Lab4 LSH

Wu Han, Class: SEIEE 2301

December 24, 2024

1 Experiment Overview

This experiment introduces the concept of Locality Sensitive Hashing (LSH). The basic idea of LSH is to first classify images based on their hash values, and then determine the final similarity by comparing feature vectors. This approach is much faster than directly comparing feature vectors one by one, significantly saving traversal time.

In this experiment, we need to complete such process: establish feature vectors and perform Hamming code analysis and hash value calculation.

2 Problem Restatement and Key Code Explanation

2.1 Creating Feature Vectors

According to the problem description, the feature vector consists of the color histograms of the four corners of the image: top-left, bottom-left, top-right, and bottom-right. We can use such code to achieve this, as shown below:

First we use this function to normalize a given vector to unit length.

```
def normalize_vector(v: List[float]) -> np.ndarray:
    """Normalize a vector to unit length."""
    return np.array(v) / np.linalg.norm(v)
```

It converts the input list `v` into a numpy array and then scales it by its Euclidean norm (magnitude), making it a unit vector.

Then we use this function to calculate the color histogram of an image by dividing the image into four quadrants and computing the sum of RGB values for each quadrant.

```
def compute_color_histogram(img: np.ndarray) -> np.ndarray:
    """Compute a color histogram for the image and return a feature
        vector."""
    # Split the image into 4 quadrants and calculate the sum of RGB
        values for each quadrant
    height, width, _ = img.shape
```

```
    quadrants = [img[:height // 2, :width // 2], img[:height // 2,
        width // 2:], img[height // 2:, :width // 2],
                 img[height // 2:, width // 2:]]

    # Sum the RGB values for each quadrant
    feature_vectors = [np.sum(q, axis=(0, 1)) for q in quadrants]

    # Normalize each quadrant's feature vector
    normalized_vectors = [normalize_vector(v) for v in
        feature_vectors]

    # Concatenate all normalized vectors into a single feature
        vector
    return np.concatenate(normalized_vectors)
```

The image is split into four quadrants (top-left, top-right, bottom-left, bottom-right), and the sum of RGB values in each quadrant is calculated. These values are normalized to form feature vectors, which are then concatenated to form a final feature vector for the image.

2.2 Getting Hamming Code

Although the PPT shows a method to obtain hash values without using Hamming code, Python's strong string manipulation capabilities allow us to directly use strings to get the Hamming code. The corresponding code is:

We use this function to normalize the histogram values into discrete values (0, 1, or 2) based on specific thresholds.

```
def normalize_histogram(v: np.ndarray) -> np.ndarray:
    """Normalize the histogram into discrete values (0, 1, 2) based
         on thresholds."""
    v = v.copy()
    for i in range(len(v)):
        if v[i] < 0.3:
            v[i] = 0
        elif v[i] < 0.6:
            v[i] = 1
        else:
            v[i] = 2
    return v
```

This function loops through each element of the histogram and assigns a value of 0, 1, or 2 depending on which threshold the value falls into (0.3, 0.6).

Then we use this function to generate a Hamming code from the normalized histogram.

```
def generate_hamming_code(v: np.ndarray) -> str:
    """Generate a Hamming code string from the normalized histogram
        ."""
    return ''.join(['00' if val == 0 else '10' if val == 1 else
        '11' for val in v])
```

It maps each value in the normalized histogram to a corresponding 2-bit string: '00' for 0, '10' for 1, and '11' for 2, and concatenates them to form the full Hamming code.

2.3 Using Locality Sensitive Hashing for Retrieval

Locality Sensitive Hashing involves directly taking certain bits from the Hamming code to obtain the hash function. The corresponding code is:

This function selects specific bits from the Hamming code based on a list of indices.

```
def select_bits(hamming_code: str, indices: List[int]) -> str:
    """Select bits from the Hamming code based on the specified
        indices."""
    return ''.join(hamming_code[i] for i in indices)
```

It selects the bits from the Hamming code according to the provided list of indices.

This function computes the cosine similarity between two feature vectors.

```
def compute_similarity(vec1: np.ndarray, vec2: np.ndarray) -> float
    :
    """Compute cosine similarity between two feature vectors."""
    return np.dot(vec1, vec2)
```

It computes the cosine similarity between two vectors by taking their dot product, assuming both vectors are normalized.

2.4 Find Similar Images

This function processes a dataset of images and finds the most similar images to the target image based on the cosine similarity of their feature vectors.

```
def find_similar_images(target_img: np.ndarray, dataset_dir: str,
    key_indices: List[int],
                        num_images: int = 40, top_n: int = 6) ->
                            List[Dict[str, float]]:
    """Find the most similar images from the dataset to the target
        image."""
    vectors, dataset = [], []

    # Process all images in the dataset
    for i in range(1, num_images + 1):
        img = cv2.imread(f"{dataset_dir}/{i}.jpg")
        color_hist = compute_color_histogram(img)
        normalized_hist = normalize_histogram(color_hist)
        hamming_code = generate_hamming_code(normalized_hist)
        selected_bits = select_bits(hamming_code, key_indices)
        dataset.append(selected_bits)
        vectors.append(color_hist)

    # Process the target image
    target_hist = compute_color_histogram(target_img)
    target_normalized_hist = normalize_histogram(target_hist)
```

```
    target_hamming_code = generate_hamming_code(
        target_normalized_hist)
    target_selected_bits = select_bits(target_hamming_code,
        key_indices)

    # Find the similarities between the target image and dataset
        images
    similarities = {}
    for i, vector in enumerate(vectors):
        if dataset[i] == target_selected_bits:
            similarity = compute_similarity(vector, target_hist)
            similarities[i] = similarity

    # Sort the similarities in descending order and pick the top N
    sorted_similarities = sorted(similarities.items(), key=lambda x
        : x[1], reverse=True)
    top_similar_images = sorted_similarities[:top_n]

    return top_similar_images
```

This function processes all the images in the dataset and calculates their feature vectors, Hamming codes, and selected bits.It first compare whether the Hamming code between the target and the object images are the same.If so,tt then compares the target image with each image in the dataset using cosine similarity and returns the top N most similar images.

3 Results Analysis and Comparison

3.1 Result Analysis

Here is the result of running the code above(LSE.py). We can see the most

```
C:\Users\WH\anaconda3\python.exe E:\MultiMedia\lab4-LSH-2\lab4-LSH\LSE.py
The most similar image is: 38 with similarity score 4.0000
The other top 5 similar images are:
Rank 2: Image 12 with similarity score 3.9952
Rank 3: Image 23 with similarity score 3.9727
Rank 4: Image 26 with similarity score 3.9692
Rank 5: Image 40 with similarity score 3.9667
Rank 6: Image 7 with similarity score 3.9656
Time taken: 0.0470 seconds
```

Figure 1: Result Of Running LSE.py

similar image is 38, and this is obviously right. And the second most similar image is image12, which is also a picture of a similar tree. Other images, such as 26 and 40, have a dominant yellow tone, which is consistent with the target. Image 7, on the other hand, is a tree and shares similar shapes with the target.

3.2 Comparison of Different Projection Sets

First, we use the full Hamming code analysis, i.e., `key = list(range(0,24))`, and obtain the following result:

```
The most similar image is: 38 with similarity score 4.0000
The other top 5 similar images are:
Rank 2: Image 12 with similarity score 3.9952
Time taken: 0.0450 seconds
```

We thus found that when using this encoding method, only two images and the target image were in the same set. Upon observation, these two images were both of similar types—large trees—which indicates that our hash matching is highly effective

By comparing their distributions, we find that the differences in the 3th, 7th, 11th, 19th bits are quite significant. Therefore, we change the key and get the following results:

```
The most similar image is: 38 with similarity score 4.0000
The other top 5 similar images are:
Rank 2: Image 12 with similarity score 3.9952
Rank 3: Image 26 with similarity score 3.9692
Rank 4: Image 40 with similarity score 3.9667
Time taken: 0.0470 seconds
```

We found that when using this method, only 4 images and the target image were in the same set. Upon observation, Among these four images, two are the same tree images as mentioned above, and the other two are images of leopards. It is worth noting that yellow is the dominant color in all these images.

Considering that our Hamming code is based on the BGR color order, and the target image has blue on top and yellow at the bottom, we try comparing the 2nd, 8th, 18th, and 23th bits, resulting in:

```
The most similar image is: 38 with similarity score 4.0000
The other top 5 similar images are:
Rank 2: Image 12 with similarity score 3.9952
Rank 3: Image 40 with similarity score 3.9667
Rank 4: Image 25 with similarity score 3.9629
Rank 5: Image 8 with similarity score 3.9625
Rank 6: Image 21 with similarity score 3.9590
Time taken: 0.0490 seconds
```

In this case, Images with more blue and yellow colors are included . It is clear that our feature vectors are somewhat crude, but they generally meet our needs.

3.3 Comparison with NN

We replace the method of LSH searching, instead, we use NN search.This is how we implement it. For each feature vector in the dataset:

1. Compute the Euclidean distance between the target image's feature vector and the dataset image's feature vector using:

$$\text{distance} = \sqrt{\sum_{i=1}^{n} (\text{vec1}_i - \text{vec2}_i)^2}$$

2. Store the image index and its corresponding distance as a tuple in the list `distances`.

Just like this:

```
def compute_euclidean_distance(vec1: np.ndarray, vec2: np.ndarray) -> float:
    """Compute the Euclidean distance between two feature vectors."""
    return np.linalg.norm(vec1 - vec2)


# Main function to process the dataset and find the most similar image using Nearest Neighbor
def find_nn_similar_images(target_img: np.ndarray, dataset_dir: str, num_images: int = 40, top_n: int = 6) -> List[int]:
    """Find the most similar images from the dataset to the target image using Nearest Neighbor (NN)."""
    vectors = []

    # Process all images in the dataset
    for i in range(1, num_images + 1):
        img = cv2.imread(f"{dataset_dir}/{i}.jpg")
        color_hist = compute_color_histogram(img)
        vectors.append(color_hist)

    # Process the target image
    target_hist = compute_color_histogram(target_img)

    # Calculate the Euclidean distances between the target image and all dataset images
    distances = []
    for i, vector in enumerate(vectors):
        distance = compute_euclidean_distance(vector, target_hist)
        distances.append((i, distance))

    # Sort the distances in ascending order and pick the top N closest images
    sorted_distances = sorted(distances, key=lambda x: x[1])

    # Extract the indices of the top N most similar images
    top_similar_images = sorted_distances[:top_n]

    return top_similar_images
```

The details are in NN.py. And the result of running NN.py is:

```
The most similar image is: 38 with similarity score (distance) 0.0000
The other top 5 similar images are:
Rank 2: Image 12 with similarity score (distance) 0.0978
Rank 3: Image 23 with similarity score (distance) 0.2338
Rank 4: Image 26 with similarity score (distance) 0.2482
Rank 5: Image 40 with similarity score (distance) 0.2582
Rank 6: Image 7 with similarity score (distance) 0.2623
Time taken: 0.0456 seconds
```

After comparison, we found that the runtime of the two processing methods is essentially the same. This is partly because our database is very small, and our device performance is excellent, making it difficult to demonstrate the advantages of the algorithm on a small database. On the other hand, we have optimized the implementation of the NN algorithm, achieving high efficiency. Regarding the output image results, both algorithms produce almost identical results. The NN algorithm also tends to output images with more blue and yellow colors.

4 Further Considerations

4.1 Use of Color Histogram Features

As mentioned earlier, the retrieval effect meets our expectations. Although the feature information is somewhat crude, it effectively filters out irrelevant images. Specifically, when exploring the full set, we can unexpectedly find quite good results.

The similarity of the retrieved images is primarily reflected in the color. Since our target image has cool tones, the retrieved images are mostly blue and yellow, reflecting the main color scheme.

4.2 Can Other Features Be Designed?

Shape features describe the geometric properties of objects within an image, capturing information about their contours, boundaries, or regions. These features are particularly useful for tasks such as object recognition, image classification, and retrieval, especially when the shape plays a key role in defining the object.We can use methods like edge detector to implement it.

4.3 The Explanation Of the Content on the 10th Page of the PPT

On the 10th page of the ppt, we show another approach to implement the hashing method.The X1, X2, X3... just refers to the number in the projection sets.For example, there is a projection set [1, 3, 5, 9], then X1 refers to 1, X2 refers to 3...

4.4 Reflection on the Experiment

This experiment significantly deepened my understanding of the role of mathematics and algorithms in solving real-world problems. During the feature extraction process, I realized the importance of fundamental mathematical tools, such as vector normalization and distance computation. These seemingly basic concepts are crucial for the proper functioning of algorithms. By implementing Locality Sensitive Hashing (LSH) and Nearest Neighbor (NN) algorithms, I also gained insights into algorithm optimization, particularly in selecting feature dimensions or indexing strategies to enhance efficiency.