CSE 5524 Project Report

Name: Neng Shi

Project Title

Content-Based Image Retrieval and Classification for Geographic Images

Project Description

This project investigates content-based image retrieval and classification for geographic images. The key to retrieval or classification is to have representative feature descriptors. Therefore, I explore and implement different feature descriptors in this project to see which fits best for geographical images. The feature descriptors in this project include:

- the raw image,

- (Pyramid) Color Histogram,
- Similitude Moments,
- (Pyramid) Histogram of Gradient Directions.

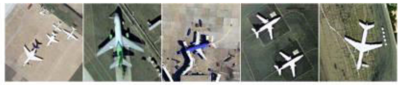
Meanwhile, I explore different similarity-matching metrics to obtain a better retrieval and classification result, which include:

- Sum-of-absolute differences (SAD),
- Sum-of-squared differences (SSD),
- Normalized cross-correlation (NCC).

Dataset

In this project, I used a dataset called UC-Merced [1] built by USGS. It is a 256 x 256 pixel remote sensing image dataset with a spatial resolution of 0.3m per pixel. It has 21 land categories with 100 images for each type of land, for a total of 2,100 images. The 21 classes are agricultural, airplane, baseball diamond, beach, buildings, chaparral, dense residential, forest, freeway, golf course, harbor, intersection, medium density residential, mobile home park, overpass, parking lot, river, runway, sparse residential, storage tanks, and tennis courts. Five samples of each class are shown in Figure 1.


(a)


(b)


(c)


(d)


(e)


(f)


(g)


(h)


(i)


(j)


(k)


(l)


(m)


(n)


(o)


(p)


(q)


(r)


(s)


(t)


(u)

Figure 1: 21 land classes from UC-Merced dataset (adapted from Yang and Newsam [1]). Five samples from each class are shown above. (a) Agricultural; (b) airplane; (c) baseball diamond; (d) beach; (e) buildings; (f) chaparral; (g) dense residential; (h) forest; (i) freeway; (j) golf course; (k) harbor; (l) intersection; (m) medium density residential; (n) mobile home park; (o) overpass; (p) parking lot; (q) river; (r) runway; (s) sparse residential; (t) storage tanks; (u) tennis courts.

Algorithms/methods explored

Feature Descriptor

As mentioned in the “Project Description” section, I implemented four feature descriptors, including the raw image, Color Histogram, Similitude Moments, and Histogram of Gradient Directions (HOG). Meanwhile, for color histogram and HOG, a histogram of an entire image may not be strong enough because it discards all the spatial information. Therefore, for these two descriptors, I incorporate spatial information by implementing “Spatial Pyramid” [2], which means the color histogram and HOG in sub-regions of different levels are calculated as well. Figure 2 illustrates the histogram-building process.

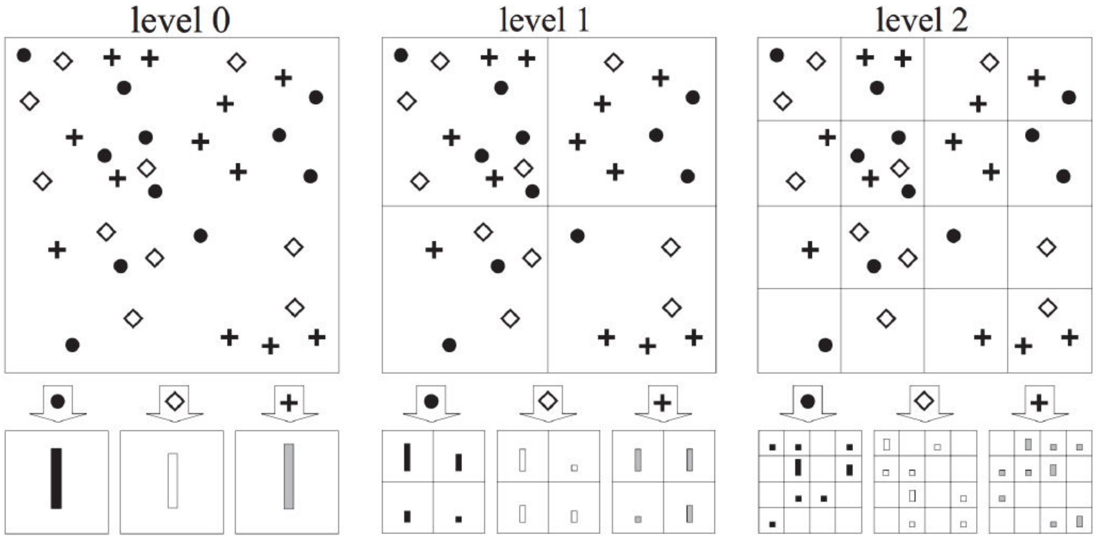


Figure 2: Spatial pyramid (Adapted from Lazebnik et al. [2])

Here are the printouts for different featuture descriptors:

(Pyramid) Color Histogram

```
import numpy as np
from skimage import color
from utils import *
```

```python
import pdb

def color_histogram(img, nbins):
    num_channels = 3
    hist = np.zeros(nbins * num_channels, dtype=np.int)
    for i in range(num_channels):
        hist[i * nbins: (i + 1) * nbins] = np.histogram(img[:, :, i], bins=nbins, range=(0., 1.))[0]

    return hist

def compute_pch(img, nbins=10, levels=2):
    hsv = color.rgb2hsv(img)
    rgb = img / 255.
    height, width = img.shape[:2]
    num_channels = 6

    coef = 1
    hist_size = 0
    for k in range(levels + 1):
        hist_size += nbins * num_channels * coef
        coef *= 4

    # Creating the descriptor.
    hist = np.zeros(hist_size, dtype=np.int)

    blocks = 1
```

```
    binPos = 0  # Next free section in the histogram
    for k in range(levels + 1):
        wstep = width // blocks
        hstep = height // blocks
        for i in range(blocks):
            for j in range(blocks):
                hist[nbins * num_channels // 2 * binPos:nbins * num_channels // 2 * (binPos + 1)] = \
                    color_histogram(rgb[i * hstep: (i+1) * hstep, j * wstep: (j+1) * wstep], nbins)
                binPos += 1
                hist[nbins * num_channels // 2 * binPos:nbins * num_channels // 2 * (binPos + 1)] = \
                    color_histogram(hsv[i * hstep: (i + 1) * hstep, j * wstep: (j + 1) * wstep], nbins)
                binPos += 1
        blocks *= 2

    hist = l2_normalize(hist)

    return hist
```

Similitude Moments

```
import numpy as np
from skimage import color
from utils import *

def similitudeMoments(im):
```

```python
    Nvals = []
    ny, nx = im.shape
    y, x  = np.arange(ny), np.arange(nx)
    yv, xv = np.meshgrid(y, x, indexing='ij')
    m00 = np.sum(im)
    m10, m01 = np.sum(xv * im), np.sum(yv * im)
    x_bar, y_bar = m10 / m00, m01 / m00
    for i in range(4):
        for iplusj in range(2, 4):
            j = iplusj - i
            if j < 0:
                continue
            # print(i, j)
            eta = np.sum((xv - x_bar)**i * (yv - y_bar)**j * im) / (np.sum(im) ** ((i+j)/2. + 1.))
            Nvals.append(eta)
    return np.array(Nvals)

def compute_color_moment(img):
    num_channels = 6
    moment_items = 7
    moments = np.zeros(moment_items * num_channels)
    for i in range(3):
        moments[i * moment_items: (i + 1) * moment_items] = similitudeMoments(img[:, :, i])

    hsv = color.rgb2hsv(img)
    for i in range(3):
```

```
        moments[(i + 3) * moment_items: (i + 4) * moment_items] = similitudeMoments(hsv[:, :, i])

    moments = l2_normalize(moments)
    return moments
```

(Pyramid) Histogram of Gradient Directions

```
import numpy as np
import math
from skimage.feature import canny
from skimage.filters import sobel_h, sobel_v
from skimage import io
from utils import *
from matplotlib import pyplot as plt

import pdb

PI_OVER_TWO = np.pi / 2.0

def getHistogram(edges, ors, mag, startX, startY, width, height, nbins):
    hist = np.zeros(nbins)
    for y in range(startY, startY + height):
        for x in range(startX, startX + width):
            if edges[y, x] > 0:
```

```
            bin = math.floor(ors[y, x])
            if bin == nbins:
                bin -= 1
            hist[bin] += mag[y, x]
    return hist

def compute_phog(img, nbins, levels):
    # io.imshow(img)
    # plt.show()
    height, width = img.shape[:2]

    # Determine desc size
    coef = 1
    desc_size = 0
    for k in range(levels + 1):
        desc_size += nbins * coef
        coef *= 4

    # Convert the image to grayscale
    if img.shape[2] == 3:
        img = (img[:, :, 0] * 0.3 + img[:, :, 1] * 0.59 + img[:, :, 2] * 0.11) / 255.
    # io.imshow(img)
    # plt.show()

    # Apply Canny Edge Detector
    mean = np.mean(img)
```

```
edges = canny(img,
    low_threshold=0.66 * mean,
    high_threshold=1.33 * mean).astype('int') * 255
# io.imshow(edges)
# plt.show()

# Computing the gradients.
grad_x = sobel_h(img)
grad_y = sobel_v(img)
# io.imshow(grad_x)
# plt.show()
# io.imshow(grad_y)
# plt.show()

# Total Gradient
grad_m = np.sqrt(grad_x ** 2 + grad_y ** 2)
# io.imshow(grad_m)
# plt.show()

# Computing orientations
grad_o = np.zeros((height, width), dtype=np.float32)
for y in range(height):
    for x in range(width):
        if grad_x[y, x] != 0.0:
            grad_o[y, x] = math.atan(grad_y[y, x] / grad_x[y, x])
        else:
```

```
            grad_o[y, x] = PI_OVER_TWO
# io.imshow(grad_o)
# plt.show()

# Quantizing orientations into bins.
grad_o = (grad_o / np.pi + 0.5) * nbins

# Creating the descriptor.
desc = np.zeros(desc_size, dtype=np.float32)

blocks = 1
binPos = 0 # Next free section in the histogram
for k in range(levels + 1):
    wstep = width // blocks
    hstep = height // blocks
    for i in range(blocks):
        for j in range(blocks):
            desc[nbins * binPos:nbins * (binPos + 1)] = \
                getHistogram(edges, grad_o, grad_m, i * wstep, j * hstep, wstep, hstep, nbins)
            binPos += 1
    blocks *= 2

desc = l2_normalize(desc)

return desc
```

And below shows the code that calls the previous functions:

```
import os
import argparse
import numpy as np
from skimage.io import imread
from skimage.transform import resize
from pch import *
from phog import *
from color_moment import *

import pdb

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--color-bins', type=int, default=10, help='Number of color bins')
    parser.add_argument('--orient-bins', type=int, default=30, help='Number of orientation bins')
    args = parser.parse_args()
    print(args)

    classes = ["agricultural", "airplane", "baseballdiamond", "beach", "buildings", "chaparral", "denseresidential",
               "forest", "freeway", "golfcourse", "harbor", "intersection", "mediumresidential", "mobilehomepark",
               "overpass", "parkinglot", "river", "runway", "sparseresidential", "storagetanks", "tenniscourt"]
    n_classes = len(classes)
    size_per_class = 100
```

```python
    for i in range(n_classes):
        for j in range(size_per_class):
            img_name = os.path.join("UCMerced_LandUse", "Images", classes[i], "{}{:02d}.tif".format(classes[i], j))
            img = imread(img_name)
            if img.shape[0] != 256 or img.shape[1] != 256:
                img = resize(img, (256, 256))

            save_dict = os.path.join("UCMerced_LandUse", "Features")
            for k in range(3):
                color_pch = compute_pch(img, nbins=args.color_bins, levels=k)
                np.save(os.path.join(save_dict, "pch", "level{:d}".format(k), classes[i], "{}{:02d}".format(classes[i], j)),
                        color_pch)

            color_mome = compute_color_moment(img)
            np.save(os.path.join(save_dict, "cm", classes[i], "{}{:02d}".format(classes[i], j)), color_mome)

            for k in range(3):
                texture_phog = compute_phog(img, nbins=args.orient_bins, levels=k)
                np.save(os.path.join(save_dict, "phog", "level{:d}".format(k), classes[i], "{}{:02d}".format(classes[i], j)),
                        texture_phog)

        print("finish processing class {}".format(classes[i]))

if __name__ == '__main__':
    main()
```

Similarity-Matching Metrics

As mentioned in the “Project Description” section, I applied three feature similarity-matching metrics, including Sum-of-absolute differences (SAD), Sum-of-squared differences (SSD), and Normalized cross-correlation (NCC). The code printouts below illustrate the process that I use different feature descriptors and similarity-matching metrics for the distance matrix.

```
import pdb

import os
import argparse
import numpy as np
from skimage.io import imread
from skimage.transform import resize

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--feat-type', type=str, required=True, help='raw, cm, pch, phog')
    parser.add_argument('--level', type=int, default=0, help='level for pch or phog')
    parser.add_argument('--dist-metric', type=str, required=True, help='SAD, SSD, NCC')
    args = parser.parse_args()
    print(args)

    classes = ["agricultural", "airplane", "baseballdiamond", "beach", "buildings", "chaparral", "denseresidential",
```

```
           "forest", "freeway", "golfcourse", "harbor", "intersection", "mediumresidential", "mobilehomepark",
           "overpass", "parkinglot", "river", "runway", "sparseresidential", "storagetanks", "tenniscourt"]

n_classes = len(classes)
train_size, val_size, test_size = 80, 10, 10
valtest_size = val_size + test_size
size_per_class = train_size + val_size + test_size
feats = []

if args.feat_type == "raw":
    feat_dir = os.path.join("UCMerced_LandUse", "Images")
    for i in range(n_classes):
        for j in range(size_per_class):
            feat_name = os.path.join(feat_dir, classes[i], "{}{:02d}.tif".format(classes[i], j))
            feat = imread(feat_name)
            if feat.shape[0] != 256 or feat.shape[1] != 256:
                feat = resize(feat, (256, 256))
            feats.append(feat)
elif args.feat_type == "pch" or args.feat_type == "phog" or args.feat_type == "cm":
    if args.feat_type == "pch" or args.feat_type == "phog":
        feat_dir = os.path.join("UCMerced_LandUse", "Features", args.feat_type, "level{:d}".format(args.level))
    else:
        feat_dir = os.path.join("UCMerced_LandUse", "Features", args.feat_type)
    for i in range(n_classes):
        for j in range(size_per_class):
            feat_name = os.path.join(feat_dir, classes[i], "{}{:02d}.npy".format(classes[i], j))
```

```python
            feat = np.load(feat_name)
            feats.append(feat)

rnd_idx = np.load("rnd_idx.npy")

dist = np.zeros((n_classes * valtest_size, n_classes * train_size))
for i in range(n_classes):
    for j in range(valtest_size):
        valtest_feat = feats[i * size_per_class + rnd_idx[train_size + j]]
        for p in range(n_classes):
            for q in range(train_size):
                train_feat = feats[p * size_per_class + rnd_idx[q]]
                if args.dist_metric == "SAD":
                    dist[i * valtest_size + j, p * train_size + q] = abs(valtest_feat - train_feat).sum()
                elif args.dist_metric == "SSD":
                    dist[i * valtest_size + j, p * train_size + q] = ((valtest_feat - train_feat) ** 2).sum()
                elif args.dist_metric == "NCC":
                    for k in range(3):
                        dist[i * valtest_size + j, p * train_size + q] += \
                            ((valtest_feat[:, :, k] - valtest_feat[:, :, k].mean()) * \
                            (train_feat[:, :, k] - train_feat[:, :, k].mean())).sum() / \
                            np.std(valtest_feat, ddof=1) * np.std(train_feat, ddof=1) * (feats[0].shape[0] * feats[0].shape[1] - 1)

    print("finish processing class {}".format(classes[i]))

if args.dist_metric == "SAD":
```

```
        np.save(os.path.join(feat_dir, "SAD"), dist)
    elif args.dist_metric == "SSD":
        np.save(os.path.join(feat_dir, "SSD"), dist)
    elif args.dist_metric == "NCC":
        np.save(os.path.join(feat_dir, "NCC"), dist)

if __name__ == '__main__':
    main()
```

Classifciation Method

I applied k-Nearest Neighbors (kNN) for classification. Meanwhile, I split the dataset into training, validation, and testing set, and used the validation set to select the best descriptor and metric. The code printout for kNN is listed below:

```
import os
import argparse
import numpy as np
from scipy import stats

import pdb

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--k', type=int, default=5, help='parameter k for KNN')
    parser.add_argument('--feat-type', type=str, required=True, help='raw, cm, pch, phog')
```

```python
parser.add_argument('--level', type=int, default=0, help='level for pch or phog')
parser.add_argument('--dist-metric', type=str, help='SAD, SSD, NCC')
parser.add_argument('--dataset', type=str, help='val or test')

args = parser.parse_args()
print(args)

classes = ["agricultural", "airplane", "baseballdiamond", "beach", "buildings", "chaparral", "denseresidential",
           "forest", "freeway", "golfcourse", "harbor", "intersection", "mediumresidential", "mobilehomepark",
           "overpass", "parkinglot", "river", "runway", "sparseresidential", "storagetanks", "tenniscourt"]

n_classes = len(classes)
train_size, val_size, test_size = 80, 10, 10

if args.feat_type == "raw":
    feat_dir = os.path.join("UCMerced_LandUse", "Images")
elif args.feat_type == "pch" or args.feat_type == "phog" or args.feat_type == "cm":
    if args.feat_type == "pch" or args.feat_type == "phog":
        feat_dir = os.path.join("UCMerced_LandUse", "Features", args.feat_type, "level{:d}".format(args.level))
    else:
        feat_dir = os.path.join("UCMerced_LandUse", "Features", args.feat_type)

if args.dist_metric == "SAD":
    dist = np.load(os.path.join(feat_dir, "SAD.npy"))
elif args.dist_metric == "SSD":
    dist = np.load(os.path.join(feat_dir, "SSD.npy"))
```

```python
elif args.dist_metric == "NCC":
    dist = -np.load(os.path.join(feat_dir, "NCC.npy"))

candi = []
if args.dataset == "val":
    for i in range(n_classes):
        candi.extend(np.arange(i * (val_size + test_size), i * (val_size + test_size) + val_size))
    valtest_size = val_size
elif args.dataset == "test":
    for i in range(n_classes):
        candi.extend(np.arange(i * (val_size + test_size) + val_size, (i + 1) * (val_size + test_size)))
    valtest_size = test_size
candi = np.array(candi)
dist = dist[candi]

confusion = np.zeros((n_classes, n_classes), dtype=int)
acc = 0
for i in range(n_classes):
    for j in range(valtest_size):
        neighbors = np.argsort(dist[i * valtest_size + j])[:args.k]
        neighbor_class_idx = neighbors // train_size
        pred = stats.mode(neighbor_class_idx)[0][0]
        acc += pred == i
        confusion[pred][i] += 1
acc /= n_classes * valtest_size
print("Overall accuracy: {:.3f}".format(acc))
```

```
for i in range(n_classes):
    binary_conf = np.zeros((2, 2), dtype=int)
    not_i = np.ones(n_classes, dtype=bool)
    not_i[i] = False
    binary_conf[0, 0] = confusion[i][i]
    binary_conf[0, 1] = confusion[i][not_i].sum()
    binary_conf[1, 0] = confusion[not_i][:, i].sum()
    binary_conf[1, 1] = confusion[not_i][:, not_i].sum()

    precision = binary_conf[0, 0] / (binary_conf[0, 0] + binary_conf[0, 1])
    recall = binary_conf[0, 0] / (binary_conf[0, 0] + binary_conf[1, 0])
    f1 = 2 * precision * recall / (precision + recall)
    print("Class {},\tprecision: {:.3f},\trecall: {:.3f},\tF1: {:.3f}".format(classes[i], precision, recall, f1))

fig, ax = plt.subplots(figsize=(13, 13))
ax.matshow(confusion, cmap=plt.cm.Blues, alpha=0.3)
for i in range(confusion.shape[0]):
    for j in range(confusion.shape[1]):
        ax.text(x=j, y=i, s=confusion[i, j], va='center', ha='center', size='medium')
plt.xlabel('Actuals', fontsize=18)
plt.ylabel('Predictions', fontsize=18)
x_major_locator=MultipleLocator(1)
y_major_locator=MultipleLocator(1)
ax.xaxis.set_major_locator(x_major_locator)
ax.yaxis.set_major_locator(y_major_locator)
```

```
    ax.set_xticklabels(['']+classes, rotation=90, fontsize=8)
    ax.set_yticklabels(['']+classes, fontsize=8)
    plt.show()

if __name__ == '__main__':
    main()
```

Results

Retrieval

I first evaluate different feature descriptors and similarity-matching metrics in this subsection through the image retrieval task. For the dataset split, 80% of the data are in the training set, and both the validation and testing set contain 10% of the dataset. For each query image, I calculated the ratio of retrieved images in the same class as the query image. The following code printouts show the evaluation proces:

```
import os
import argparse
import numpy as np

import pdb

def main():
    parser = argparse.ArgumentParser()
```

```python
parser.add_argument('--num-retrieved', type=int, default=12, help='Number of images retrieved')
parser.add_argument('--feat-type', type=str, required=True, help='raw, cm, pch, phog')
parser.add_argument('--level', type=int, default=0, help='level for pch or phog')
parser.add_argument('--dist-metric', type=str, help='SAD, SSD, NCC')
parser.add_argument('--dataset', type=str, help='val or test')
args = parser.parse_args()
print(args)

classes = ["agricultural", "airplane", "baseballdiamond", "beach", "buildings", "chaparral", "denseresidential",
           "forest", "freeway", "golfcourse", "harbor", "intersection", "mediumresidential", "mobilehomepark",
           "overpass", "parkinglot", "river", "runway", "sparseresidential", "storagetanks", "tenniscourt"]

n_classes = len(classes)
train_size, val_size, test_size = 80, 10, 10

if args.feat_type == "raw":
    feat_dir = os.path.join("UCMerced_LandUse", "Images")
elif args.feat_type == "pch" or args.feat_type == "phog" or args.feat_type == "cm":
    if args.feat_type == "pch" or args.feat_type == "phog":
        feat_dir = os.path.join("UCMerced_LandUse", "Features", args.feat_type, "level{:d}".format(args.level))
    else:
        feat_dir = os.path.join("UCMerced_LandUse", "Features", args.feat_type)

if args.dist_metric == "SAD":
    dist = np.load(os.path.join(feat_dir, "SAD.npy"))
elif args.dist_metric == "SSD":
```

```
    dist = np.load(os.path.join(feat_dir, "SSD.npy"))
elif args.dist_metric == "NCC":
    dist = -np.load(os.path.join(feat_dir, "NCC.npy"))

candi = []
if args.dataset == "val":
    for i in range(n_classes):
        candi.extend(np.arange(i * (val_size + test_size), i * (val_size + test_size) + val_size))
    valtest_size = val_size
elif args.dataset == "test":
    for i in range(n_classes):
        candi.extend(np.arange(i * (val_size + test_size) + val_size, (i + 1) * (val_size + test_size)))
    valtest_size = test_size
candi = np.array(candi)
dist = dist[candi]

precisions = np.zeros(n_classes)
for i in range(n_classes):
    for j in range(valtest_size):
        neighbors = np.argsort(dist[i * valtest_size + j])[:args.num_retrieved]
        belongs = neighbors // train_size == i
        precisions[i] += belongs.sum()
    precisions[i] /= args.num_retrieved * valtest_size
    print("The accuracy for class {} is {:.3f}".format(classes[i], precisions[i]))

print("The overall accuracy is {:.3f}".format(precisions.mean()))
```

```python
if __name__ == '__main__':
    main()
```

Table 1, Table 2, Table 3, and Table 4 list the ratio while using the raw image, (Pyramid) Color Histogram, Similitude Moments, and PHOG as feature descriptors, respectively. We can see that among the four feature descriptors, the color histogram performs the best, which is explainable since different land usually leads to different colors in remote sensing images. I apply the best feature descriptor and similarity metric selected on the validation set (Pyramid Color Histogram, level = 2, SAD) to the testing set, and the ratio obtained is 0.564.

Table 1: Using the raw image as the feature descriptor, when retrieveing 5 images, the ratio of retrieved images in the same class as the query image.

Similarity-Matching Metrics	Ratio
SAD	0.121
SSD	**0.240**
NCC	0.141

Table 2: Using (Pyramid) Color Histogram as the feature descriptor, when retrieveing 5 images, the ratio of retrieved images in the same class as the query image.

Similarity-Matching Metrics & Pyramid Level	Ratio
SAD, level = 0	0.554
SAD, level = 1	0.557
SAD, level = 2	**0.569**
SSD, level = 0	0.525
SSD, level = 1	0.513
SSD, level = 2	0.522

Table 3: Using Similitude Moments as the feature descriptor, when retrieveing 5 images, the ratio of retrieved images in the same class as the query image.

Similarity-Matching Metrics	Ratio
SAD	**0.258**
SSD	0.248

Table 4: Using PHOG as the feature descriptor, when retrieveing 5 images, the ratio of retrieved images in the same class as the query image.

Similarity-Matching Metrics & Pyramid Level	Ratio
SAD, level = 0	0.352
SAD, level = 1	0.355
SAD, level = 2	**0.359**
SSD, level = 0	0.349
SSD, level = 1	0.350

SSD, level = 2	0.339

Given one query image, we can show the retrieved results. The code printouts are listed below:

```
import os
import argparse
import pdb
from collections import defaultdict
import numpy as np

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--retrieved-class', type=str, required=True, help='The class want to retrieve')
    parser.add_argument('--retrieved-idx', type=int, required=True, help='The retrieved image index in class')
    parser.add_argument('--num-retrieved', type=int, default=12, help='Number of images retrieved')
    parser.add_argument('--feat-type', type=str, required=True, help='raw, cm, pch, phog')
    parser.add_argument('--level', type=int, default=0, help='level for pch or phog')
    parser.add_argument('--dist-metric', type=str, required=True, help='SAD, SSD, NCC')
    args = parser.parse_args()
    print(args)

    classes = ["agricultural", "airplane", "baseballdiamond", "beach", "buildings", "chaparral", "denseresidential",
```

```
            "forest", "freeway", "golfcourse", "harbor", "intersection", "mediumresidential", "mobilehomepark",
            "overpass", "parkinglot", "river", "runway", "sparseresidential", "storagetanks", "tenniscourt"]

n_classes = len(classes)
train_size, val_size, test_size = 80, 10, 10
valtest_size = val_size + test_size
size_per_class = train_size + val_size + test_size

class2idx = defaultdict(lambda: -1)
for i in range(n_classes):
    class2idx[classes[i]] = i

rnd_idx = np.load("rnd_idx.npy")
ori2permed = np.zeros(size_per_class, dtype=int)
ori2permed[rnd_idx] = np.arange(size_per_class)

if args.feat_type == "raw":
    feat_dir = os.path.join("UCMerced_LandUse", "Images")
elif args.feat_type == "pch" or args.feat_type == "phog" or args.feat_type == "cm":
    if args.feat_type == "pch" or args.feat_type == "phog":
        feat_dir = os.path.join("UCMerced_LandUse", "Features", args.feat_type, "level{:d}".format(args.level))
    else:
        feat_dir = os.path.join("UCMerced_LandUse", "Features", args.feat_type)

if args.dist_metric == "SAD":
    dist = np.load(os.path.join(feat_dir, "SAD.npy"))
```

```python
    elif args.dist_metric == "SSD":
        dist = np.load(os.path.join(feat_dir, "SSD.npy"))
    elif args.dist_metric == "NCC":
        dist = -np.load(os.path.join(feat_dir, "NCC.npy"))

    retrieved_class_idx = class2idx[args.retrieved_class]
    if retrieved_class_idx == -1:
        print("The class you input is not in the database!")
        return

    retrieved_permed_idx = ori2permed[args.retrieved_idx] - train_size
    if retrieved_permed_idx < 0:
        print("You are retrieving a template image!")
        return

    neighbors = np.argsort(dist[retrieved_class_idx * valtest_size + retrieved_permed_idx])[:args.num_retrieved]
    neighbor_class_idx = neighbors // train_size
    neighbor_inclass_idx = rnd_idx[neighbors % train_size]
    for i in range(args.num_retrieved):
        print("The rank {:d} most similar image is {}{:02d}".format(i + 1, classes[neighbor_class_idx[i]], neighbor_inclass_idx[i]))

if __name__ == '__main__':
    main()
```

Figure 3 illustrates a retrieval process.

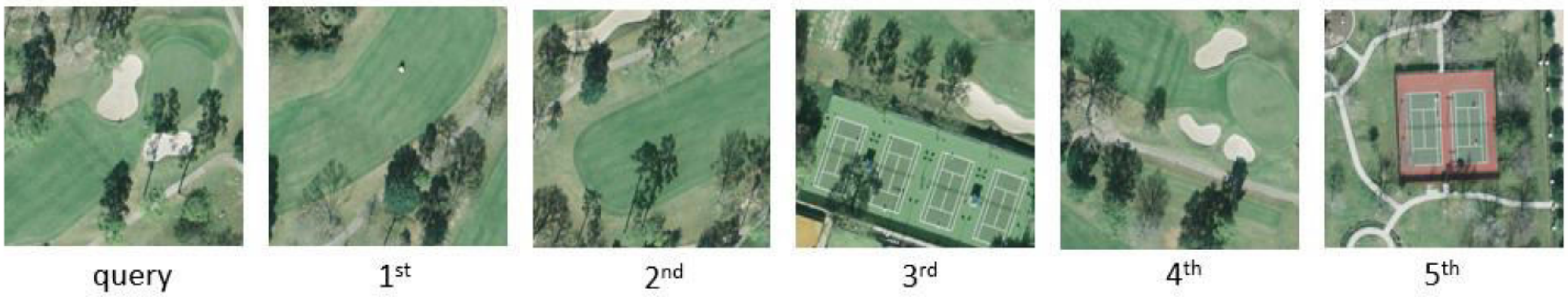


Figure 3: One query image from the golf course class and the retrieved results. The 1st, 2nd, and 4th image are from the same class as the query image, while the 3rd and 5th image belongs to the tennis court class.

Classification

We can use the retrieved images to perform classification for the query image. As we apply the kNN method, one hyper-parameter *k* needs to be tuned. Table 5 shows the results with different *k* on the validation set, and we can see that the accuracy drops as *k* increases. Therefore, I choose *k=1*.

Table 5: k-Nearest Neighbors results with different hyper-parameter *k* on the validation set.

k	Accuracy
1	**0.762**
3	0.676
5	0.657
7	0.643

The accuracy with *k=1* on the testing set is 0.700. I also list the table containing precision, recall, and F1-score for each class in Table 6. In Figure 4, I show the connfusion matrix.

Table 6: The precision, recall, and F1-score for kNN with *k=1* on the testing set.

Class	Precision	Recall	F1-score
agricultural	0.800	0.800	0.800
airplane	0.833	0.500	0.625

baseball diamond	0.714	0.500	0.588
beach	0.909	1.000	0.952
buildings	0.875	0.700	0.778
chaparral	0.625	1.000	0.769
dense residential	0.556	0.500	0.526
forest	0.643	0.900	0.750
freeway	0.500	0.300	0.375
golf course	0.818	0.900	0.857
harbor	1.000	1.000	1.000
intersection	0.714	0.500	0.588
medium density residential	0.562	0.900	0.692
mobile home park	0.750	0.900	0.818
overpass	0.500	0.400	0.444

parking lot	0.571	0.800	0.667
river	0.667	0.800	0.727
runway	0.667	0.800	0.727
sparse residential	0.692	0.900	0.783
storage tanks	0.750	0.300	0.429
tennis courts	0.750	0.300	0.429

Predictions \ Actuals	agricultural	airplane	baseballdiamond	beach	buildings	chaparral	denseresidential	forest	freeway	golfcourse	harbor	intersection	mediumresidential	mobilehomepark	overpass	parkinglot	river	runway	sparseresidential	storagetanks	tenniscourt
agricultural	8	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
airplane	0	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
baseballdiamond	0	0	5	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0
beach	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
buildings	0	0	0	0	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
chaparral	0	0	0	0	0	10	1	0	1	0	0	0	1	1	0	0	0	1	1	0	0
denseresidential	0	0	0	0	2	0	5	0	1	0	0	0	0	0	0	0	0	0	0	0	1
forest	0	0	0	0	0	0	0	9	2	0	0	0	0	0	0	1	2	0	0	0	0
freeway	0	0	0	0	0	0	0	0	3	0	0	2	0	0	1	0	0	0	0	0	0
golfcourse	1	1	0	0	0	0	0	0	0	9	0	0	0	0	0	0	0	0	0	0	0
harbor	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0
intersection	0	0	0	0	0	0	0	0	2	0	0	5	0	0	0	0	0	0	0	0	0
mediumresidential	0	0	0	0	0	0	3	0	0	0	0	1	9	0	2	0	0	0	0	0	1
mobilehomepark	0	0	0	0	0	0	0	0	0	0	0	0	0	9	0	0	0	0	0	2	1
overpass	0	0	0	0	0	0	0	0	1	0	0	1	0	0	4	0	0	0	0	1	1
parkinglot	0	2	0	0	1	0	0	0	0	0	0	1	0	0	1	8	0	0	0	1	0
river	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	8	1	0	0	0
runway	0	2	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	8	0	0	0
sparseresidential	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	1	2
storagetanks	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	1
tenniscourt	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3

Figure 4: The confusion matrix for kNN with *k=1* on the testing set.

Future Work

In the future, I would consider more feature descriptors such as (Pyramid) SIFT. Moreoever, I would explore fusing the existing features to have a more robust feature.

References

1 Yang, Y., and Newsam, S.: ‘Bag-of-visual-words and spatial extensions for land-use classification’, in Editor (Ed.)^(Eds.): ‘Book Bag-of-visual-words and spatial extensions for land-use classification’ (ACM, 2010, edn.), pp. 270-279

2 Lazebnik, S., Schmid, C., and Ponce, J.: ‘Beyond Bags of Features: Spatial Pyramid Matching for Recognizing Natural Scene Categories’, in Editor (Ed.)^(Eds.): ‘Book Beyond Bags of Features: Spatial Pyramid Matching for Recognizing Natural Scene Categories’ (2006, edn.), pp. 2169-2178

Appendix

Here is the printout of my code for vector normalization, including L1 normalization and L2 normalization.

```
import numpy as np

def l1_normalize(v):
    norm = abs(v).sum()
    if norm == 0:
        return v
    return v / norm

def l2_normalize(v):
    norm = np.linalg.norm(v)
    if norm == 0:
        return v
    return v / norm
```