results = vector_store_similarity_search(query, k=k_val)
        end_time = time.time()
        total_time = end_time - start_time
        print("Similarity Search with Scores Results")
        for (document, score) in similarity_search_results:
            # Raw score = cosine distance
            # Cosine similarity = 1 - cosine distance
            cosine_similarity = 1 - score
            print("Document: " + document.metadata["title"] + ", Page: " + str(document.metadata["page"]) + ", Cosine S
        print("Similarity SSearch Latency: " + str(total_time))

        # Get Max Marginal Relevance Search results and track total time of MMR
        start_time = time.time()
```

```
        mmr_search_results = vector_store_max_marginal_relevance_search(query, k=k_val)
        end_time = time.time()
        total_time = end_time - start_time
        print("Max Marginal Relevance Search Results")
        for document in mmr_search_results:
            print("Document: " + document.metadata["title"] + ", Page: " + str(document.metadata["page"]))
        print("Max Marginal Relevance Search Latency: " + str(total_time))
    print("============================================================")
```

```
K Value: 1
Query: artificial intelligence applications
Similarity Search with Scores Results
Document: Machine learning - Wikipedia, Page: 11, Cosine Similarity Score: 0.5009536743164062
Similarity SSearch Latency: 0.032504558563232 42
Max Marginal Relevance Search Results
Document: Machine learning - Wikipedia, Page: 11
Max Marginal Relevance Search Latency: 0.027923583984375
Query: climate change effects
Similarity Search with Scores Results
Document: Technology - Wikipedia, Page: 20, Cosine Similarity Score: 0.5481756925582886
Similarity SSearch Latency: 0.021694183349609375
Max Marginal Relevance Search Results
Document: Technology - Wikipedia, Page: 20
Max Marginal Relevance Search Latency: 0.025656700134277344
Query: Olympic games history
Similarity Search with Scores Results
Document: History of sport - Wikipedia, Page: 1, Cosine Similarity Score: 0.611660361289978
Similarity SSearch Latency: 0.021325349807739258
Max Marginal Relevance Search Results
Document: History of sport - Wikipedia, Page: 1
Max Marginal Relevance Search Latency: 0.024941205978393555
Query: economic policy impacts
Similarity Search with Scores Results
Document: History - Wikipedia, Page: 8, Cosine Similarity Score: 0.41303467750549316
Similarity SSearch Latency: 0.020811080932617188
Max Marginal Relevance Search Results
Document: History - Wikipedia, Page: 8
Max Marginal Relevance Search Latency: 0.027765989303588867
================================================================
K Value: 5
Query: artificial intelligence applications
Similarity Search with Scores Results
Document: Machine learning - Wikipedia, Page: 11, Cosine Similarity Score: 0.5009536743164062
Document: Machine learning - Wikipedia, Page: 14, Cosine Similarity Score: 0.4993807077407837
Document: Machine learning - Wikipedia, Page: 18, Cosine Similarity Score: 0.4594351053237915
Document: Machine learning - Wikipedia, Page: 0, Cosine Similarity Score: 0.459328293800354
Document: Machine learning - Wikipedia, Page: 0, Cosine Similarity Score: 0.44899463653564453
Similarity SSearch Latency: 0.030376911163330078
Max Marginal Relevance Search Results
Document: Machine learning - Wikipedia, Page: 11
Document: Machine learning - Wikipedia, Page: 14
Document: Machine learning - Wikipedia, Page: 18
Document: Science - Wikipedia, Page: 9
Document: Machine learning - Wikipedia, Page: 6
Max Marginal Relevance Search Latency: 0.026303529739379883
Query: climate change effects
Similarity Search with Scores Results
```

```
Document: Technology - Wikipedia, Page: 20, Cosine Similarity Score: 0.5481756925582886
Document: Technology - Wikipedia, Page: 20, Cosine Similarity Score: 0.5200040340423584
Document: Science - Wikipedia, Page: 37, Cosine Similarity Score: 0.4506065845489502
Document: Science - Wikipedia, Page: 37, Cosine Similarity Score: 0.434528648853302
Document: Technology - Wikipedia, Page: 6, Cosine Similarity Score: 0.4281388521194458
Similarity SSearch Latency: 0.02080392837524414
Max Marginal Relevance Search Results
Document: Technology - Wikipedia, Page: 20
Document: Technology - Wikipedia, Page: 20
Document: Machine learning - Wikipedia, Page: 35
Document: Science - Wikipedia, Page: 15
Document: Technology - Wikipedia, Page: 5
Max Marginal Relevance Search Latency: 0.02450728416442871
Query: Olympic games history
Similarity Search with Scores Results
Document: History of sport - Wikipedia, Page: 1, Cosine Similarity Score: 0.611660361289978
Document: History of sport - Wikipedia, Page: 14, Cosine Similarity Score: 0.5991773009300232
Document: History of sport - Wikipedia, Page: 1, Cosine Similarity Score: 0.5907113552093506
Document: Sport - Wikipedia, Page: 15, Cosine Similarity Score: 0.5745881795883179
Document: History of sport - Wikipedia, Page: 2, Cosine Similarity Score: 0.5369930267333984
Similarity SSearch Latency: 0.022122621536254883
Max Marginal Relevance Search Results
Document: History of sport - Wikipedia, Page: 1
Document: Sport - Wikipedia, Page: 15
Document: History of sport - Wikipedia, Page: 14
Document: Sport - Wikipedia, Page: 8
Document: History of sport - Wikipedia, Page: 9
Max Marginal Relevance Search Latency: 0.026004552841186523
Query: economic policy impacts
Similarity Search with Scores Results
Document: History - Wikipedia, Page: 8, Cosine Similarity Score: 0.41303467750549316
Document: Technology - Wikipedia, Page: 5, Cosine Similarity Score: 0.4066343307495117
Document: Technology - Wikipedia, Page: 25, Cosine Similarity Score: 0.40126627683639526
Document: Technology - Wikipedia, Page: 20, Cosine Similarity Score: 0.3885226249694824
Document: Technology - Wikipedia, Page: 17, Cosine Similarity Score: 0.38830137252807617
Similarity SSearch Latency: 0.02340102195739746
Max Marginal Relevance Search Results
Document: History - Wikipedia, Page: 8
Document: Technology - Wikipedia, Page: 20
Document: Technology - Wikipedia, Page: 5
Document: Science - Wikipedia, Page: 14
Document: Technology - Wikipedia, Page: 21
Max Marginal Relevance Search Latency: 0.030003786087036133
================================================================
K Value: 10
Query: artificial intelligence applications
Similarity Search with Scores Results
Document: Machine learning - Wikipedia, Page: 11, Cosine Similarity Score: 0.5009536743164062
```

```
Document: Machine learning - Wikipedia, Page: 14, Cosine Similarity Score: 0.4993807077407837
Document: Machine learning - Wikipedia, Page: 18, Cosine Similarity Score: 0.4594351053237915
Document: Machine learning - Wikipedia, Page: 0, Cosine Similarity Score: 0.459328293800354
Document: Machine learning - Wikipedia, Page: 0, Cosine Similarity Score: 0.44899463653564453
Document: Machine learning - Wikipedia, Page: 1, Cosine Similarity Score: 0.44777363538742065
Document: Machine learning - Wikipedia, Page: 5, Cosine Similarity Score: 0.44655823707580566
Document: Machine learning - Wikipedia, Page: 22, Cosine Similarity Score: 0.4444947838783264
Document: Machine learning - Wikipedia, Page: 8, Cosine Similarity Score: 0.433132529258728
Document: Science - Wikipedia, Page: 9, Cosine Similarity Score: 0.4324820041656494
Similarity SSearch Latency: 0.02861309051513672
Max Marginal Relevance Search Results
Document: Machine learning - Wikipedia, Page: 11
Document: Machine learning - Wikipedia, Page: 14
Document: Machine learning - Wikipedia, Page: 18
Document: Machine learning - Wikipedia, Page: 22
Document: Machine learning - Wikipedia, Page: 8
Document: Science - Wikipedia, Page: 9
Document: Machine learning - Wikipedia, Page: 14
Document: Machine learning - Wikipedia, Page: 15
Document: Machine learning - Wikipedia, Page: 6
Document: Machine learning - Wikipedia, Page: 0
Max Marginal Relevance Search Latency: 0.027344465255737305
Query: climate change effects
Similarity Search with Scores Results
Document: Technology - Wikipedia, Page: 20, Cosine Similarity Score: 0.5481756925582886
Document: Technology - Wikipedia, Page: 20, Cosine Similarity Score: 0.5200040340423584
Document: Science - Wikipedia, Page: 37, Cosine Similarity Score: 0.4506065845489502
Document: Science - Wikipedia, Page: 37, Cosine Similarity Score: 0.434528648853302
Document: Technology - Wikipedia, Page: 6, Cosine Similarity Score: 0.4281388521194458
Document: Science - Wikipedia, Page: 37, Cosine Similarity Score: 0.4209745526313782
Document: Machine learning - Wikipedia, Page: 35, Cosine Similarity Score: 0.40935397148132324
Document: Technology - Wikipedia, Page: 20, Cosine Similarity Score: 0.38700437545776367
Document: Science - Wikipedia, Page: 6, Cosine Similarity Score: 0.34673207998275757
Document: Technology - Wikipedia, Page: 25, Cosine Similarity Score: 0.32454246282577515
Similarity SSearch Latency: 0.02879166603088379
Max Marginal Relevance Search Results
Document: Technology - Wikipedia, Page: 20
Document: Technology - Wikipedia, Page: 20
Document: Science - Wikipedia, Page: 37
Document: Science - Wikipedia, Page: 37
Document: Machine learning - Wikipedia, Page: 35
Document: Science - Wikipedia, Page: 6
Document: Science - Wikipedia, Page: 15
Document: Technology - Wikipedia, Page: 5
Document: Technology - Wikipedia, Page: 8
Document: Science - Wikipedia, Page: 35
Max Marginal Relevance Search Latency: 0.027024507522583008
Query: Olympic games history
```

```
Similarity Search with Scores Results
Document: History of sport - Wikipedia, Page: 1, Cosine Similarity Score: 0.611660361289978
Document: History of sport - Wikipedia, Page: 14, Cosine Similarity Score: 0.5991773009300232
Document: History of sport - Wikipedia, Page: 1, Cosine Similarity Score: 0.5907113552093506
Document: Sport - Wikipedia, Page: 15, Cosine Similarity Score: 0.5745881795883179
Document: History of sport - Wikipedia, Page: 2, Cosine Similarity Score: 0.5369930267333984
Document: History of sport - Wikipedia, Page: 14, Cosine Similarity Score: 0.5348071455955505
Document: Sport - Wikipedia, Page: 3, Cosine Similarity Score: 0.5337450504302979
Document: Sport - Wikipedia, Page: 9, Cosine Similarity Score: 0.5255532264709473
Document: Sport - Wikipedia, Page: 11, Cosine Similarity Score: 0.5240976214408875
Document: Sport - Wikipedia, Page: 8, Cosine Similarity Score: 0.5053505897521973
Similarity SSearch Latency: 0.024884700775146484
Max Marginal Relevance Search Results
Document: History of sport - Wikipedia, Page: 1
Document: Sport - Wikipedia, Page: 15
Document: History of sport - Wikipedia, Page: 2
Document: History of sport - Wikipedia, Page: 14
Document: Sport - Wikipedia, Page: 9
Document: Sport - Wikipedia, Page: 8
Document: History of sport - Wikipedia, Page: 7
Document: Sport - Wikipedia, Page: 15
Document: Sport - Wikipedia, Page: 18
Document: History of sport - Wikipedia, Page: 9
Max Marginal Relevance Search Latency: 0.027594566345214844
Query: economic policy impacts
Similarity Search with Scores Results
Document: History - Wikipedia, Page: 8, Cosine Similarity Score: 0.41303467750549316
Document: Technology - Wikipedia, Page: 5, Cosine Similarity Score: 0.4066343307495117
Document: Technology - Wikipedia, Page: 25, Cosine Similarity Score: 0.40126627683639526
Document: Technology - Wikipedia, Page: 20, Cosine Similarity Score: 0.3885226249694824
Document: Technology - Wikipedia, Page: 17, Cosine Similarity Score: 0.38830137252807617
Document: History of technology - Wikipedia, Page: 1, Cosine Similarity Score: 0.36956238746643066
Document: Technology - Wikipedia, Page: 5, Cosine Similarity Score: 0.366313636302948
Document: Technology - Wikipedia, Page: 19, Cosine Similarity Score: 0.3553599715232849
Document: History of technology - Wikipedia, Page: 23, Cosine Similarity Score: 0.35477733612060547
Document: Technology - Wikipedia, Page: 18, Cosine Similarity Score: 0.34874439239501953
Similarity SSearch Latency: 0.022357702255249023
Max Marginal Relevance Search Results
Document: History - Wikipedia, Page: 8
Document: Technology - Wikipedia, Page: 25
Document: Technology - Wikipedia, Page: 20
Document: Technology - Wikipedia, Page: 5
Document: Science - Wikipedia, Page: 14
Document: Technology - Wikipedia, Page: 17
Document: Technology - Wikipedia, Page: 21
Document: Politics - Wikipedia, Page: 10
Document: History of technology - Wikipedia, Page: 0
Document: Politics - Wikipedia, Page: 17
```

```
Max Marginal Relevance Search Latency: 0.029781579971313477
================================================================
```

In [72]:
```
# Measuring and reporting search latency for different values of k (1, 5, 10)
# For k = 1, the average search latency for regular vector similarity search = 0.02408379316
# For k = 1, the average search latency for max marginal relevance search = 0.02657186985

# For k = 5, the average search latency for regular vector similarity search = 0.02417612075
# For k = 5, the average search latency for max marginal relevance search = 0.0267047882

# For k = 10, the average search latency for regular vector similarity search = 0.02616178989
# For k = 10, the average search latency for max marginal relevance search = 0.02793627977

# Analyzing the effect of different search types on result diversity (compare similarity vs MMR results for the same qu
"""
For k = 1, both search methods produced the same exact results(with respect to page number and overall document) for ea
"""
```

Out[72]:
```
'\nFor k = 1, both search methods produced the same exact results(with respect to page number and overall document) fo
r each query which makes sense since there's not really much room for diversity in the results when both search method
s can only retrieve one single result. For k = 5 and k = 10, the max marginal relevance search method results tend to
show higher diversity(results spread across more documents) while still maintaining decent relevance with the query. T
he max marginal relevance search method tends to reduce the repetition of chunks from the same document while increasi
ng the number of unique documents in the k retrieved chunks which is what the max marginal relevance search method was
designed to do in the first place.\n'
```

Measuring and reporting search latency for different values of k (1, 5, 10)

For k = 1, the average search latency for regular vector similarity search = 0.02408379316

For k = 1, the average search latency for max marginal relevance search = 0.02657186985

For k = 5, the average search latency for regular vector similarity search = 0.02417612075

For k = 5, the average search latency for max marginal relevance search = 0.0267047882

For k = 10, the average search latency for regular vector similarity search = 0.02616178989

For k = 10, the average search latency for max marginal relevance search = 0.02793627977

Analyzing the effect of different search types on result diversity (compare similarity vs MMR results for the same query)

For k = 1, both search methods produced the same exact results(with respect to page number and overall document) for each query which makes sense since there's not really much room for diversity in the results when both search methods can only retrieve one single result. For k = 5 and k = 10, the max marginal relevance search method results tend to show higher diversity(results spread across more documents) while still maintaining decent relevance with the query. The max marginal relevance search method tends to reduce the repetition of chunks

from the same document while increasing the number of unique documents in the k retrieved chunks which is what the max marginal relevance search method was designed to do in the first place.

In [73]:
```
# Demonstrating scenarios where MMR provides better coverage than similarity search
"""
For the query "economic policy impacts", from the results produced above, we can see that for k = 10, similarity search
"""
```

Out[73]:
```
'\nFor the query "economic policy impacts", from the results produced above, we can see that for k = 10, similarity se
arch retrieves chunks from just the History Wikipedia article and the Technology Wikipedia article. Whereas the max ma
rginal relevance search method retrieves chunks from the History Wikipedia article, Technology Wikipedia article, Scie
nce Wikipedia article, and the Politics Wikipedia article. The max marginal relevance search method is able to retriev
e chunks from the Science and Politics Wikipedia articles which the regular similarity search method missed even thoug
h they probably are relevant to the query thus showing how max marginal relevance search provided better coverage than
the regular similarity search method here. For the query “Olympic games history”, from the results produced above, we
can see that for k = 10, the similarity search and max marginal relevance search methods retrieve chunks from the same
two articles. However, similarity search tends to repeat the pages that it retrieves chunks from more often than the m
ax marginal relevance search method which does a better job of returning results from a more diverse range of pages fo
r any certain document thus showing another way in which max marginal relevance search provided better coverage than t
he regular similarity search method.\n'
```

Demonstrating scenarios where MMR provides better coverage than similarity search

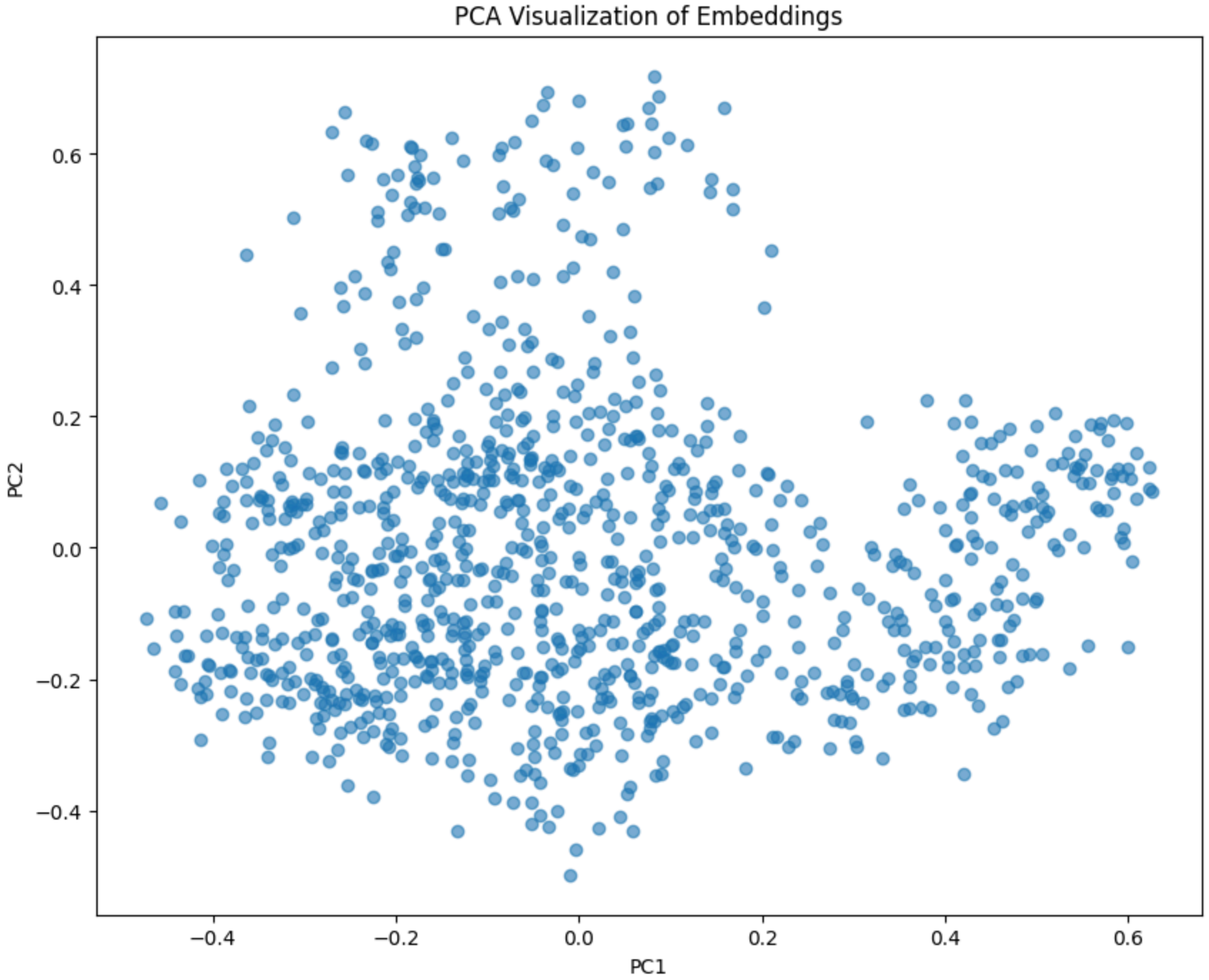
For the query "economic policy impacts", from the results produced above, we can see that for k = 10, similarity search retrieves chunks from just the History Wikipedia article and the Technology Wikipedia article. Whereas the max marginal relevance search method retrieves chunks from the History Wikipedia article, Technology Wikipedia article, Science Wikipedia article, and the Politics Wikipedia article. The max marginal relevance search method is able to retrieve chunks from the Science and Politics Wikipedia articles which the regular similarity search method missed even though they probably are relevant to the query thus showing how max marginal relevance search provided better coverage than the regular similarity search method here. For the query “Olympic games history”, from the results produced above, we can see that for k = 10, the similarity search and max marginal relevance search methods retrieve chunks from the same two articles. However, similarity search tends to repeat the pages that it retrieves chunks from more often than the max marginal relevance search method which does a better job of returning results from a more diverse range of pages for any certain document thus showing another way in which max marginal relevance search provided better coverage than the regular similarity search method.

In [75]:
```
# Creating visualizations of embedding similarities using dimensionality reduction
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
import numpy as np

# Get chunks for this chunk_size
splitter = RecursiveCharacterTextSplitter(
    chunk_size=CHUNK_SIZE,
    chunk_overlap=CHUNK_OVERLAP,
```

```
    add_start_index=True
)
chunks = splitter.split_documents(docs)

# Generate embeddings for each chunk using a pre-trained embedding model
# Store the embeddings in a vector database (e.g., Chroma) with proper metadata to enable cosine distance calculations
vs = Chroma.from_documents(
    documents=chunks,
    embedding=emb,
    persist_directory=PERSIST_DIR + "_visualizations_vector_store",
    collection_name=COLLECTION,
    collection_metadata={"hnsw:space": "cosine"}
)
vs.persist()
```

In [80]:

```
# Creating visualizations of embedding similarities using dimensionality reduction

# Extract embeddings for each chunk from Chroma vector database
data = vs.get(include=['embeddings'])
embeddings = np.array(data['embeddings'])

# Use PCA to reduce dimensionality of embeddings to 2 components
pca = PCA(n_components=2)
coords = pca.fit_transform(embeddings)

# Plot PC1 against PC2
plt.figure(figsize=(10, 8))
plt.scatter(coords[:, 0], coords[:, 1], alpha=0.6)
plt.title('PCA Visualization of Embeddings')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.show()
```



In [83]:
```
# Discussion of trade-offs between search accuracy and efficiency
"""
The pros of the regular similarity search method are that it's typically faster on average than MMR as seen in the resu
"""
```

Out[83]:
```
"\nThe pros of the regular similarity search method are that it's typically faster on average than MMR as seen in the
results produced above and simpler. It also delivers more accurate results than MMR for any query in theory. However,
even though MMR is slower, it offers more diverse/less-redundant results and better coverage of the available data/doc
uments. And although the accuracy of the results produced by MMR may not be as accurate as those produced by normal si
milarity search in theory, the added diversity provided by MMR enables MMR to generate better overall quality results
where users can get a better view of all the available data relevant to the search query instead of a more narrow view
where only information from a few of the available documents are present in the results.\n"
```

Discussion of trade-offs between search accuracy and efficiency

The pros of the regular similarity search method are that it's typically faster on average than MMR as seen in the results produced above and simpler. It also delivers more accurate results than MMR for any query in theory. However, even though MMR is slower, it offers more diverse/less-redundant results and better coverage of the available data/documents. And although the accuracy of the results produced by MMR may not be as accurate as those produced by normal similarity search in theory, the added diversity provided by MMR enables MMR to generate better overall quality results where users can get a better view of all the available data relevant to the search query instead of a more narrow view where only information from a few of the available documents are present in the results.

In [ ]:

Problem 3. PreTraining

Read the paper "Training Compute-Optimal Large Language Models" Hoffmann et al. [2022] carefully and answer the following questions. For each question, provide specific evidence and citations from the paper to support your answer.

Note: Your answers should be specific and include relevant numerical evidence where appropriate. For each answer, clearly indicate which sections or tables of the paper you are referencing to support your arguments

1. **(5 points)** The paper presents three different approaches to determine the optimal trade-off between

model size and number of training tokens. For Approach 2 (IsoFLOP profiles), what were the exponents a and b found for the relationships N_{opt} $\propto$C_aandD_{opt} $\propto$C_b$? How do these values differ from Kaplan et al.'s findings, and what is the practical implication of this difference for training large language models?

Answer: According to Table 2, for approach 2, the exponents a and b found for the relationships N_{opt} $\propto$C_aandD_{opt} $\propto$C_b$ were a = 0.49 and b = 0.51. These values differ from Kaplan et al.'s findings(a = 0.73 and b = 0.27) which are also shown in Table 2. So while the results of approach 2 indicate that the model size and amount of training data provided should scale equally with compute capacity, Kaplan et al.'s findings suggest that model size should increase much more than the amount of training data provided when the compute capacity is increased. Practically, this suggests that many large models have been haven't been trained on enough data to be compute optimal. Given a compute capacity, this paper suggests that it's more optimal(in terms of both compute capacity and overall model accuracy/performance) to train a smaller model with more training data.

2. (5 points) For a given compute budget of 576×10^{23} FLOPs (same as Gopher), what is the optimal model size and number of training tokens according to the paper's analysis? Compare this to Gopher's actual configuration.

Answer: According to Table 3, for a given compute budget of 5.76e23 FLOPs (same as Gopher), the optimal model has 67 billion parameters and 1.5 trillion training tokens. According to Table 1, Gopher's actual configuration was 280 billion parameters and around 300 billion training tokens. As a result, the paper estimates that the computationally optimal model would be 4-5 times as small and contain around 5 times more training tokens then Gopher's setup.

3. (5 points) Did Chinchilla's improvements in performance come at a higher computational cost compared to Gopher? Explain your answer using evidence from the paper about compute budget and model efficiency.

Answer: No, Chinchilla's improvements in performance did not come at a higher computational cost compared to Gopher. Evidence of this can be seen in section 4 of the paper which explictly states that "Both Chinchilla and Gopher have been trained for the same number of FLOPs but differ in the size of the model and the number of training tokens." In addition, Chinchilla's inference and fine-tuning costs are substantially lower. Evidence of this can again be seen in section 4 of the paper which explicitly states that "Due to being 4×smaller than Gopher, both the memory footprint and inference cost of Chinchilla are also smaller." As a result, these quotes from the paper about compute budget and model efficiency therefore explain how Chinchilla's performance gains did not require more pretraining FLOPs but from reallocating the same compute budget more optimally(using a smaller model and providing more training tokens).

4. (5 points) On the MMLU benchmark, what was Chinchilla's average accuracy and how did it compare to both Gopher and human expert performance? Cite specific numbers from the paper.

Answer: According to Table 6, Chinchilla's average 5-shot accuracy on the MMLU benchmark was 67.6%. This is 7.6% larger than Gopher's average 5-shot accuracy of 60% and around 22.2% smaller than the average human expert performance of 89.8% which are also explicitly stated in Table 6.

5. (5 points) According to the paper's analysis, what are the implications for training a 1 trillion parameter model? Would this be compute-optimal with current practices? Explain using evidence from Table 3 of the paper.

Answer: According to Table 3, a 1 trillion parameter model would require around 1.27e+26 FLOPs(221.3 times the Gopher standardized unit) and around 21.2 trillion training tokens to be provided for training. The paper further states in section 3.4 that "Unless one has a compute budget of 10e26 FLOPs (over 250× the compute used to train Gopher), a 1 trillion parameter model is unlikely to be the optimal model to train." On top of the compute budget required, the paper also states in section 3.4 that "the amount of training data that is projected to be needed is far beyond what is currently used to train large models." Overall, the provided evidence clearly shows how training a 1 trillion parameter model is not compute optimal with current practices due to the extremely high compute budget(number of FLOPs required) and the need for way more data than what's currently used to train large models.

In [ ]:

Question 4: Quantization

Please also consult the HW PDF to know the specific things you need to do.

Initial Setup

Before beginning the assignment, we import the CIFAR dataset, and train a simple convolutional neural network (CNN) to classify it.

In [1]:
```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

Reminder: set the runtime type to "GPU", or your code will run much more slowly on a CPU.

In [2]:
```
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')
```

Load training and test data from the CIFAR10 dataset.

In [3]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)
```

```
100%|██████████| 170M/170M [00:03<00:00, 47.8MB/s]
```

Define a simple CNN that classifies CIFAR images.

In [4]:
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5, bias=False)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5, bias=False)
        self.fc1 = nn.Linear(16 * 5 * 5, 120, bias=False)
        self.fc2 = nn.Linear(120, 84, bias=False)
        self.fc3 = nn.Linear(84, 10, bias=False)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net().to(device)
```

Train this CNN on the training dataset (this may take a few moments).

In [5]:
```
from torch.utils.data import DataLoader

def train(model: nn.Module, dataloader: DataLoader):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

    for epoch in range(2):  # loop over the dataset multiple times

        running_loss = 0.0
        for i, data in enumerate(dataloader, 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data

            inputs = inputs.to(device)
            labels = labels.to(device)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
```

```
            optimizer.step()

            # print statistics
            running_loss += loss.item()
            if i % 2000 == 1999:    # print every 2000 mini-batches
                print('[%d, %5d] loss: %.3f' %
                      (epoch + 1, i + 1, running_loss / 2000))
                running_loss = 0.0

    print('Finished Training')

def test(model: nn.Module, dataloader: DataLoader, max_samples=None) -> float:
    correct = 0
    total = 0
    n_inferences = 0

    with torch.no_grad():
        for data in dataloader:
            images, labels = data

            images = images.to(device)
            labels = labels.to(device)

            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            if max_samples:
                n_inferences += images.shape[0]
                if n_inferences > max_samples:
                    break

    return 100 * correct / total
```

In [6]:
```
train(net, trainloader)
```

```
[1,  2000] loss: 2.221
[1,  4000] loss: 1.901
[1,  6000] loss: 1.686
[1,  8000] loss: 1.568
[1, 10000] loss: 1.553
[1, 12000] loss: 1.516
[2,  2000] loss: 1.447
[2,  4000] loss: 1.410
[2,  6000] loss: 1.383
[2,  8000] loss: 1.350
[2, 10000] loss: 1.335
[2, 12000] loss: 1.337
Finished Training
```

Now that the CNN has been trained, let's test it on our test dataset.

In [7]:
```
score = test(net, testloader)
print('Accuracy of the network on the test images: {}%'.format(score))
```

```
Accuracy of the network on the test images: 53.22%
```

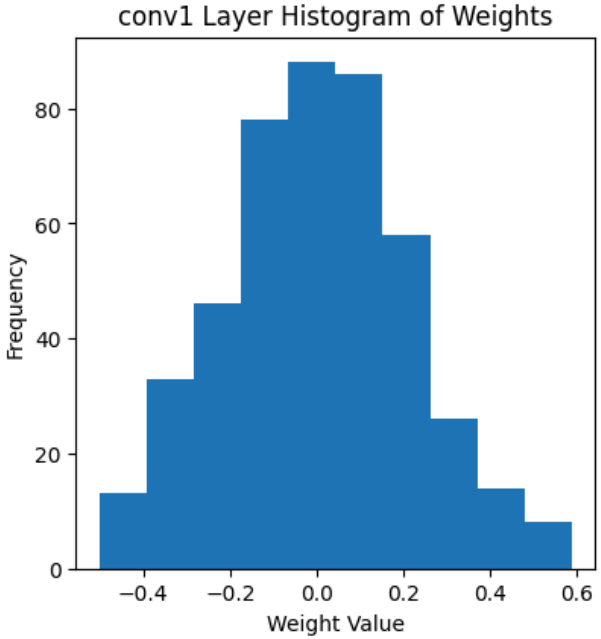
In [8]:
```
from copy import deepcopy

# A convenience function which we use to copy CNNs
def copy_model(model: nn.Module) -> nn.Module:
    result = deepcopy(model)

    # Copy over the extra metadata we've collected which copy.deepcopy doesn't capture
    if hasattr(model, 'input_activations'):
        result.input_activations = deepcopy(model.input_activations)

    for result_layer, original_layer in zip(result.children(), model.children()):
        if isinstance(result_layer, nn.Conv2d) or isinstance(result_layer, nn.Linear):
            if hasattr(original_layer.weight, 'scale'):
                result_layer.weight.scale = deepcopy(original_layer.weight.scale)
            if hasattr(original_layer, 'activations'):
                result_layer.activations = deepcopy(original_layer.activations)
            if hasattr(original_layer, 'output_scale'):
                result_layer.output_scale = deepcopy(original_layer.output_scale)

    return result
```

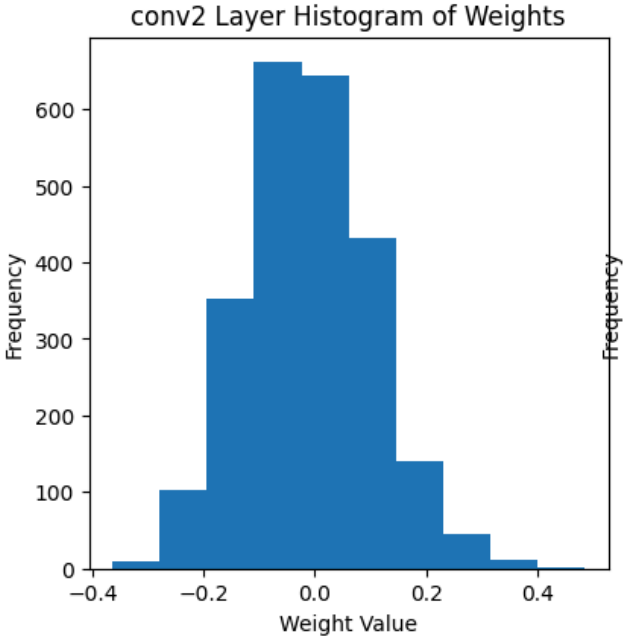
Question 1: Visualize Weights

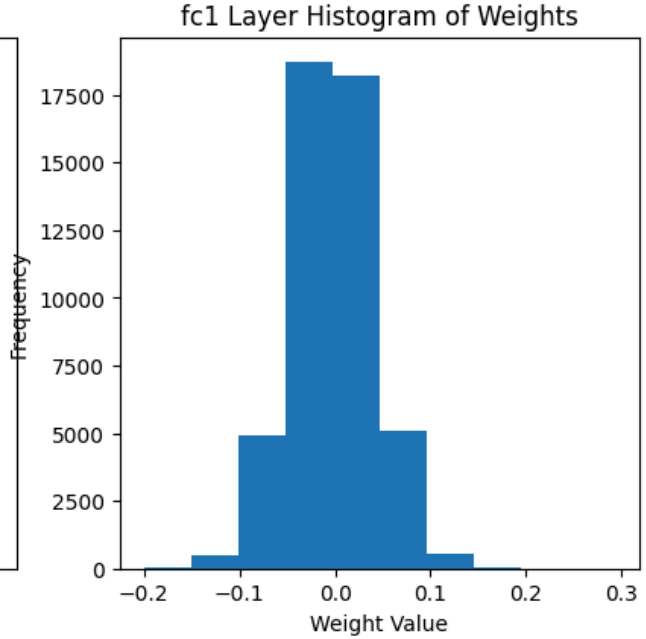
In [9]:
```
import matplotlib.pyplot as plt
import numpy as np
```

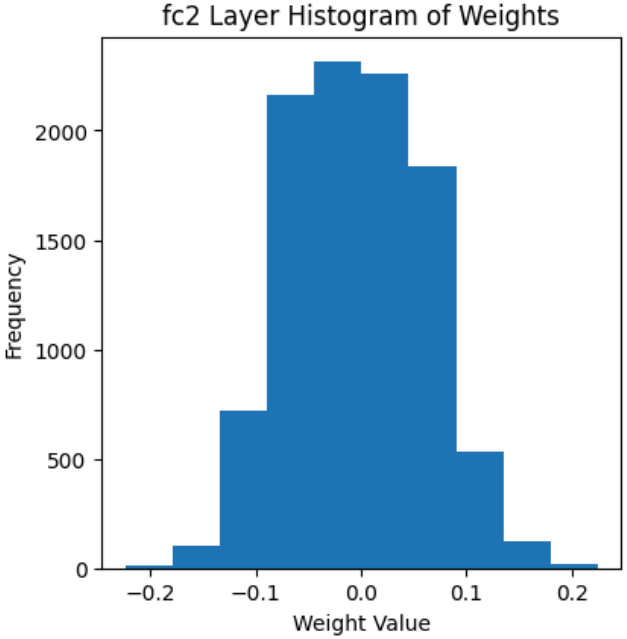
In [13]:
```
# Plot histograms of weights for each layer

# Get a flattened vector of the weights for each layer
layers = {
    'conv1': net.conv1.weight.data.cpu().view(-1),
    'conv2': net.conv2.weight.data.cpu().view(-1),
    'fc1': net.fc1.weight.data.cpu().view(-1),
    'fc2': net.fc2.weight.data.cpu().view(-1),
    'fc3': net.fc3.weight.data.cpu().view(-1)
}
statistics = {
    'conv1': {},
    'conv2': {},
    'fc1': {},
    'fc2': {},
    'fc3': {}
}

# Create subplots for histograms
fig, axes = plt.subplots(2, 3, figsize=(15, 10))

# Convert multi-dimensional subplot axes array into a 1D array
axes = axes.flatten()
i = 0
for key in layers:
    # Get layer name and weights
    layer_name = key
    layer_weights = layers[key]

    # Convert weights into a numpy array
    weights = layer_weights.numpy()

    # Store statistics(range, mean, std) for layer
    weights_range = (weights.min(), weights.max())
    statistics[key]['range'] = weights_range
    weights_mean = weights.mean()
    statistics[key]['mean'] = weights_mean
    weights_std = weights.std()
    statistics[key]['std'] = weights_std

    # Plot histogram of weights for layer
    axes[i].hist(weights)
    axes[i].set_title(layer_name + " Layer Histogram of Weights")
    axes[i].set_xlabel('Weight Value')
```

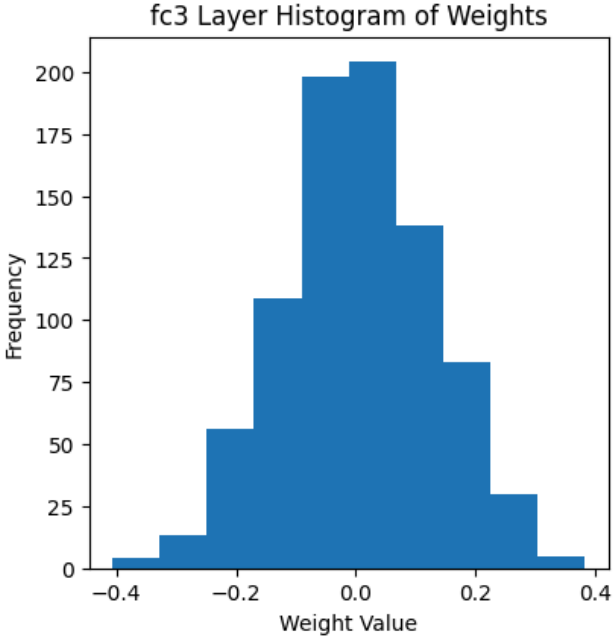
```
    axes[i].set_ylabel('Frequency')
    i += 1

# Remove the extra subplot
axes[-1].axis('off')
plt.show()

# Calculate and print statistical measures (range, mean, standard deviation) for each layer
for key in statistics:
    print(key)
    print("Range: " + str(statistics[key]['range']))
    print("Mean: " + str(statistics[key]['mean']))
    print("Standard Deviation: " + str(statistics[key]['std']))
    print()
    print()
```


conv1 Layer Histogram of Weights
Frequency
Weight Value
0
20
40
60
80
−0.4
−0.2
0.0
0.2
0.4
0.6


conv2 Layer Histogram of Weights
Frequency
Weight Value
0
100
200
300
400
500
600
−0.4
−0.2
0.0
0.2
0.4


fc1 Layer Histogram of Weights
Frequency
Weight Value
0
2500
5000
7500
10000
12500
15000
17500
−0.2
−0.1
0.0
0.1
0.2
0.3


fc2 Layer Histogram of Weights
Frequency
Weight Value
0
500
1000
1500
2000
−0.2
−0.1
0.0
0.1
0.2


fc3 Layer Histogram of Weights
Frequency
Weight Value
0
25
50
75
100
125
150
175
200
−0.4
−0.2
0.0
0.2
0.4

```
conv1
Range: (-0.50060385, 0.5876992)
Mean: 0.0012284977
Standard Deviation: 0.21102612


conv2
Range: (-0.3643383, 0.48578006)
Mean: -0.010588029
Standard Deviation: 0.11290347


fc1
Range: (-0.20037553, 0.29379752)
Mean: -0.0024880795
Standard Deviation: 0.04287504


fc2
Range: (-0.22354394, 0.22480442)
Mean: -0.0031889004
Standard Deviation: 0.06426893


fc3
Range: (-0.4064013, 0.38186222)
Mean: 0.0037882142
Standard Deviation: 0.12598744
```

In [15]:
```
# Analyze distribution patterns across layers
"""
The conv1 layer has the largest range and the highest variance. The conv2 layer has a narrower range and a variance tha
"""
```

Out[15]:
```
'\nThe conv1 layer has the largest range and the highest variance. The conv2 layer has a narrower range and a variance
that nearly havles the variance in the conv1 layer. The fc1 layer has an even tigher range and the smallest variance o
ut of all the layers. The fc2 layer has a tigher range than the fc1 layer but a slightly larger variance at the same t
ime. And the fc3 layer has a significantly wider range and almost double the variance of the fc2 layer.\n'
```

Analyze distribution patterns across layers

The conv1 layer has the largest range and the highest variance. The conv2 layer has a narrower range and a variance that nearly havles the variance in the conv1 layer. The fc1 layer has an even tigher range and the smallest variance out of all the layers. The fc2 layer has a tigher range than the fc1 layer but a slightly larger variance at the same time. And the fc3 layer has a significantly wider range and almost double the variance of the fc2 layer.

Question 2: Quantize Weights

In [14]:
```
net_q2 = copy_model(net)
```

In [18]:
```
from typing import Tuple

def quantized_weights(weights: torch.Tensor) -> Tuple[torch.Tensor, float]:
    '''
    Quantize the weights so that all values are integers between -128 and 127.
    You may want to use the total range, 3-sigma range, or some other range when
    deciding just what factors to scale the float32 values by.

    Parameters:
    weights (Tensor): The unquantized weights

    Returns:
    (Tensor, float): A tuple with the following elements:
                       * The weights in quantized form, where every value is an integer between -128 and 127.
                         The "dtype" will still be "float", but the values themselves should all be integers.
                       * The scaling factor that your weights were multiplied by.
                         This value does not need to be an 8-bit integer.
    '''

    # ADD YOUR CODE HERE
    # 3-Sigma clipping
    mean = weights.mean()
    std = weights.std()
    new_min = mean - (3 * std)
    new_max = mean + (3 * std)
    # Defining the range as mean +/- 3*std
    new_range = new_max - new_min
    # # Calculate scale((127 - -128) / range)
    scale = 255.0 / new_range
    # Clip weights, center them, and then scale them
    clipped_weights = torch.clamp(weights, new_min, new_max)
    result = ((clipped_weights - new_min) * scale - 128).round()
    return torch.clamp(result, min=-128, max=127), scale
```

In [49]:
```
# 3-Sigma Quantization strategy explanation
"""
The 3-Sigma Quantization strategy essentially just clips outliers in the weights/activations data that are beyong 3 sta
"""
```

Out[49]:
```
'\nThe 3-Sigma Quantization strategy essentially just clips outliers in the weights/activations data that are beyong 3 standard deviations of the mean which still ensures that the remaining data covers around 99.7%(assuming it follows a normal distribution).\n'
```

3-Sigma Quantization strategy explanation

The 3-Sigma Quantization strategy essentially just clips outliers in the weights/activations data that are beyond 3 standard deviations of the mean which still ensures that the remaining data covers around 99.7%(assuming it follows a normal distribution).

In [19]:
```
def quantize_layer_weights(model: nn.Module):
    for layer in model.children():
        if isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear):
            q_layer_data, scale = quantized_weights(layer.weight.data)
            q_layer_data = q_layer_data.to(device)

            layer.weight.data = q_layer_data
            layer.weight.scale = scale

            if (q_layer_data < -128).any() or (q_layer_data > 127).any():
                raise Exception("Quantized weights of {} layer include values out of bounds for an 8-bit signed integer
            if (q_layer_data != q_layer_data.round()).any():
                raise Exception("Quantized weights of {} layer include non-integer values".format(layer.__class__.__nam

quantize_layer_weights(net_q2)
```

In [20]:
```
score = test(net_q2, testloader)
print('Accuracy of the network after quantizing all weights: {}%'.format(score))
```

```
Accuracy of the network after quantizing all weights: 52.22%
```

Impact on model accuracy

After quantizing all weights to be within int8 range, the accuracy of the network only dropped by 1% on the test set. This shows that 32-bit floating point data may not be necessary during inference since quantizing all the weights to be within int8 range during inference achieved almost the same performance.

Question 3: Visualize Activations

In [21]:
```
def register_activation_profiling_hooks(model: Net):
    model.input_activations = np.empty(0)
    model.conv1.activations = np.empty(0)
    model.conv2.activations = np.empty(0)
    model.fc1.activations = np.empty(0)
    model.fc2.activations = np.empty(0)
```

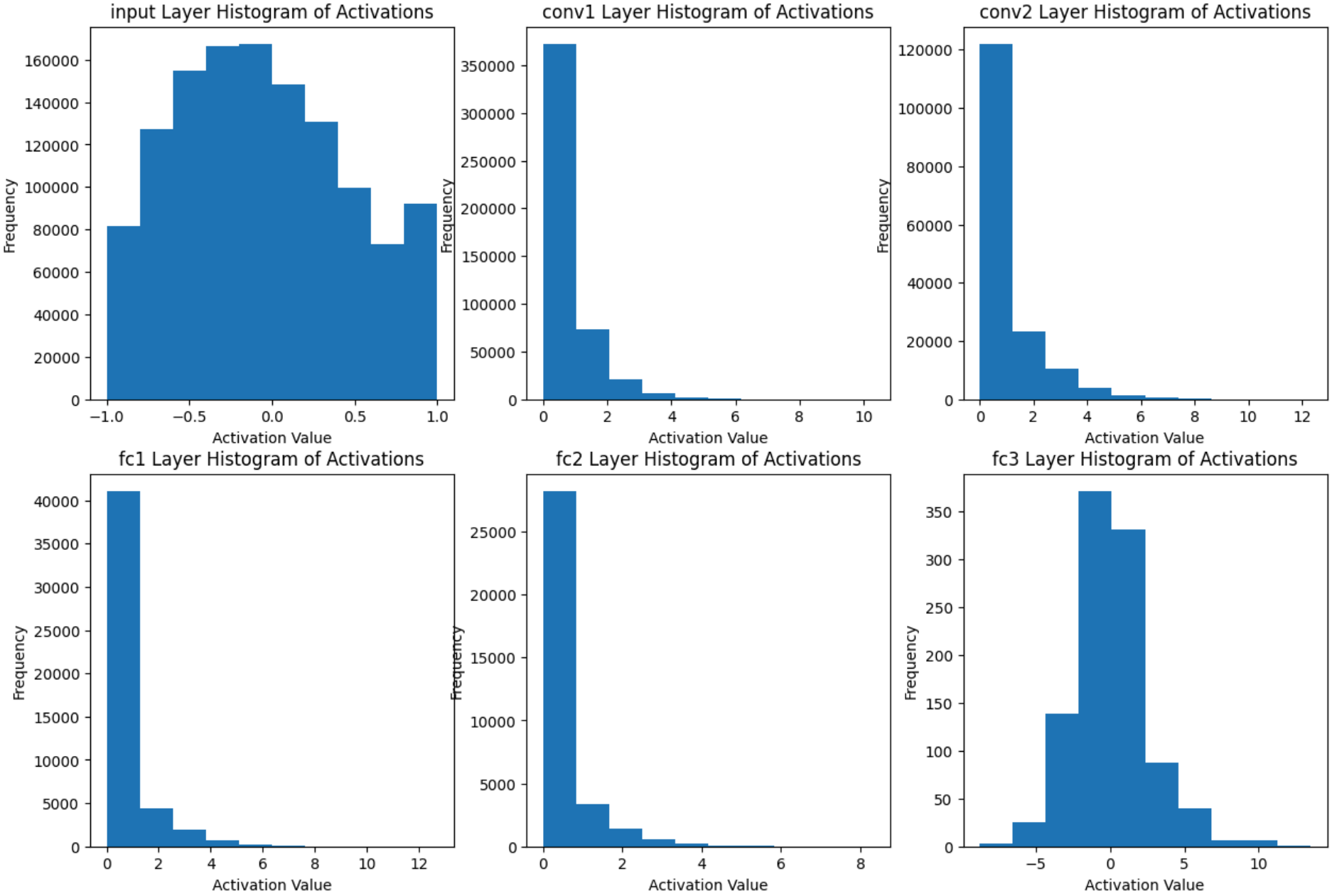
```
    model.fc3.activations = np.empty(0)

    model.profile_activations = True

    def conv1_activations_hook(layer, x, y):
        if model.profile_activations:
            model.input_activations = np.append(model.input_activations, x[0].cpu().view(-1))
    model.conv1.register_forward_hook(conv1_activations_hook)

    def conv2_activations_hook(layer, x, y):
        if model.profile_activations:
            model.conv1.activations = np.append(model.conv1.activations, x[0].cpu().view(-1))
    model.conv2.register_forward_hook(conv2_activations_hook)

    def fc1_activations_hook(layer, x, y):
        if model.profile_activations:
            model.conv2.activations = np.append(model.conv2.activations, x[0].cpu().view(-1))
    model.fc1.register_forward_hook(fc1_activations_hook)

    def fc2_activations_hook(layer, x, y):
        if model.profile_activations:
            model.fc1.activations = np.append(model.fc1.activations, x[0].cpu().view(-1))
    model.fc2.register_forward_hook(fc2_activations_hook)

    def fc3_activations_hook(layer, x, y):
        if model.profile_activations:
            model.fc2.activations = np.append(model.fc2.activations, x[0].cpu().view(-1))
            model.fc3.activations = np.append(model.fc3.activations, y[0].cpu().view(-1))
    model.fc3.register_forward_hook(fc3_activations_hook)
```

In [22]:
```
net_q3 = copy_model(net)
register_activation_profiling_hooks(net_q3)

# Run through the training dataset again while profiling the input and output activations this time
# We don't actually have to perform gradient descent for this, so we can use the "test" function
test(net_q3, trainloader, max_samples=400)
net_q3.profile_activations = False
```

In [23]:
```
input_activations = net_q3.input_activations
conv1_output_activations = net_q3.conv1.activations
conv2_output_activations = net_q3.conv2.activations
fc1_output_activations = net_q3.fc1.activations
fc2_output_activations = net_q3.fc2.activations
fc3_output_activations = net_q3.fc3.activations
```

In [27]:
```
# Plot distributions of activations for each layer
```

```python
activations = {
    'input': input_activations,
    'conv1': conv1_output_activations,
    'conv2': conv2_output_activations,
    'fc1': fc1_output_activations,
    'fc2': fc2_output_activations,
    'fc3': fc3_output_activations
}

statistics = {
    'input': {},
    'conv1': {},
    'conv2': {},
    'fc1': {},
    'fc2': {},
    'fc3': {}
}

# Create subplots for histograms
fig, axes = plt.subplots(2, 3, figsize=(15, 10))

# Convert multi-dimensional subplot axes array into a 1D array
axes = axes.flatten()
i = 0
for key in activations:
    # Get layer name and activations
    layer_name = key
    layer_activations = activations[key]


    # Store statistics(full range, 3-sigma range) for layer
    activations_range = (layer_activations.min(), layer_activations.max())
    statistics[key]['full_range'] = activations_range
    activations_mean = layer_activations.mean()
    activations_std = layer_activations.std()
    statistics[key]['3-sigma_range'] = (activations_mean - 3 * activations_std, activations_mean + 3 * activations_std)

    # Plot histogram of activations for layer
    axes[i].hist(layer_activations)
    axes[i].set_title(layer_name + " Layer Histogram of Activations")
    axes[i].set_xlabel('Activation Value')
    axes[i].set_ylabel('Frequency')
    i += 1

plt.show()
```



In [28]:
```python
# Calculate and print statistical measures (full range, 3-sigma range) for each layer
# Document distribution patterns across network depth
for key in statistics:
    print(key)
    print("Full Range: " + str(statistics[key]['full_range']))
    print("3 Sigma Range: " + str(statistics[key]['3-sigma_range']))
    print()
    print()
```

```
input
Full Range: (-1.0, 1.0)
3 Sigma Range: (-1.6093617983624198, 1.5012448407394523)


conv1
Full Range: (0.0, 10.323747634887695)
3 Sigma Range: (-1.9293176755119321, 3.1215164079295104)


conv2
Full Range: (0.0, 12.310510635375977)
3 Sigma Range: (-2.8966380566793086, 4.453709713946421)


fc1
Full Range: (0.0, 12.71954345703125)
3 Sigma Range: (-2.67623602324843, 3.6837435509826015)


fc2
Full Range: (0.0, 8.341231346130371)
3 Sigma Range: (-1.8511606899638595, 2.5923604948241312)


fc3
Full Range: (-8.797361373901367, 13.489840507507324)
3 Sigma Range: (-7.532655283940421, 7.687392734856851)
```

Analyze distribution patterns across layers

According to the histrograms, the input layer activations have a kind of evenly distributed distribution while the conv1, conv2, fc1, and fc2 layer activations have distributions that are very positively skewed to the right. The fc3 layer activations have a pretty roughly symmetric distribution though. The input layer activations have a pretty standard range of (-1, 1). The conv1 layer activations are very sparse and positively skewed because of the ReLU function with a slightly larger 3 sigma range than the input layer activations and a much larger full range. The conv2 layer activations are even more positively skewed and have a slightly wider 3 sigma range and full range than the conv2 layer. The fc1 layer activations have a slightly larger full range but tighter 3 sigma range while also still being as positively skewed as the conv2 layer. The fc2 layer activations have a noticeably smaller full range and tighter 3 sigma range and are less positively skewed than the fc1 layer activations. Finally, the fc3 layer activations are more symmetric with a wider overall range than any of the previous layers due to the ReLU activation not being applied to this final layer.

Question 4: Quantize Activations

In [35]:

```
from typing import List

class NetQuantized(nn.Module):
    def __init__(self, net_with_weights_quantized: nn.Module):
        super(NetQuantized, self).__init__()

        net_init = copy_model(net_with_weights_quantized)

        self.conv1 = net_init.conv1
        self.pool = net_init.pool
        self.conv2 = net_init.conv2
        self.fc1 = net_init.fc1
        self.fc2 = net_init.fc2
        self.fc3 = net_init.fc3

        for layer in self.conv1, self.conv2, self.fc1, self.fc2, self.fc3:
            def pre_hook(l, x):
                x = x[0]
                if (x < -128).any() or (x > 127).any():
                    raise Exception("Input to {} layer is out of bounds for an 8-bit signed integer".format(l.__class__
                if (x != x.round()).any():
                    raise Exception("Input to {} layer has non-integer values".format(l.__class__.__name__))

            layer.register_forward_pre_hook(pre_hook)

        # Calculate the scaling factor for the initial input to the CNN
        self.input_activations = net_with_weights_quantized.input_activations
        self.input_scale = NetQuantized.quantize_initial_input(self.input_activations)

        # Calculate the output scaling factors for all the layers of the CNN
        preceding_layer_scales = []
        for layer in self.conv1, self.conv2, self.fc1, self.fc2, self.fc3:
            layer.output_scale = NetQuantized.quantize_activations(layer.activations, layer.weight.scale, self.input_sc
            preceding_layer_scales.append((layer.weight.scale, layer.output_scale))

    @staticmethod
    def quantize_initial_input(pixels: np.ndarray) -> float:
        '''
        Calculate a scaling factor for the images that are input to the first layer of the CNN.

        Parameters:
        pixels (ndarray): The values of all the pixels which were part of the input image during training

        Returns:
        float: A scaling factor that the input should be multiplied by before being fed into the first layer.
               This value does not need to be an 8-bit integer.
```

```
        '''

        # 3-Sigma clipping
        mean = pixels.mean()
        std = pixels.std()
        new_min = mean - (3 * std)
        new_max = mean + (3 * std)
        # Defining the range as mean +/- 3*std
        new_range = new_max - new_min
        # Calculate scale((127 - -128) / range)
        scale = 255.0 / new_range
        return scale

    @staticmethod
    def quantize_activations(activations: np.ndarray, n_w: float, n_initial_input: float, ns: List[Tuple[float, float]]
        '''
        Calculate a scaling factor to multiply the output of a layer by.

        Parameters:
        activations (ndarray): The values of all the pixels which have been output by this layer during training
        n_w (float): The scale by which the weights of this layer were multiplied as part of the "quantize_weights" fur
        n_initial_input (float): The scale by which the initial input to the neural network was multiplied
        ns ([(float, float)]): A list of tuples, where each tuple represents the "weight scale" and "output scale" (in

        Returns:
        float: A scaling factor that the layer output should be multiplied by before being fed into the first layer.
               This value does not need to be an 8-bit integer.
        '''

        # Calculate the cumulative input scale to this layer
        cumulative_input_scale = n_initial_input
        for weight_scale, output_scale in ns:
            # Each layer multiplies by weight_scale and divides by output_scale
            cumulative_input_scale *= (weight_scale / output_scale)

        # The scale of the output would be: cumulative_input_scale * n_w
        original_output_scale = cumulative_input_scale * n_w

        # 3-Sigma clipping
        mean = activations.mean()
        std = activations.std()
        new_min = mean - (3 * std)
        new_max = mean + (3 * std)
        # Defining the range as mean +/- 3*std
        new_range = new_max - new_min
        # Calculate scale((127 - -128) / range)
        new_scale = 255.0 / new_range
```

```
        # The output scale needed is: original_output_scale / new_scale
        output_scale = original_output_scale / new_scale
        return output_scale

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # You can access the output activation scales like this:
        #   fc1_output_scale = self.fc1.output_scale

        # To make sure that the outputs of each layer are integers between -128 and 127, you may need to use the follow
        #   * torch.Tensor.round
        #   * torch.clamp

        # scale input and clamp input to [-128, 127]
        x = x * self.input_scale
        x = torch.clamp(x.round(), min=-128, max=127)

        # Divide output of conv1 layer by its output scale and clamp output to [-128, 127]
        x = self.conv1(x)
        x = x / self.conv1.output_scale
        x = torch.clamp(x.round(), min=-128, max=127)
        # ReLU
        x = torch.clamp(x, min=0)
        # Pooling
        x = self.pool(x)

        # Divide output of conv2 layer by its output scale and clamp output to [-128, 127]
        x = self.conv2(x)
        x = x / self.conv2.output_scale
        x = torch.clamp(x.round(), min=-128, max=127)
        # ReLU
        x = torch.clamp(x, min=0)
        # Pooling
        x = self.pool(x)

        # Flatten
        x = x.view(-1, 16 * 5 * 5)

        # Divide output of fc1 layer by its output scale and clamp output to [-128, 127]
        x = self.fc1(x)
        x = x / self.fc1.output_scale
        x = torch.clamp(x.round(), min=-128, max=127)
        # ReLU
        x = torch.clamp(x, min=0)  # ReLU

        # Divide output of fc2 layer by its output scale and clamp output to [-128, 127]
        x = self.fc2(x)
        x = x / self.fc2.output_scale
        x = torch.clamp(x.round(), min=-128, max=127)
```

```
        # ReLU
        x = torch.clamp(x, min=0)

        # Return output of fc3 layer
        x = self.fc3(x)
        x = x / self.fc3.output_scale
        x = torch.clamp(x.round(), min=-128, max=127)
        # ReLU
        x = torch.clamp(x, min=0)

        return x
```

In [36]:
```
# Merge the information from net_q2 and net_q3 together
net_init = copy_model(net_q2)
net_init.input_activations = deepcopy(net_q3.input_activations)
for layer_init, layer_q3 in zip(net_init.children(), net_q3.children()):
    if isinstance(layer_init, nn.Conv2d) or isinstance(layer_init, nn.Linear):
        layer_init.activations = deepcopy(layer_q3.activations)

net_quantized = NetQuantized(net_init)
```

In [37]:
```
score = test(net_quantized, testloader)
print('Accuracy of the network after quantizing both weights and activations: {}%'.format(score))
```

```
Accuracy of the network after quantizing both weights and activations: 49.79%
```

Impact on model accuracy

After quantizing all weights and activations to be within int8 range, the accuracy of the network still only dropped by less than 3% on the test set. This shows that 32-bit floating point data may not be necessary during inference since quantizing all the weights and activations to be within int8 range during inference achieved pretty comparable performance.

Question 5: Quantize Biases

In [38]:
```
class NetWithBias(nn.Module):
    def __init__(self):
        super(NetWithBias, self).__init__()

        self.conv1 = nn.Conv2d(3, 6, 5, bias=False)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5, bias=False)
        self.fc1 = nn.Linear(16 * 5 * 5, 120, bias=False)
        self.fc2 = nn.Linear(120, 84, bias=False)
        self.fc3 = nn.Linear(84, 10, bias=True)
```

```
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net_with_bias = NetWithBias().to(device)
```

In [39]:
```
train(net_with_bias, trainloader)
```

```
[1,  2000] loss: 2.207
[1,  4000] loss: 1.895
[1,  6000] loss: 1.704
[1,  8000] loss: 1.614
[1, 10000] loss: 1.556
[1, 12000] loss: 1.519
[2,  2000] loss: 1.448
[2,  4000] loss: 1.417
[2,  6000] loss: 1.379
[2,  8000] loss: 1.370
[2, 10000] loss: 1.318
[2, 12000] loss: 1.329
Finished Training
```

In [40]:
```
score = test(net_with_bias, testloader)
print('Accuracy of the network (with a bias) on the test images: {}%'.format(score))
```

```
Accuracy of the network (with a bias) on the test images: 52.55%
```

In [41]:
```
register_activation_profiling_hooks(net_with_bias)
test(net_with_bias, trainloader, max_samples=400)
net_with_bias.profile_activations = False
```

In [42]:
```
net_with_bias_with_quantized_weights = copy_model(net_with_bias)
quantize_layer_weights(net_with_bias_with_quantized_weights)

score = test(net_with_bias_with_quantized_weights, testloader)
print('Accuracy of the network on the test images after all the weights are quantized but the bias isn\'t: {}%'.format(
```

```
Accuracy of the network on the test images after all the weights are quantized but the bias isn't: 48.09%
```

Impact on model accuracy

After quantizing all weights to be within int8 range but not the bias, the accuracy of the network dropped by more than 4% on the test set.

This shows that only quantizing all the weights to be within int8 range without quantizing the bias in some way hurts the performance more

during inference.

In [43]:

```
class NetQuantizedWithBias(NetQuantized):
    def __init__(self, net_with_weights_quantized: nn.Module):
        super(NetQuantizedWithBias, self).__init__(net_with_weights_quantized)

        preceding_scales = [(layer.weight.scale, layer.output_scale) for layer in self.children() if isinstance(layer,

        self.fc3.bias.data = NetQuantizedWithBias.quantized_bias(
            self.fc3.bias.data,
            self.fc3.weight.scale,
            self.input_scale,
            preceding_scales
        )

        if (self.fc3.bias.data < -2147483648).any() or (self.fc3.bias.data > 2147483647).any():
            raise Exception("Bias has values which are out of bounds for an 32-bit signed integer")
        if (self.fc3.bias.data != self.fc3.bias.data.round()).any():
            raise Exception("Bias has non-integer values")

    @staticmethod
    def quantized_bias(bias: torch.Tensor, n_w: float, n_initial_input: float, ns: List[Tuple[float, float]]) -> torch.
        '''
        Quantize the bias so that all values are integers between -2147483648 and 2147483647.

        Parameters:
        bias (Tensor): The floating point values of the bias
        n_w (float): The scale by which the weights of this layer were multiplied
        n_initial_input (float): The scale by which the initial input to the neural network was multiplied
        ns ([(float, float)]): A list of tuples, where each tuple represents the "weight scale" and "output scale" (in

        Returns:
        Tensor: The bias in quantized form, where every value is an integer between -2147483648 and 2147483647.
                The "dtype" will still be "float", but the values themselves should all be integers.
        '''

        # Calculate the cumulative input scale to this layer
        cumulative_input_scale = n_initial_input
        for weight_scale, output_scale in ns:
            # Each layer multiplies by weight_scale and divides by output_scale
            cumulative_input_scale *= (weight_scale / output_scale)

        # The scale of the bias would be: cumulative_input_scale * n_w
        bias_scale = cumulative_input_scale * n_w

        # Round and clamp the quantized_bias to range [-2147483648, 2147483647]
        quantized_bias = (bias * bias_scale).round()
```

```
        quantized_bias = torch.clamp(quantized_bias, min=-2147483648, max=2147483647)
        return quantized_bias
```

In [44]:
```
net_quantized_with_bias = NetQuantizedWithBias(net_with_bias_with_quantized_weights)
```

In [45]:
```
score = test(net_quantized_with_bias, testloader)
print('Accuracy of the network on the test images after all the weights and the bias are quantized: {}%'.format(score))
```

```
Accuracy of the network on the test images after all the weights and the bias are quantized: 49.93%
```

Impact on model accuracy

After quantizing all weights to be within int8 range and quantizing the bias to be within the 32-bit range, the accuracy of the network dropped by less. This shows that only quantizing all the weights to be within int8 range and quantizing the bias in some way does a better job of achieving a comparable performance during inference.

Discussion of trade-offs

Overall, the results from this experiment show that post-training quantization techniques for inference actually do a pretty good job of maintaining the performance of small pre-trained models while also saving computentionally expensive cost in terms of area, performance, and energy. So post-training quantization is actually worth it and high precision might not be necessary in these cases.

In [ ]:
```
```