ICE2607 Lab 4: LSH

SJTU-SEIEE cny123222

December 19, 2024

1 Experiment Overview (实验概览)

Locality Sensitive Hashing (LSH) is a technique that facilitates efficient Approximate Nearest Neighbor Search, offering a faster alternative to traditional methods like Nearest Neighbor (NN) or K-Nearest Neighbor (KNN) search. **By grouping data into buckets based on hash functions, LSH significantly accelerates search processes by narrowing down the search space.**

This experiment aims to compare the efficiency of LSH with NN search methods. Using a dataset of 40 images, the goal is to find the most similar image to a target image and analyze the time disparity between LSH and NN searches. The influence of different projection sets on initialization and search speed is also investigated. Furthermore, the experiment explores the use of ResNet for extracting image features to enhance retrieval performance.

2 Solution Approach (解决思路)

2.1 Step 1: Feature Vector Generation

Firstly, image features are extracted using color histograms. Each image is evenly divided into four parts, and the RGB proportions of these quadrants are concatenated to create a 12-dimensional vector. Through an encoding step, a feature vector consisting of 12 components with values of 0, 1, or 2 is generated for each image. The process, including the encoding rule, is illustrated in the following pseudocode:

```
def generate_feature_vector(img): # img.shape = (H, W, 3)
    H, W, C = img.shape
    half_H, half_W = H // 2, W // 2
    # Calculate RGB proportions
    rgbs = []
    for i in range(2):
        for j in range(2):
            subimage = img[i * half_H: (i + 1) * half_H, j * half_W: (j + 1) * half_W] # subimage.shape = (H//2, W//2, 3)
            rgb = np.sum(subimage, axis=(0, 1)) # rgb.shape = (3,)
            rgbs.extend(rgb / np.sum(rgb))
    rgbs = np.array(rgbs) # rgbs.shape = (12,)
```

```
    # Encoding
    low_th = min(rgbs) + (max(rgbs) - min(rgbs)) / 3 # Lower threshold
    high_th = max(rgbs) - (max(rgbs) - min(rgbs)) / 3 # Higher threshold
    feature_vec = np.ones(12, dtype=np.uint8)
    feature_vec[rgbs >= high_th] = 2 # Assign 2
    feature_vec[rgbs <= low_th] = 0 # Assign 0
    return feature_vec
```

2.2 Step 2: Hash Function Computation

Next, the 12-dimensional feature vector is mapped to a Hamming space of d' dimensions and then projected onto a projection set, dividing the N images into n groups, where $n \ll N$. Since the details of the hash function are intricate, its descriptions are simplified here. However, the implementation, as shown in the following pseudocode, can be concise.

```
def hash_proj(feature_vec, proj_set):
    aux1 = (proj_set - 1) // 2
    aux2 = (proj_set - 1) % 2 + 1
    hash_val = (aux2 <= feature_vec[aux1]).astype(np.uint8)
    return tuple(hash_val) # hash_val will be used as the key in a dict
```

2.3 Step 3: LSH Retrieval

After the first two steps, the dataset images are classified into n categories. When given the target image, its hash value is computed and the image is classified. Then, we only need to check the dataset images in the same category. To find the most similar image, we compute the minimum distance between the feature vectors. The process can be illustrated using the following pseudocode:

```
def LSH_retrieval(hash_dict, target_img, proj_set): # hash_dict maps hash values to a list of image paths
    # Computate hash value of target image
    target_vec = generate_feature_vector(img)
    hash_val = hash_proj(target_vec, proj_set)
    # Find the nearest dataset image
    candidates = hash_dict[hash_val] # Assume hash_val in hash_dict.keys()
    result = find_nearest(target_vec, candidates)
    return result
```

3 Experimental Results (实验结果)

3.1 Experimental Environment

The experiments were conducted on macOS Sonoma 14.6.1 with Python 3.13.0. Key libraries included OpenCV-Python 4.10.0.84, Numpy 2.1.3 and Matplotlib 3.9.2.

3.2 Retrieval Results

Figure 1 illustrates the retrieval results of the LSH and NN algorithms. Both methods consider **38.jpg** as the most similar image, which is identical to the target image. However, LSH demonstrates **a speedup of approximately 12 times** compared to NN, aligning with our expectations. This improvement is attributed to LSH effectively narrowing down the search space, and thus reducing the search time required.

```
LSH:
Projection Set:[1, 8, 16, 24]
Most similar image: dataset/38.jpg
Time taken: 0.002923727035522461s

NN:
Most similar image: dataset/38.jpg
Time taken: 0.035111188888549805s

Speed up: 12.01x
```

Figure 1 Retrieval Results of LSH and NN with Speed Comparison

4 Analysis and Discussion (分析与思考)

4.1 Influence of Projection Sets

In the previous experiment, the projection set was manually set as [1, 8, 16, 24]. To enhance the efficiency of the LSH method, we delve into the impact of varying projection sets on retrieval speed. The dimensions of the projection set range from 1 to 24, and we scatter the numbers in the projection sets so that they contain more information. For example, with a dimension of 3, the projection set is set as [1, 12, 24].

4.1.1 Influence on Search Time

Figure 2 illustrates how different projection sets influence the speedup of LSH over NN. From the figure, it is evident that as **the dimension of the projection sets m** increases, the retrieval speed initially experiences a sharp rise and then stabilizes. This pattern is straightforward to comprehend.

- When $m = 1$, the dataset is divided into two categories, halving the number of images for comparison and resulting in a 2x speedup.
- As m increases, the number of categories grows exponentially, reducing the number of candidate images exponentially, thereby sharply increasing the retrieval speed.
- When $m = 5$, given that $\log_2 40 \approx 5$, there may be only one image in each category, necessitating comparison with just one image against the target image.

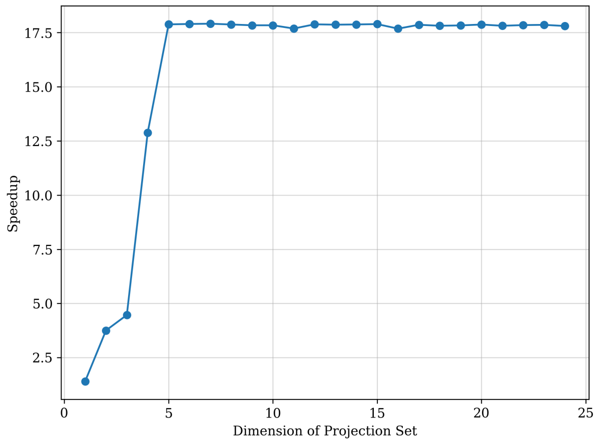

Figure 2 Impact of Projection Sets on Retrieval Speed

- With further increments in m, where there remains only one image in each category, the situation remains unchanged. LSH reaches the retrieval speed limit at this point.

4.1.2 Influence on Initialization Time

Does a larger value of m always lead to better retrieval performance? Figure 3 illustrates the impact of m on both the initialization time and search time of LSH. When search time decreases with increasing m, as we discussed earlier, the initialization time for LSH, which involves computing hash values and classifying dataset images, increases linearly. The exact reason for this is not clear and could be attributed to the additional time required for computing hash values and inserting them into the hash table.

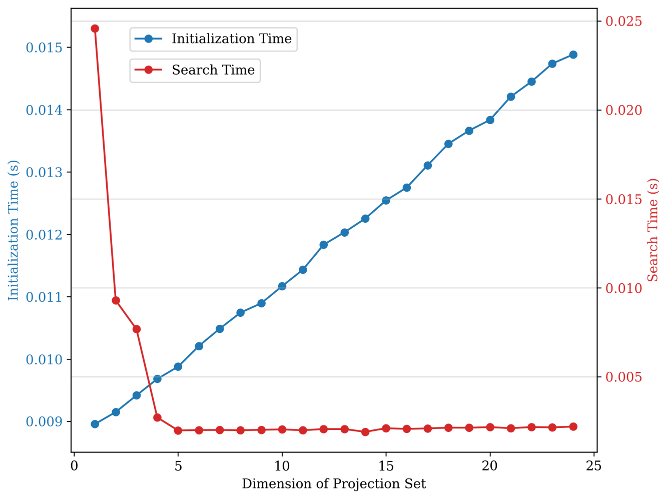

Figure 3 Impact of Projection Sets on Initialization and Search Time

4.1.3 Suggestion for Choosing Projection Sets

From the observations, it is evident that as the dimension of the projection sets increases, the initialization time gradually increases, while the search time first sharply decreases before stabilizing. Therefore, selecting an appropriate projection set is crucial for the efficiency of the LSH algorithm.

Based on our experimental results, we recommend **opting for a uniformly distributed projection set with a dimension of $\lfloor \log_2 N \rfloor - 1$**, where N represents the number of images in the dataset. This choice strikes a balance between shorter initialization time and shorter search time. Therefore, in this specific case, we selected the projection set [1, 8, 16, 24].

4.2 Feature Analysis and ResNet Feature Extraction

4.2.1 Feature Analysis

In the previous experiment, color histograms were employed to extract image features. Figure 4 displays the cosine similarity between image feature vectors, providing valuable insights into the extracted features.

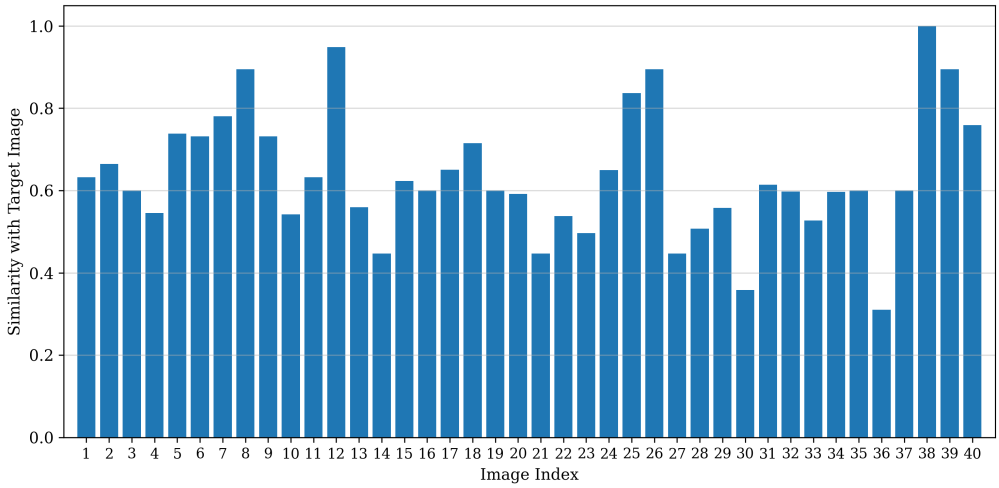

Figure 4 Similarity between Dataset Images and Target Image Using Color Histogram Features

Analyzing Figure 4, it is clear that, besides 38.jpg, images such as 12.jpg, 8.jpg, 26.jpg, 39.jpg, and 25.jpg exhibit a strong similarity with the target image based on the feature vectors derived from color histograms. Referring to Figure 5, it can be noticed that these images share a similar hue of yellow, aligning with our expectations regarding the features extracted by color histograms.

The second most similar image (12.jpg) indeed bears a striking resemblance to the target image, validating the efficacy of the algorithm. Images 26.jpg, 39.jpg, and 25.jpg also present a similar desert background, further confirming the effectiveness of retrieval, particularly concerning similar colors.

However, image 8.jpg, despite having yellow as the predominant color, does not share significant visual similarity with the target image. Consequently, we proceed to explore alternative feature extraction methods.

4.2.2 ResNet Feature Extraction

Considering the effectiveness of deep learning models in extracting high-level features, **a pretrained ResNet-18 model** is utilized to extract features from the images. To be consistent with the previous experiment, we employ the first 12 dimensions of the output of the network. Figure 6 illustrates the similarity between the image feature vectors generated by ResNet.

Observing Figure 6, it can be noted that the second most similar image is also 12.jpg, consistent with

38.jpg (Target Image)

12.jpg

8.jpg

26.jpg

39.jpg

25.jpg

2.jpg

Figure 5 Several Images in the Dataset

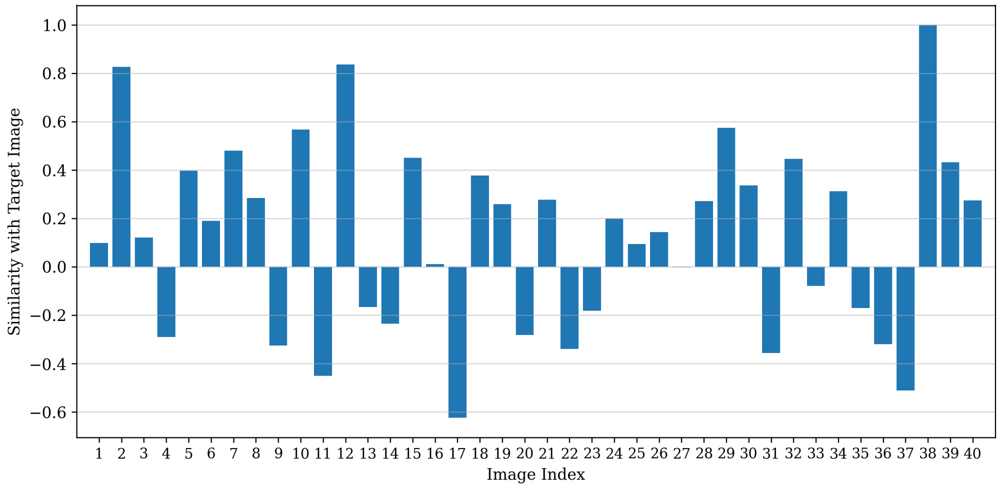

Figure 6　Similarity between Dataset Images and Target Image Using ResNet Features

the results obtained using color histograms. However, the third most similar image is 2.jpg, which features a plant similar to the one in the target image. This highlights the effectiveness of ResNet in extracting high-level features, surpassing the color histograms that primarily focus on color distribution. Furthermore, the feature vectors provided by ResNet exhibit, on average, lower similarity levels than those given by color histograms, which aids in distinguishing between different images.

4.3 Highlights of Codes

- **Multithreaded Image Loading:** Employing multithreading for image loading to expedite LSH initialization, mitigating potential I/O bottlenecks.
- **Integration of tqdm Progress Bar:** Incorporating a tqdm progress bar to display real-time progress updates, improving user experience and task monitoring.
- **Object-Oriented Approach:** Utilizing an object-oriented design to enhance code readability and maintainability.
- **Command-Line Interface:** Implementation of a command-line interface allowing for the input of various parameters, enhancing versatility, usability, and cross-system compatibility.

5 Reflection and Conclusion (实验感想)

5.1 Learning from Experiments

Through the course of these experiments, we have deepened our comprehension of LSH and image retrieval, refined our skills in parameter selection, and further honed our proficiency in LaTeX. These experiences have significantly boosted our research capabilities, proving to be invaluable for those embarking on scientific exploration.

5.2 Challenges Encountered

Two main challenges were encountered during the experiments.

- Firstly, the encoding rules provided by the slides mainly resulted in binary encoding (0 and 1) without the possibility of obtaining a value of 2. This limitation led to poor encoding effectiveness, necessitating the exploration of alternative encoding methods for experimentation.
- Secondly, measuring the algorithm's runtime posed difficulties as it required multiple measurements to calculate averages. Each measurement was susceptible to interference from other processes running on the computer, causing abrupt fluctuations in the results. Additionally, this process was time-consuming.

5.3 Conclusion

In this experiment, we employed the LSH algorithm to retrieve the target image from the dataset. We investigated the impact of projection sets on retrieval speed and proposed guidelines for selecting suitable projection sets. Furthermore, we analyzed features extracted by color histograms and enhanced them using ResNet.

A Source Code File List

Table 1 File List

File Name	Description
knn.py	Nearest Neighbor algorithm
lsh.py	Locality Sensitive Hashing algorithm
main.py	Image retrieval process and main function
plot.py	Functions for generating analytical plots
preprocess.py	Image preprocessing functions

B Source Code

```python
"""
File name: knn.py
"""
import numpy as np
from preprocess import Image


class KNN:
    """
    K-Nearest Neighbors (K = 1)
    """

    def __init__(
            self,
            resnet: bool = False,
            normalize: bool = True
    ):
        self.image_paths = []
        self.resnet = resnet
        self.normalize = normalize

    def add(
            self,
            img_path: str
    ):
        """
        Add a dataset image
        """
        self.image_paths.append(img_path)

```

```python
    def search(
            self,
            img_path: str
    ):
        """
        Search for the most similar image in the dataset
        """
        image = Image(img_path, self.resnet, self.normalize)

        min_dist = float('inf')
        min_img = None

        for img_path in self.image_paths:
            img, dist = self._calc_dist(img_path, image)
            if dist < min_dist:
                min_dist = dist
                min_img = img

        return min_img.path

    def _calc_dist(
            self,
            img_path: str,
            target_img: Image
    ) -> float:
        """
        Calculate the distance between the target image and the image in the dataset
        """
        img = Image(img_path, self.resnet, self.normalize)
        dist = np.linalg.norm(img.feature_vec - target_img.feature_vec)
        return img, dist
```

```python
"""
File name: lsh.py
"""
import numpy as np
from typing import Union
from preprocess import Image


class LSH:
    """
    Locality Sensitive Hashing
    """
```

```python

    def __init__(
            self,
            indicators: Union[list, np.ndarray], # Projection set
            resnet: bool = False,
            normalize: bool = True
    ):
        self.hash_tab = {} # Hash table to store the dataset
        self.indicators = np.array(indicators) if isinstance(indicators, list) else indicators
        self.resnet = resnet
        self.normalize = normalize

    def add(
            self,
            img_path: str
    ):
        """
        Add a dataset image to the hash table
        """
        img = Image(img_path, self.resnet, self.normalize)
        hash_val = self._projection(img.feature_vec, self.indicators)
        if hash_val not in self.hash_tab:
            self.hash_tab[hash_val] = [img_path]
        else:
            self.hash_tab[hash_val].append(img_path)

    def search(
            self,
            img_path: str
    ):
        """
        Search for the most similar image in the dataset
        """
        image = Image(img_path, self.resnet, self.normalize)
        hash_val = self._projection(image.feature_vec, self.indicators)
        if hash_val not in self.hash_tab:
            return None

        candidates = self.hash_tab[hash_val]
        min_dist = float('inf')
        min_img = None

        for img_path in candidates:
```

```
            img, dist = self._calc_dist(img_path, image)
            if dist < min_dist:
                min_dist = dist
                min_img = img

        return min_img.path

    def _projection(
            self,
            feature_vec: np.ndarray,
            indicators: np.ndarray
    ) -> np.ndarray:
        """
        Project the feature vector to the given projection set
        """
        proj_1 = (indicators - 1) // 2
        proj_2 = (indicators - 1) % 2 + 1
        hash_val = (proj_2 <= feature_vec[proj_1]).astype(np.uint8)
        return tuple(hash_val) # hash_val will be used as the key in a dict

    def _calc_dist(
            self,
            img_path: str,
            target_img: Image
    ) -> float:
        """
        Calculate the distance between the target image and the image in the dataset
        """
        img = Image(img_path, self.resnet, self.normalize)
        dist = np.linalg.norm(img.feature_vec - target_img.feature_vec)
        return img, dist
```

```
"""
File name: main.py
"""
import sys
import glob
import time
import argparse
from lsh import LSH
from knn import KNN


def parse_args(args):
```

```python
    """
    Parse command line arguments
    """
    parser = argparse.ArgumentParser()
    parser.add_argument("--image-dir", type=str, default="dataset", help="Path to folder containing dataset images")
    parser.add_argument("--type", type=str, choices=["LSH", "NN", "comp"], default="LSH", help="Choose the algorithm to use, comp for comparison")
    parser.add_argument("--indicator", type=list, default=[1, 8, 16, 24], help="Projection set for LSH")
    parser.add_argument("--resnet", type=bool, default=False, help="Use ResNet to generate feature vector")
    parser.add_argument("--target-dir", type=str, default="target.jpg", help="Path to the target image")

    args = parser.parse_args(args)
    return args


def get_image_paths(input_dir, extensions = ("jpg", "jpeg", "png", "bmp")):
    """
    Get image paths from the given directory
    """
    pattern = f"{input_dir}/**/*"
    img_paths = []
    for extension in extensions:
        img_paths.extend(glob.glob(f"{pattern}.{extension}", recursive=True))

    if not img_paths:
        raise FileNotFoundError(f"No images found in {input_dir}. Supported formats are: {', '.join(extensions)}")

    return img_paths


def main(args):
    args = parse_args(args)

    tasks = ["LSH", "NN"] if args.type == "comp" else [args.type]

    for type in tasks:
        # Initialize the searcher
        searcher = LSH(args.indicator, args.resnet) if type == "LSH" else KNN(args.resnet)

```

```python
        # Add images to the searcher
        img_paths = get_image_paths(args.image_dir)
        for img_path in img_paths:
            searcher.add(img_path)

        # Search for the most similar image
        start_time = time.time()
        result_path = searcher.search(args.target_dir)
        finish_time = time.time()

        # Record time taken for each algorithm
        if args.type == "comp":
            if type == "LSH":
                lsh_time = finish_time - start_time
            else:
                knn_time = finish_time - start_time

        print()
        print(f"{type}:")
        if type == "LSH":
            print(f"Projection Set:{args.indicator}")
        print(f"Most similar image: {result_path}")
        print(f"Time taken: {finish_time - start_time}s")

    # Compare the time taken for LSH and NN
    if args.type == "comp":
        speed_up = knn_time / lsh_time
        print()
        print(f"Speed up: {speed_up:.2f}x")


if __name__ == '__main__':
    main(sys.argv[1:])
```

```python
"""
File name: plot.py
"""
import sys
import time
import threading
from tqdm import tqdm
import numpy as np
from typing import Tuple
import matplotlib.pyplot as plt
```

```python
from lsh import LSH
from knn import KNN
from preprocess import Image

# Plot settings
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.size'] = 11
plt.rcParams['axes.labelsize'] = 11
plt.rcParams['xtick.labelsize'] = 11


def compute_knn_time(
        repeat_times: int = 100
):
    """
    Compute the average time for KNN search
    """
    knn = KNN(normalize=False) # Skip RGB normalization for better result
    for i in range(1, 41):
        img_path = f"dataset/{i}.jpg"
        knn.add(img_path)

    # Compute the time for KNN search
    knn_time = 0
    for i in range(repeat_times):
        start_time = time.time()
        knn.search("target.jpg")
        knn_time += (time.time() - start_time)
    knn_time /= repeat_times

    return knn_time


def compute_lsh_time(
        indicators: list,
        repeat_times: Tuple[int, int] = (1, 100),
        multithread: bool = True
):
    """
    Compute the average time for LSH search
    """
    # Compute the time for LSH initialization
    init_time = 0
    for i in range(repeat_times[0]):
```

```
        start_time = time.time()
        lsh = LSH(indicators, normalize=False) # Skip RGB normalization for better result
        if multithread: # Use multithreading for better performance
            threads = []
            for i in range(1, 41):
                img_path = f"dataset/{i}.jpg"
                thread = threading.Thread(target=lsh.add, args=(img_path,))
                threads.append(thread)
                thread.start()
            for thread in threads:
                thread.join()
        else:
            for i in range(1, 41):
                img_path = f"dataset/{i}.jpg"
                lsh.add(img_path)
        init_time += (time.time() - start_time)
    init_time /= repeat_times[0]

    # Compute the time for LSH search
    search_time = 0
    for i in range(repeat_times[1]):
        start_time = time.time()
        lsh.search("target.jpg")
        search_time += (time.time() - start_time)
    search_time /= repeat_times[1]

    return init_time, search_time


def generate_scatter_indicator(
        bucket_num: int
):
    """
    Generate scatter indicators for LSH
    """
    assert 1 <= bucket_num <= 24
    indicators = []
    indicators = np.linspace(1, 24, bucket_num, dtype=int)
    return indicators


def plot_init_search(
        bucket_nums: range = range(1, 25),
        repeat_times: Tuple[int, int] = (500, 100),
```

```
        save_path: str = "results/time_init_search.png"
):
    """
    Compare the initialization time and search time of different indicators
    """
    # Compute the time for LSH initialization and search
    lsh_init_times = []
    lsh_search_times = []
    for bucket_num in tqdm(bucket_nums, desc="Different Dimension of Projection Set"):
        indicators = generate_scatter_indicator(bucket_num)
        init_time, search_time = compute_lsh_time(indicators, repeat_times)
        lsh_init_times.append(init_time)
        lsh_search_times.append(search_time)
    lsh_init_times = np.array(lsh_init_times)
    lsh_search_times = np.array(lsh_search_times)

    # Plot initialization time
    fig, ax1 = plt.subplots(figsize=(8, 6))
    ax1.plot(bucket_nums, lsh_init_times, marker='o', label="Initialization Time",
         color='tab:blue')
    ax1.set_xlabel("Dimension of Projection Set")
    ax1.set_ylabel("Initialization Time (s)", color='tab:blue')
    ax1.set_ylim(min(lsh_init_times) * 0.95, max(lsh_init_times) * 1.05)
    ax1.tick_params(axis='y', labelcolor='tab:blue')

    # Plot search time
    ax2 = ax1.twinx()
    ax2.plot(bucket_nums, lsh_search_times, marker='o', label="Search Time",
         color='tab:red')
    ax2.set_ylabel("Search Time (s)", color='tab:red')
    ax2.tick_params(axis='y', labelcolor='tab:red')

    # Plot settings
    plt.grid(True, alpha=0.5)
    ax1.legend(loc='upper left', bbox_to_anchor=(0.1, 0.97))
    ax2.legend(loc='upper left', bbox_to_anchor=(0.1, 0.9))
    plt.tight_layout()
    plt.savefig(save_path, dpi=300)
    plt.show()


def plot_lsh_knn(
        bucket_nums: range = range(1, 25),
        repeat_times: int = 100,
```

```python
        save_path: str = "results/time_comp.png"
):
    """
    Compare the searching speedup of LSH over KNN
    """
    # Compute the time for KNN search
    knn_time = compute_knn_time(repeat_times)

    # Compute the time for LSH search
    lsh_times = []
    for bucket_num in tqdm(bucket_nums, desc="Different Dimension of Projection Set"):
        indicators = generate_scatter_indicator(bucket_num)
        _, lsh_time = compute_lsh_time(indicators, (1, repeat_times))
        lsh_times.append(lsh_time)
    lsh_times = np.array(lsh_times)

    # Compute the speedup
    speedup = knn_time / lsh_times

    # Plot the speedup
    fig = plt.figure(figsize=(8, 6))
    plt.plot(bucket_nums, speedup, marker='o')
    plt.xlabel("Dimension of Projection Set")
    plt.ylabel("Speedup")
    plt.grid(alpha=0.5)
    plt.savefig(save_path, dpi=300)
    plt.show()


def plot_similarity(
        resnet: bool = False
):
    """
    Plot the similarity between the target image and all dataset images
    """
    # Compute feature vector for the target image
    target_img = Image("target.jpg", resnet)
    target_vec = target_img.feature_vec
    target_vec = target_vec / np.linalg.norm(target_vec)

    # Compute similarity between the target image and all dataset images
    similarities = []
    for i in range(1, 41):
        img = Image(f"dataset/{i}.jpg", resnet)
```

```
        img_vec = img.feature_vec
        img_vec = img_vec / np.linalg.norm(img_vec)
        similarities.append(np.dot(target_vec, img_vec))

    # Plot the similarity
    fig = plt.figure(figsize=(10, 5))
    plt.bar(range(1, 41), similarities)
    plt.xlabel("Image Index")
    plt.ylabel("Similarity with Target Image")
    plt.xlim(0, 41)
    plt.xticks(range(1, 41), fontsize=10)
    plt.grid(axis='y', alpha=0.5)
    plt.tight_layout()
    save_path = "results/similarity_new.png" if not resnet else "results/similarity_resnet_new.png"
    plt.savefig(save_path, dpi=300)
    plt.show()


def main(args):
    if args[0] == "time_comp":
        plot_lsh_knn(range(1, 25), 100, "results/time_comp_new.png")
    elif args[0] == "time_init_search":
        plot_init_search(range(1, 25), (500, 100), "results/time_init_search_new.png")
    elif args[0] == "similarity":
        plot_similarity(resnet=False)
    elif args[0] == "similarity_resnet":
        plot_similarity(resnet=True)


if __name__ == '__main__':
    main(sys.argv[1:])
```

```
"""
File name: preprocess.py
"""
import cv2
import numpy as np
import torch
import torch.nn as nn
import torchvision.transforms as transforms
from torchvision import models
from PIL import Image as PILImage

```

```
class Image:
    """
    Image class
    """

    def __init__(
            self,
            img_path: str,
            resnet: bool = False,
            normalize: bool = True
    ):
        self.path = img_path
        img = cv2.imread(img_path)
        if not resnet:
            self.feature_vec = extract_color_feature(img, normalize)
        else:
            self.feature_vec = extract_resnet_feature(img)


def extract_color_feature(
        img: np.ndarray,
        normalize: bool = True
) -> np.ndarray:
    """
    Generate color feature vector for the given image
    """
    H, W, C = img.shape
    half_H, half_W = H // 2, W // 2

    # Calculate the sum of RGB values in each quadrant
    rgb = []
    for i in range(2):
        for j in range(2):
            quadrant = img[i * half_H: (i + 1) * half_H, j * half_W: (j + 1) * half_W]
            quadrant_sum = np.sum(quadrant, axis=(0, 1))
            if normalize:
                rgb.extend(quadrant_sum / np.sum(quadrant_sum))
            else:
                rgb.extend(quadrant_sum)
    rgb = np.array(rgb)

    # Generate feature vector
    lb = min(rgb) + (max(rgb) - min(rgb)) / 3
```

```
    ub = max(rgb) - (max(rgb) - min(rgb)) / 3
    feature_vec = np.ones(12, dtype=np.uint8)
    feature_vec[rgb >= ub] = 2
    feature_vec[rgb <= lb] = 0
    return feature_vec


def extract_resnet_feature(
        img: np.ndarray
) -> np.ndarray:
    """
    Generate feature vector for the given image using ResNet
    """
    model = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)
    model.eval()
    img = preprocess_resnet(img)
    with torch.no_grad():
        features = model(img)
    features = features.squeeze() # Remove the batch dimension
    feature_vector = features[:12] # Extract the first 12 features
    return feature_vector.numpy()


def preprocess_resnet(
        img: np.ndarray
):
    """
    Preprocess the image for ResNet
    """
    transform = transforms.Compose([
        transforms.Resize((224, 224)), # Resize to 224x224
        transforms.ToTensor(),       # Convert to tensor
        transforms.Normalize(        # Normalize
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]
        )
    ])
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = PILImage.fromarray(img) # Convert to PIL image
    img = transform(img).unsqueeze(0) # Add batch dimension
    return img
```