4-th Exercise - Embeddings, Vector Databases & Search

We will use FAISS and CLIP to create and search pre-trained embedding vectors.

First, we need to install required libraries:

- CLIP: For generating embeddings
- FAISS: For indexing

In [1]:
```
import sys

# Uncomment as needed
# !{sys.executable} -m pip install --no-input openai-clip

# Either use CPU-only version
# !conda install --yes --prefix {sys.prefix}  -c pytorch faiss-cpu=1.9.0

# Alternatively use the GPU(+CPU) version
# !conda install --yes --prefix {sys.prefix} -c pytorch -c nvidia faiss-gpu=1.9.

# !conda install --yes --prefix {sys.prefix}
```

Load required libraries

In [2]:
```
import os
# Avoid different versions of numpy make kernel collaps
os.environ['KMP_DUPLICATE_LIB_OK']='True'

import torch
import faiss
import clip
import json
import requests

from PIL import Image
from io import BytesIO

import time
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from matplotlib.patches import Rectangle

from tqdm import tqdm
```

Some helper methods for testing results

In [3]:
```
images_URL = "http://vision.stanford.edu/aditya86/ImageNetDogs/images/"
```

```
def load_embeddings(embeddings_path):
    df = pd.read_csv(embeddings_path, sep=';', compression='gzip', index_col=0)

    # Next, we will convert the embeddings String-column into a numpy-vector
    start_time = time.time()
    data = df["embedding"].apply(json.loads).values
    embeddings = np.zeros((data.shape[0], len(data[0])), dtype=np.float32)
    for i, d in enumerate(data):
        embeddings[i] = d

    end_time = time.time()
    print(f"Time to load {np.round(end_time-start_time,3)}s")

    return df, embeddings


# Calculate Precision@k
def precision_at_k(indices, k):
    y_true = df.iloc[indices[:,0]]["class"].values
    y_pred = df.iloc[indices.flatten()]["class"].values.reshape(indices.shape[0]
    precisions = []
    # Check how many of the retrieved items are relevant
    for i, pred in enumerate(y_pred[:, :k]):
        relevant = np.sum(y_true[i] == pred)
        precisions.append(relevant / k)
    return np.mean(precisions)

# Plot search results
def plot_results(query_file_names, indices, k):
    if indices is not None:
        for pos, ind in enumerate(indices):
            fig, ax = plt.subplots(1, k+1, figsize=(3*(k+1),4))

            if query_file_names:
                path_query = f'{query_file_names[pos]}'
                if os.path.isfile(path_query):
                    ax[0].imshow(mpimg.imread(path_query))
                    ax[0].axis('off')
                    ax[0].set_title("Query")
                elif len(query_file_names) > pos:
                    # ax[0].set_title()
                    ax[0].text(0.5, 0.5,
                            f"{query_file_names[pos]}", fontsize=20,
                        ha='center', va='center')
                    ax[0].axis('off')
            else:
                ax[0].remove()

            for i in range(0, k):
                d_train = df.iloc[ind[i]]

                url = f'{images_URL}/{d_train["dir"]}/{d_train["filename"]}.jpg'
                response = requests.get(url, stream=True)
                if response.status_code == 200:
                    img = Image.open(BytesIO(response.content))
                    ax[1+i].imshow(img)
                    ax[1+i].axis('off')
                    ax[1+i].set_title(f"{i+1}-NN: " + d_train["class"])

                    xy = (d_train["xmin"], d_train["ymin"])
```

```
                    width = d_train["xmax"]-d_train["xmin"]
                    height = d_train["ymax"]-d_train["ymin"]
                    rect = Rectangle(xy, width, height, edgecolor='white', fill=
                    ax[1+i].add_patch(rect)
                    ax[1+i].text(xy[0], xy[1], d_train["class"], fontsize=12, co
                                 verticalalignment='bottom', horizontalalignment

                else:
                    ax[i+1].remove()

            plt.tight_layout()
            plt.show()

def plot_precision_at_k(indices, k):
    precisions = [precision_at_k(indices, i) for i in range (1,k+1)]
    for i, precision in enumerate(precisions):
        print(f"precision@{i}: {precision}")

    fig, ax = plt.subplots(figsize=(8,4))
    ax.set_title("Precision@k")
    ax.plot(np.arange(1,k+1, dtype=np.int32), precisions)
    ax.set_xlabel("k")
    ax.set_ylabel("Precision@k")
    plt.show()
```

Load Embeddings and Meta-Data

You must download the requires files first, and adapt paths:

- `queries/` : Queries needed for Tasks 1-4. Load from Moodle
- `data/vectors.csv.gz` : Embeddings needed for Tasks 1-5. Load from Moodle

In [4]:
```
### modify this path, if needed!
queries_start_path = "queries"
embeddings_path = 'data/vectors.csv.gz'

df, embeddings = load_embeddings(embeddings_path)
embedding_dim = embeddings.shape[1]
df.head()
```

```
Time to load 3.362s
```

Out[4]:

	filename	dir	class	pose	xmin	ymin	xmax	ymax	
0	n02097658_26	n02097658-silky_terrier	silky_terrier	Unspecified	41	30	296	398	[0.2 0.2
1	n02097658_4869	n02097658-silky_terrier	silky_terrier	Unspecified	43	270	321	498	[-0. -0.
2	n02097658_595	n02097658-silky_terrier	silky_terrier	Unspecified	82	7	378	355	[0.1 -0.0
3	n02097658_9222	n02097658-silky_terrier	silky_terrier	Unspecified	0	12	331	498	[0.2 0.
4	n02097658_422	n02097658-silky_terrier	silky_terrier	Unspecified	146	10	356	332	[0.0 -0.1

Task 1: Exact Search using Flat Index

FAISS implements several metrics and indices for similarity search.

Your task is to index all embeddings using `IndexFlatL2` index first.

```
faiss.IndexFlatL2(d)
...
```

You can read more about their implementation and how to use IndexFlatL2 on their GitHub page: https://github.com/facebookresearch/faiss/wiki

In [5]:
```
def create_IndexFlatL2(embeddings):
    # Answer:
    # faiss.IndexFlatL2(dimension)
    index = faiss.IndexFlatL2(embeddings.shape[1])
    return index


start_time = time.time()
index_flat = create_IndexFlatL2(embeddings)
end_time = time.time()
print(f"Time to index {np.round(end_time-start_time,3)}s")
```

```
Time to index 0.001s
```

Task 2: Generate embeddings for queries using OpenAI's CLIP

You are given a set of images in the `queries` subfolder. Your task is to:

- Generate CLIP-embedddings for each image
- Search for the image embeddings using the FAISS index

CLIP [1] an embedding model by OpenAI, which is used to extract a high-dimensional vector representation of an image, which captures its semantic and perceptual features.

[1] https://github.com/openai/CLIP

Initialize OpenAi's CLIP Encoder for feature extraction

In [6]:
```
device = "cuda" if torch.cuda.is_available() else "cpu"
print("running on: " + device)
model, preprocess = clip.load("ViT-B/32", device=device)
```

```
running on: cuda
```

Now Generate Embeddings of Queries

Use CLIP to generate for all images in the queries folder the corresponding embedding.

`model.encode_image(image)`
See: https://github.com/openai/CLIP

Upon generation, make sure to convert all data to float32, which is required to FAISS.

In [22]:
```
def create_query_embeddings(queries_start_path):

    # Load all images from the queries subfolder
    df_data = pd.DataFrame()
    query_file_names = []
    for root, dirs, files in os.walk(queries_start_path):
        for file_name in files:
            if ".jpg" in file_name:
                query_file_names.append(f"{root}/{file_name}")
                #query_file_names.append(file_name)


    # Answer:
    query_embeddings = np.zeros((0,embeddings.shape[1]), dtype=np.float32)
    for query_file_name in query_file_names:
        image = preprocess(Image.open(query_file_name)).unsqueeze(0).to(device)
        with torch.no_grad():
            image_features = model.encode_image(image)
        query_embeddings = np.vstack((query_embeddings, image_features.cpu().num
    return query_embeddings, query_file_names

query_embeddings, query_file_names = create_query_embeddings(queries_start_path)
```

Querying: Search for top-3 relevant images

We need to define a search function below to first vectorize our query image, and then search for the vectors with the closest distance.

```
index_flat.search(query_embeddings, k)
```

In [8]:
```
k = 3

def indexed_search(index, query_embeddings, k):
    # Search for the top-3 nearest neighbors

    # Answer:
    if not index.is_trained:
        print("Training index...")
        index.train(embeddings)

    if index.ntotal == 0:
        print("Adding embeddings to index...")
        index.add(embeddings)

    distances, indices = index.search(query_embeddings, k)

    return indices


start_time = time.time()
flat_indices = indexed_search(index_flat, query_embeddings, k)
end_time = time.time()
print(f"Time to index {np.round(end_time-start_time,3)}s")
```

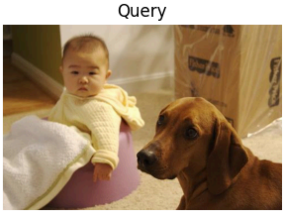
```
Adding embeddings to index...
Time to index 0.021s
```

Plot results

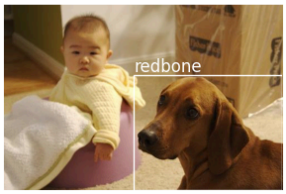
We use the helper function, to illustrate the results

In [9]:
```
plot_results(query_file_names, flat_indices, k)
```

Query



1-NN: redbone



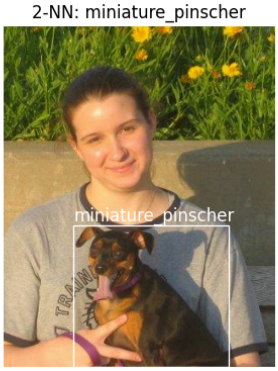
2-NN: redbone

3-NN: redbone

Query

1-NN: borzoi

2-NN: miniature_pinscher



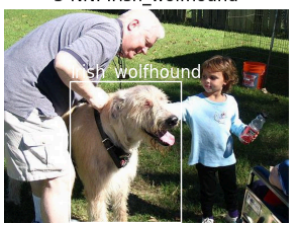
3-NN: affenpinscher

Query

1-NN: borzoi

2-NN: curly-coated_retriever

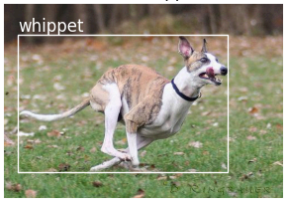
3-NN: Irish_wolfhound



Query

1-NN: Ibizan_hound

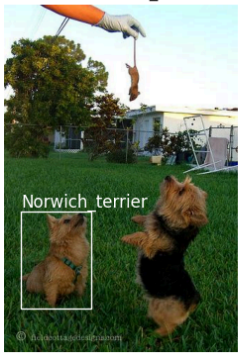
2-NN: whippet



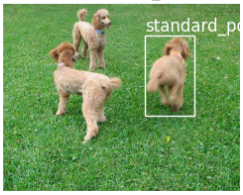
3-NN: Ibizan_hound

Query

1-NN: Norwich_terrier



2-NN: standard_poodle



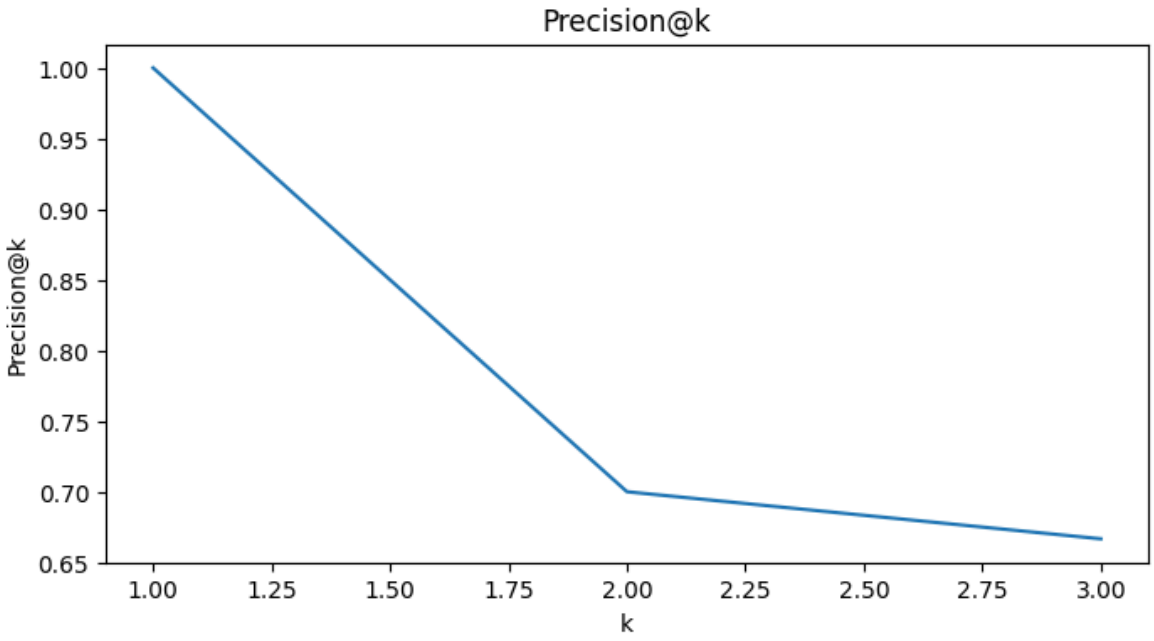
3-NN: briard

Measure precision@k

We measure the precision@k for different values of k.

In [10]:
```
# A simple test to test correctness
if flat_indices is None:
    print("You must implement Task 1 and 2 first")
else:
    assert precision_at_k(flat_indices, 1) == 1.0

    plot_precision_at_k(flat_indices, k)
```

```
precision@0: 1.0
precision@1: 0.7
precision@2: 0.6666666666666667
```



Task 3: Use Hierarchical Small World Graphs

To use Hierarchical Small World Graphs in FAISS, we need to work with the HNSW (Hierarchical Navigable Small World) index that FAISS provides. HNSW is a type of small-world graph used for approximate nearest neighbor search, which is highly efficient for high-dimensional data.

```
index = faiss.IndexHNSWFlat(d, M)
hswg_index.hnsw.efConstruction = ...
hswg_index.hnsw.efSearch = ...
```

Tuning the HNSW Index:

- `M` : A higher value will give more accurate results but with increased memory usage and slower indexing times.
- `EfConstruction` : This parameter controls the number of candidate neighbors that are evaluated during the construction of the graph. A larger value can lead to better quality of the graph at the expense of longer construction time.
- `EfSearch` : This parameter controls the number of candidate neighbors considered during search. A larger value can improve accuracy but makes the search slower.

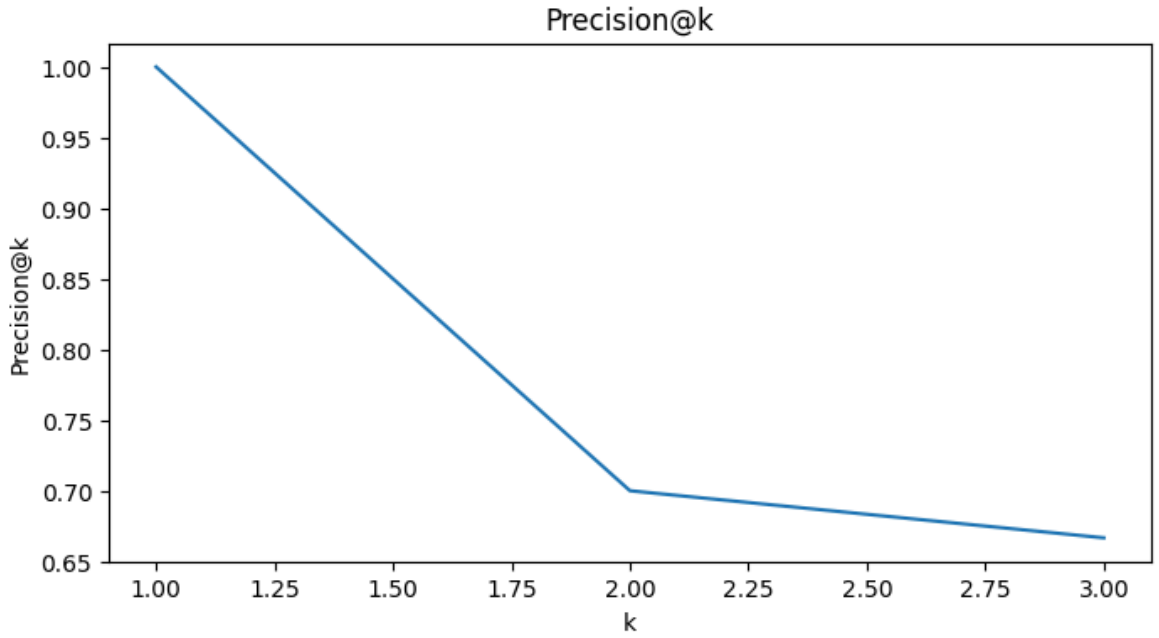
In [11]:
```
k=3

def index_and_query_hswg(embeddings, query_embeddings, k):

    # Answer:
    dim = embeddings.shape[1]

    hswg_index = faiss.IndexHNSWFlat(dim, 32) # 32 is the number of neighbors to
    hswg_index.hnsw.efConstruction = 40 # 40 is the number of k-NN, when the gra
    hswg_index.hnsw.efSearch = 64 # 64 is the number of candidates to explore du

    hswg_index.add(embeddings)
```

```
    distances, indices = hswg_index.search(query_embeddings, k)   # used to stor

    return hswg_index, indices


start_time = time.time()
hswg_index, hswg_indices = index_and_query_hswg(embeddings, query_embeddings, k)
end_time = time.time()
print(f"Time to index and query {np.round(end_time-start_time,3)}s")
```

```
Time to index and query 0.509s
```

In [12]:
```
# A simple test to test correctness
if hswg_indices is None:
    print("You must implement Task 3 first")
else:
    assert precision_at_k(hswg_indices, 1) == 1.0
    plot_precision_at_k(hswg_indices, k)
```

```
precision@0: 1.0
precision@1: 0.7
precision@2: 0.6666666666666667
```



Task 4: Optimize Hyper-Paramerers

Test different ranges of the three input parameters

```
M_values = [8, 16, 32]
efConstruction_values = [40, 100, 200]
efSearch_values = [10, 50, 100]
metric_types = [faiss.METRIC_L2, faiss.METRIC_L1]
```

For each parameter, create a `IndexHNSWFlat`, and search for the 5-NN.

For each set of parameters, record the query-time and the precision@5. Output the best 5 sets of parameters for the best quer-times and the best precision@5.

You may measure runtimes using:

```
start_time = time.time()
             ...
             end_time = time.time()
```

In [13]:

```
# Range of parameters to try for tuning
M_values = [8, 16, 32]
efConstruction_values = [40, 100, 200]
efSearch_values = [10, 50, 100]
metric_types = [faiss.METRIC_L2, faiss.METRIC_L1]
k = 5  # Number of neighbors


def hswg_grid_search(embeddings, query_embeddings, k):
    # Answer:
    # For each set of parameters, record (a) the query-time and (b) the precisio
    # Output the best 5 sets of parameters for the best quer-times and the best
    results = []
    for M in M_values:
        for efConstruction in efConstruction_values:
            for efSearch in efSearch_values:
                for metric in metric_types:
                    hswg_index = faiss.IndexHNSWFlat(embeddings.shape[1], M, met
                    hswg_index.hnsw.efConstruction = efConstruction
                    hswg_index.hnsw.efSearch = efSearch
                    hswg_index.reset()

                    if not hswg_index.is_trained:
                        hswg_index.train(embeddings)
                    if hswg_index.ntotal == 0:
                        print("Adding embeddings to index...")
                        hswg_index.add(embeddings)

                    start_time = time.time()
                    distances, hswg_indices = hswg_index.search(query_embeddings
                    end_time = time.time()
                    query_time = end_time - start_time
                    precision = precision_at_k(hswg_indices, k)

                    results.append((M, efConstruction, efSearch, metric, query_t

                    print(f"M={M}, efConstruction={efConstruction}, efSearch={ef

    # Best performance tracking
    best_params_query_time, best_params_precision_at_5 = [], []
    best_params_query_time = sorted(results, key=lambda x: x[4])[:5]
    best_params_precision_at_5 = sorted(results, key=lambda x: x[5], reverse=Tru

    return best_params_query_time, best_params_precision_at_5


best_params_query_time, best_params_precision_at_5 = hswg_grid_search(embeddings
# print ("Best Query Time", best_params_query_time)
# Best Query Time Results
print("\n=== Best Query Time Results ===")
for i, result in enumerate(best_params_query_time, 1):
    print(f"{i}. Time: {result[4]:.4f}s, M={result[0]}, efC={result[1]}, efS={re

# Best Precision Results
# print("\n=== Best Precision Results ===")
```

```
for i, result in enumerate(best_params_precision_at_5, 1):
    print(f"{i}. Precision: {result[5]:.4f}, M={result[0]}, efC={result[1]}, efS
print ("Best Precision@5", best_params_precision_at_5)
```

```
Adding embeddings to index...
M=8, efConstruction=40, efSearch=10, metric=1: query_time=0.0, precision@5=0.4800
000000000001
Adding embeddings to index...
M=8, efConstruction=40, efSearch=10, metric=2: query_time=0.0, precision@5=0.4600
000000000001
Adding embeddings to index...
M=8, efConstruction=40, efSearch=50, metric=1: query_time=0.0, precision@5=0.5000
000000000001
Adding embeddings to index...
M=8, efConstruction=40, efSearch=50, metric=2: query_time=0.0010035037994384766,
precision@5=0.5200000000000001
Adding embeddings to index...
M=8, efConstruction=40, efSearch=100, metric=1: query_time=0.0005042552947998047,
precision@5=0.5200000000000001
Adding embeddings to index...
M=8, efConstruction=40, efSearch=100, metric=2: query_time=0.0010046958923339844,
precision@5=0.5200000000000001
Adding embeddings to index...
M=8, efConstruction=100, efSearch=10, metric=1: query_time=0.0, precision@5=0.500
0000000000001
Adding embeddings to index...
M=8, efConstruction=100, efSearch=10, metric=2: query_time=0.0, precision@5=0.480
0000000000001
Adding embeddings to index...
M=8, efConstruction=100, efSearch=50, metric=1: query_time=0.0, precision@5=0.520
0000000000001
Adding embeddings to index...
M=8, efConstruction=100, efSearch=50, metric=2: query_time=0.0, precision@5=0.520
0000000000001
Adding embeddings to index...
M=8, efConstruction=100, efSearch=100, metric=1: query_time=0.0, precision@5=0.52
00000000000001
Adding embeddings to index...
M=8, efConstruction=100, efSearch=100, metric=2: query_time=0.0, precision@5=0.52
00000000000001
Adding embeddings to index...
M=8, efConstruction=200, efSearch=10, metric=1: query_time=0.0005049705505371094,
precision@5=0.5200000000000001
Adding embeddings to index...
M=8, efConstruction=200, efSearch=10, metric=2: query_time=0.0, precision@5=0.480
0000000000001
Adding embeddings to index...
M=8, efConstruction=200, efSearch=50, metric=1: query_time=0.0, precision@5=0.520
0000000000001
Adding embeddings to index...
M=8, efConstruction=200, efSearch=50, metric=2: query_time=0.0010018348693847656,
precision@5=0.5200000000000001
Adding embeddings to index...
M=8, efConstruction=200, efSearch=100, metric=1: query_time=0.001002311706542968
8, precision@5=0.5200000000000001
Adding embeddings to index...
M=8, efConstruction=200, efSearch=100, metric=2: query_time=0.001002311706542968
8, precision@5=0.5200000000000001
Adding embeddings to index...
M=16, efConstruction=40, efSearch=10, metric=1: query_time=0.0, precision@5=0.500
0000000000001
Adding embeddings to index...
M=16, efConstruction=40, efSearch=10, metric=2: query_time=0.0005135536193847656,
precision@5=0.5200000000000001
```

```
Adding embeddings to index...
M=16, efConstruction=40, efSearch=50, metric=1: query_time=0.0010073184967041016,
precision@5=0.5200000000000001
Adding embeddings to index...
M=16, efConstruction=40, efSearch=50, metric=2: query_time=0.0, precision@5=0.520
0000000000001
Adding embeddings to index...
M=16, efConstruction=40, efSearch=100, metric=1: query_time=0.001000404357910156
2, precision@5=0.5200000000000001
Adding embeddings to index...
M=16, efConstruction=40, efSearch=100, metric=2: query_time=0.0005035400390625, p
recision@5=0.5200000000000001
Adding embeddings to index...
M=16, efConstruction=100, efSearch=10, metric=1: query_time=0.0, precision@5=0.52
00000000000001
Adding embeddings to index...
M=16, efConstruction=100, efSearch=10, metric=2: query_time=0.001003503799438476
6, precision@5=0.5200000000000001
Adding embeddings to index...
M=16, efConstruction=100, efSearch=50, metric=1: query_time=0.001000881195068359
4, precision@5=0.5200000000000001
Adding embeddings to index...
M=16, efConstruction=100, efSearch=50, metric=2: query_time=0.0, precision@5=0.52
00000000000001
Adding embeddings to index...
M=16, efConstruction=100, efSearch=100, metric=1: query_time=0.001001358032226562
5, precision@5=0.5200000000000001
Adding embeddings to index...
M=16, efConstruction=100, efSearch=100, metric=2: query_time=0.001000165939331054
7, precision@5=0.5200000000000001
Adding embeddings to index...
M=16, efConstruction=200, efSearch=10, metric=1: query_time=0.0, precision@5=0.52
00000000000001
Adding embeddings to index...
M=16, efConstruction=200, efSearch=10, metric=2: query_time=0.0, precision@5=0.50
00000000000001
Adding embeddings to index...
M=16, efConstruction=200, efSearch=50, metric=1: query_time=0.001004219055175781
2, precision@5=0.5200000000000001
Adding embeddings to index...
M=16, efConstruction=200, efSearch=50, metric=2: query_time=0.000504970550537109
4, precision@5=0.5200000000000001
Adding embeddings to index...
M=16, efConstruction=200, efSearch=100, metric=1: query_time=0.001000165939331054
7, precision@5=0.5200000000000001
Adding embeddings to index...
M=16, efConstruction=200, efSearch=100, metric=2: query_time=0.001000881195068359
4, precision@5=0.5200000000000001
Adding embeddings to index...
M=32, efConstruction=40, efSearch=10, metric=1: query_time=0.0, precision@5=0.520
0000000000001
Adding embeddings to index...
M=32, efConstruction=40, efSearch=10, metric=2: query_time=0.0, precision@5=0.520
0000000000001
Adding embeddings to index...
M=32, efConstruction=40, efSearch=50, metric=1: query_time=0.0, precision@5=0.520
0000000000001
Adding embeddings to index...
M=32, efConstruction=40, efSearch=50, metric=2: query_time=0.0010042190551757812,
precision@5=0.5200000000000001
```

```
Adding embeddings to index...
M=32, efConstruction=40, efSearch=100, metric=1: query_time=0.001003742218017578
1, precision@5=0.5200000000000001
Adding embeddings to index...
M=32, efConstruction=40, efSearch=100, metric=2: query_time=0.001004695892333984
4, precision@5=0.5200000000000001
Adding embeddings to index...
M=32, efConstruction=100, efSearch=10, metric=1: query_time=0.0, precision@5=0.52
00000000000001
Adding embeddings to index...
M=32, efConstruction=100, efSearch=10, metric=2: query_time=0.0, precision@5=0.52
00000000000001
Adding embeddings to index...
M=32, efConstruction=100, efSearch=50, metric=1: query_time=0.000507116317749023
4, precision@5=0.5200000000000001
Adding embeddings to index...
M=32, efConstruction=100, efSearch=50, metric=2: query_time=0.005021810531616211,
precision@5=0.5200000000000001
Adding embeddings to index...
M=32, efConstruction=100, efSearch=100, metric=1: query_time=0.001005411148071289
, precision@5=0.5200000000000001
Adding embeddings to index...
M=32, efConstruction=100, efSearch=100, metric=2: query_time=0.000504255294799804
7, precision@5=0.5200000000000001
Adding embeddings to index...
M=32, efConstruction=200, efSearch=10, metric=1: query_time=0.0, precision@5=0.52
00000000000001
Adding embeddings to index...
M=32, efConstruction=200, efSearch=10, metric=2: query_time=0.000998735427856445
3, precision@5=0.5200000000000001
Adding embeddings to index...
M=32, efConstruction=200, efSearch=50, metric=1: query_time=0.002018690109252929
7, precision@5=0.5200000000000001
Adding embeddings to index...
M=32, efConstruction=200, efSearch=50, metric=2: query_time=0.000506401062011718
8, precision@5=0.5200000000000001
Adding embeddings to index...
M=32, efConstruction=200, efSearch=100, metric=1: query_time=0.001005172729492187
5, precision@5=0.5200000000000001
Adding embeddings to index...
M=32, efConstruction=200, efSearch=100, metric=2: query_time=0.002007484436035156
2, precision@5=0.5200000000000001

=== Best Query Time Results ===
1. Time: 0.0000s, M=8, efC=40, efS=10, Metric=L2
2. Time: 0.0000s, M=8, efC=40, efS=10, Metric=L1
3. Time: 0.0000s, M=8, efC=40, efS=50, Metric=L2
4. Time: 0.0000s, M=8, efC=100, efS=10, Metric=L2
5. Time: 0.0000s, M=8, efC=100, efS=10, Metric=L1
1. Precision: 0.5200, M=8, efC=40, efS=50, Metric=L1
2. Precision: 0.5200, M=8, efC=40, efS=100, Metric=L2
3. Precision: 0.5200, M=8, efC=40, efS=100, Metric=L1
4. Precision: 0.5200, M=8, efC=100, efS=50, Metric=L2
5. Precision: 0.5200, M=8, efC=100, efS=50, Metric=L1
Best Precision@5 [(8, 40, 50, 2, 0.0010035037994384766, 0.5200000000000001), (8,
40, 100, 1, 0.0005042552947998047, 0.5200000000000001), (8, 40, 100, 2, 0.0010046
958923339844, 0.5200000000000001), (8, 100, 50, 1, 0.0, 0.5200000000000001), (8,
100, 50, 2, 0.0, 0.5200000000000001)]
```

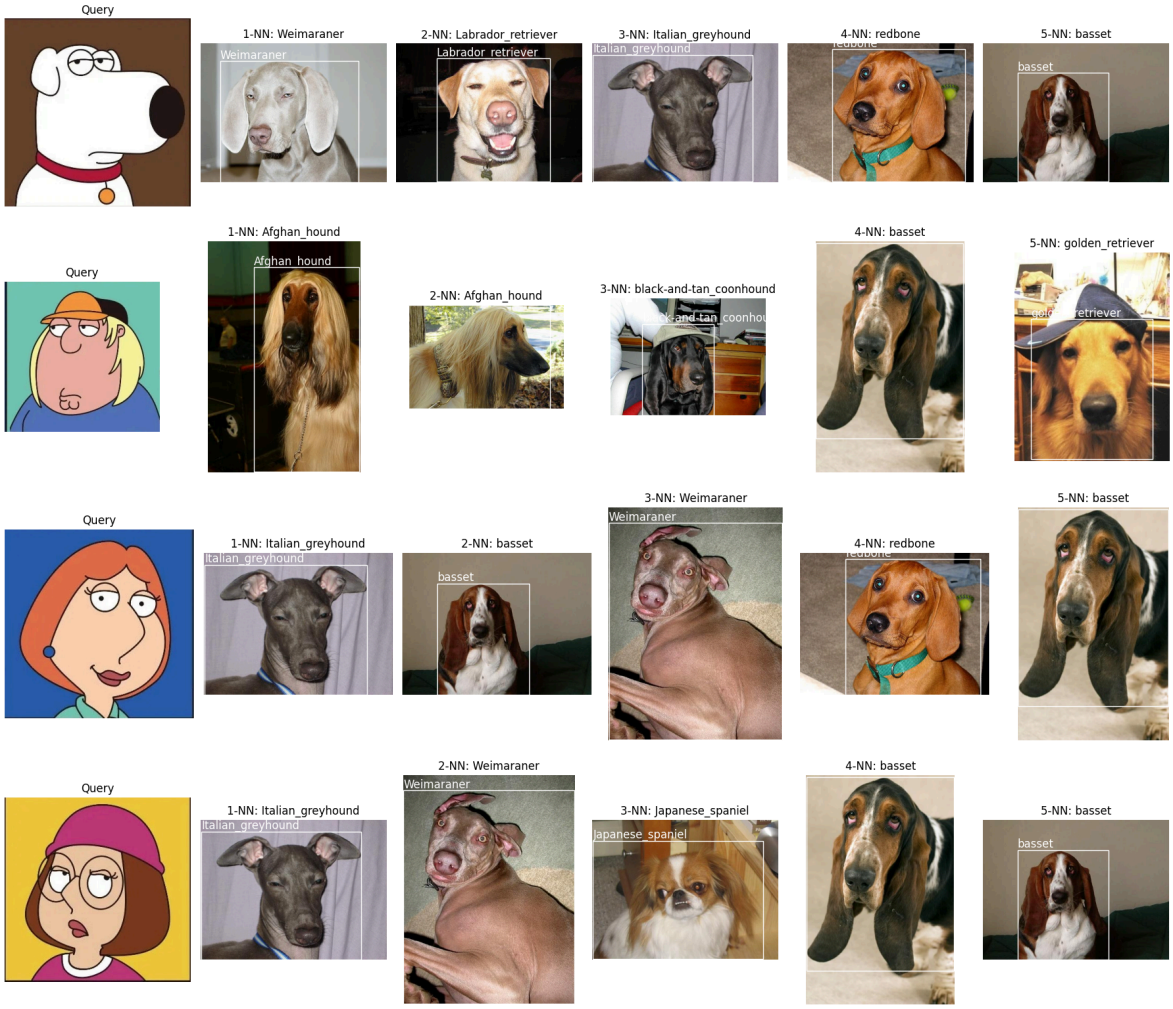
Task 5: Search for Dogs that look like you.

Use an image of yourself, and find dogs that look like you.

Alternatively: Search the internet and find at least 5 images of people, who look similar to dogs. For each image,

- Transform the image using CLIP
- Perform a search using any faiss index and plot the results.

In [24]:
```
def dogs_that_look_like_you(embeddings, k):
    # Answer:
    query_path ="queries_task5"
    query_embeddings, query_file_names = create_query_embeddings(query_path)
    hswg_index, hswg_indices = index_and_query_hswg(embeddings, query_embeddings
    plot_results(query_file_names, hswg_indices, k)

    return None

dogs_that_look_like_you(embeddings, k)
```



Task 6: Text-To-Image Retrieval

We will now use CLIP text embeddings to find the top-3 matching images.

You are given a set of sentences. Your task is to:

- Generate CLIP-embedddings for each sentence
- Search for the Top-3 images using `IndexFlatL2` index and Metric `METRIC_L1`

Please refer to [1] for additional information on how to generate text-embeddings.

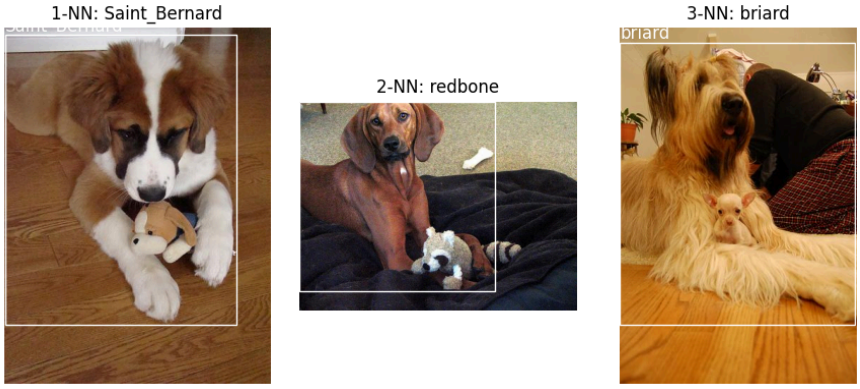
[1] https://github.com/openai/CLIP

You may add additional sentences to do further experiments.

In [30]:
```
k = 3  # Number of neighbors
sentences = ["a lion",
             "a dog and a girl",
             "a dog and a man",
             "a toy and a dog",
             "a group of people"]


def text_to_image(embeddings, sentences, k):
    metric = faiss.METRIC_L1

    # Answer:
    index_flat = None
    indices = None   # used to store the result

    # CLIP
    text_tokens = clip.tokenize(sentences).to(device)
    with torch.no_grad():
        text_embeddings = model.encode_text(text_tokens)
    text_embeddings = text_embeddings.cpu().numpy().astype('float32')

    index_flat = faiss.IndexFlatL2(embeddings.shape[1])
    # IndexFlatL2 is not suitable for L1 distance
    # Pre process the embeddings to simulate L1 distance
```

```
    # Both embeddings
    processed_embeddings = np.sign(embeddings) * np.sqrt(np.abs(embeddings))
    processed_text_embeddings = np.sign(text_embeddings) * np.sqrt(np.abs(text_e

    # Use processed embeddings
    index_flat.add(processed_embeddings)
    # Use provessed embeddings
    distances, indices = index_flat.search(processed_text_embeddings, k)

    return indices


indices = text_to_image(embeddings, sentences, k)

if flat_indices is None:
    print("You must implement Task 6")
else:
    assert precision_at_k(flat_indices, 1) == 1.0
    plot_results(sentences, indices, k)
```

a toy and a dog



a group of people

Hand in your solution via Moodle

Submit two files:

- This notebook
- A html-export of this notebook