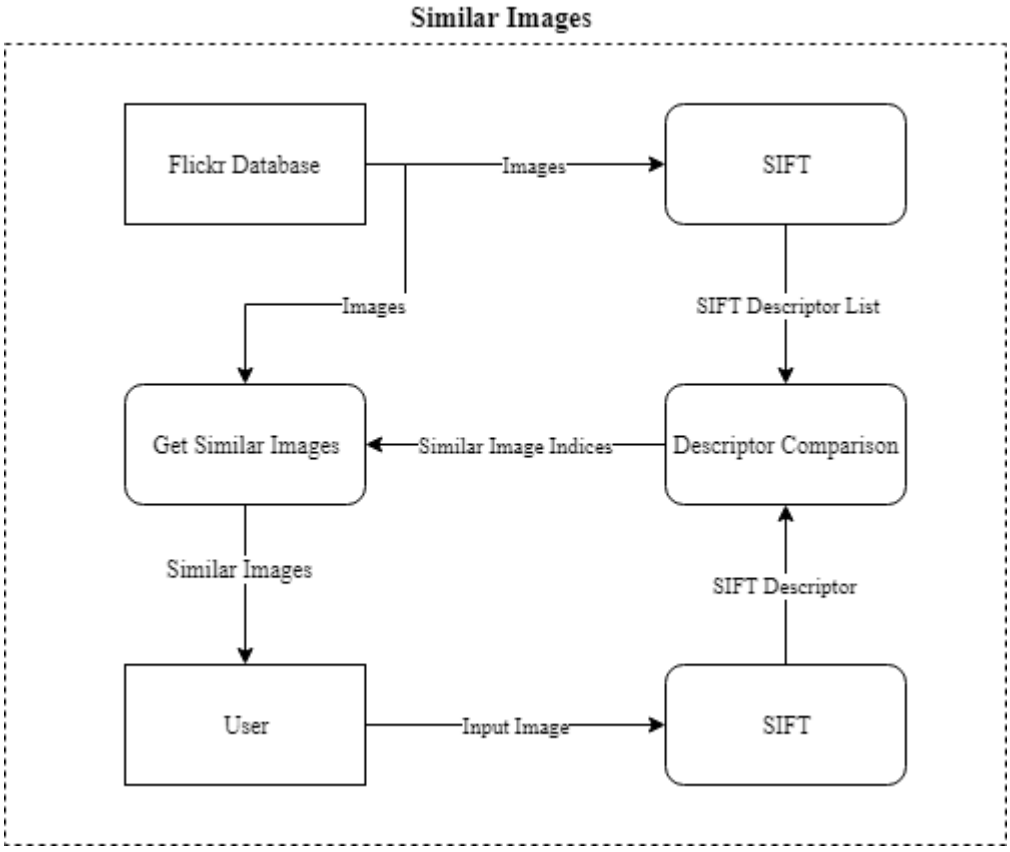
Introduction

Image similarity matching is an important problem in computer vision. The fundamental problem is, given an input image, find the most similar images from a list of images. There are many different approaches to doing this, from naïve implementations that just use pixel intensities, to CNNs and more advanced methods like that. This implementation uses SIFT feature descriptors to calculate a similarity metric between two images, which then are compared to find the image with the highest similarity, which is likely the most similar image. In this implementation, the list of images is retrieved from a Flickr database by tag using a Flickr API[1], the images were all generally the same size to make it easy to compare descriptors. Then the images are stored on the local file system, and later retrieved for analysis. The process by which the images are transformed and compared is shown below.



Most of the functionality is provided by OpenCV, which provides SIFT in a non-free extension, and a brute force matcher in the BFMatcher class, however since SIFT is patented this could not be used for non-academic purposes. The BFMatcher class provides keypoint matching between two images, by the finding the closest keypoint to the original one using a brute force algorithm, the cross-check option is

[1] https://github.com/alexis-mignon/python-flickr-api

used to exclude any potential false positives by assuring that the matched descriptor in the original image has the matched descriptor in the test image as a best match, and vice versa.

SIFT

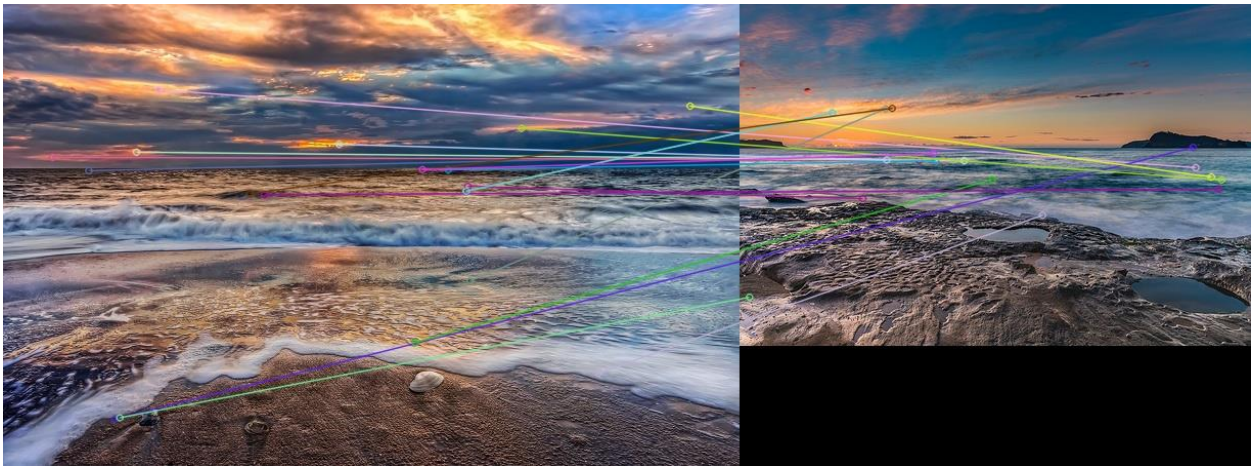
SIFT is the algorithm that was used to calculate the image descriptors for this project, I will only give a brief introduction to the algorithm as a more detailed introduction can be found in Lowe's paper[2]. SIFT stands for scale invariant feature transform, as it can find common keypoints that are invariant to scaling and orientation. It does this through a complicated process that involves applying the gaussian filters to a scale pyramid of the image, and then subtracting, producing the difference of gaussians. Then the extrema are found, sampled, and then further processes are done to find the best keypoint candidates.

Similarity

Determining the similarity between two images for comparison requires that some method of finding a numerical similarity between the two images be found. The method by which this is done in this project is by an average similarity across keypoints. To match two keypoints in different images, a brute force approach is used which finds the closest keypoint, which uses cross checking where both keypoints must agree that they're the best match. Once the all the matchings have been found, the similarity between the two keypoint descriptors can be calculated as follows, where d_1 and d_2 are the two keypoint descriptors.

$$sim(d_1, d_2) = \frac{1}{1 + dist(d_1, d_2)}$$

This bounds the similarity in the range (0, 1], where a similarity of 1.0 indicates that the two images are the same. While a similarity close to zero would indicate that the two images are dissimilar. The similarity scores for all keypoints are averaged to get the overall similarity between the two images.



Matching of closest keypoints between two images from Flickr, the similarity between the two is 0.0039

[2] Refer to references for the paper.

Results

Results were promising but not as far reaching as I hoped, though 1.0 similarity was achieved when both images were the same, it decreased considerably when images were even slightly different. For example, for two images of the same scene but at a slightly different orientation, the similarity dropped to 0.00878, that being said it was still able to determine that they were the closest images in the dataset for that tag, and so it is able to determine which scenes are the same despite orientation. It seems to work well when images are generally the same but with minor differences, if an image however is distinct from any other image in the dataset then the closest image found is entirely unrelated to the original image.

Original Image and Most Similar Image, with similarity 0.005296

In the example above, you can see that the two images are of the same scene, but different orientations. This works for any similar enough images in the dataset, these two images just happened to be from the desert tagged dataset which had about 99 images in it. So out of 99 images it was able to discern which image had the closest descriptor match.

Conclusion

Though image matching seems like a difficult problem, it was a bit easier than I thought with SIFT doing most of the work. Implementing SIFT would have taken a lot more time, and my own implementation wouldn't have been able to perform as well. As for the accuracy, it usually found the most similar image, though usually that was just the same scene with a different orientation, or with slight changes, if both of those images were in the dataset. I expected it to be able to discern images that are of the same, or similar scenes but with different orientations, which it was able to do very well. This was due mostly I believe to the orientation invariance of SIFT. I think further improvements can be made with the similarity measure, maybe including descriptor angular orientation into judging the similarity.

References

D. G. Lowe, “Distinctive Image Features from Scale-Invariant Keypoints,” *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, Jan. 2004.

“Feature Matching¶,” Image Denoising - OpenCV-Python Tutorials 1 documentation. [Online]. Available: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_matcher/py_matcher.html. [Accessed: 13-Dec-2018].

Appendix

Source Code

flickr_manager.py

```
from flickr_api import Photo
import os


def save_images_by_tag(tag):
    """
    Saves all the images of a specific tag by flickr search.
    :param tag: Tag to search for
    :return:
    """
    path = os.path.join(os.path.dirname(os.path.realpath(__file__)), 'images',
tag)
    if not os.path.isdir(path):
        os.chdir('images')
        os.mkdir(tag)
    for photo in Photo.search(tags=tag):
        photo.save(os.path.join(path, photo.id), size_label='Medium 640')


if __name__ == '__main__':
    save_images_by_tag('jungle')
```

sift_similarity.py

```
import cv2
import numpy as np
import os



def similarity(des1, des2):
    """
    The similarity between two image descriptors by finding the closest
    neighbor for each vector in des1, and then averaging them. Then averaging this distance
    to find the distance between des1 and des2.
    :param des1: First list of image descriptors
    :param des2: Second list of image descriptors
    :return: Distance between the two
    """
    matcher = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    matches = matcher.match(des1, des2)
    distances = np.array([1 / (1 + m.distance) for m in matches])
    return np.sum(distances) / len(distances)



def closest_image(source_im, images):
    """
    Finds the closest image in images to im using SIFT similarity.
```

```
    :param source_im: source image
    :param images: images to compare against
    :return: the index in images of the most similar image
    """
    _, source_des = apply_sift(source_im)
    similarities = []
    for img in images:
        _, des = apply_sift(img)
        similarities.append(similarity(source_des, des))
    max_index = np.argmax(similarities)
    print('Similarity: {:3f}'.format(similarities[max_index]))
    return max_index


def apply_sift(img):
    """
    Apply SIFT to an image.
    :param img: Image to apply
    :return: SIFT keypoints
    """
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    sift = cv2.xfeatures2d.SIFT_create()
    kp, des = sift.detectAndCompute(gray, None)
    return kp, des


def display_closest_image(i, images):
    closest_im_index = closest_image(images[i],
                                     images[:i] + images[i+1:])
    closest_img = images[closest_im_index]
```

```
    cv2.imwrite('img1.jpg', images[i])
    cv2.imwrite('img2.jpg', closest_img)
    cv2.imshow('original', images[i])
    cv2.imshow('most similar', closest_img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()


if __name__ == '__main__':
    path = os.path.join(os.path.dirname(os.path.realpath(__file__)), 'images',
'desert')
    images = []
    for fname in os.listdir(path):
        img_path = os.path.join(path, fname)
        images.append(cv2.imread(img_path, cv2.IMREAD_COLOR))
    display_closest_image(5, images)
```