Scene Classification Project

Pablo García Fernández
(pablo.garcia.fernandez2@rai.usc.es)

1 Introduction

The objective of this work is to implement a scene recognition method based on the combination of two classical descriptors: Bags of visual words (BoW) over dense SIFT features and Local Binary Patterns (LBP) *[extra credit 2]*.

BoW consists of building visual vocabularies of words that can be used to describe images by counting features occurrences. In this work, the BoW is constructed over dense SIFT features. LBP are one of the most prominent texture descriptors. It consists of characterize the spatial structure of a local image patch by encoding the differences between the pixel value of the central point and those of its neighboring points. With each of the methods, a vector describing the image is obtained in a compact form. The concatenation of the two gives the final vector to be used in the classification task. Classification is performed with a Support Vector Machine (SVM) with RBF kernel.

For achieve these goals, the Indoor Scene Recognition database [1] is provided. Only a subset composed of 10 categories (bathroom, bakery, bookstore, casino, corridor, gym, kitchen, locker_room, subway and winecellar) and 150 images per category is used in the experiments. Among the categories, a validation set of 20 images is reserved. It is used to tune the learning parameters of the SVM (specifically the regularization factor and gamma) *[extra credit 1]*.

In addition to the baseline evaluation of the method, a set of additional experiments varying the vocabulary size (number of clusters) is also performed *[extra credit 2]*.

2 Method

This section briefly describes the methodology followed to solve the problem and the decisions made. It can be divided in three parts: image description based on BoW, image description based on LBP and classification.

Appendix A includes all the code developed. The main files are `bow.py` (vocabulary building and bow image description), `lbp.py` (lbp computation for each image) and `classifier.py` (SVM and evaluation utilities). The remaining files `main.py`, `data.py` and `config.py` include general logic, data set manipulation and constant definition, respectively.

3 Image description based on BoW

Once the images have been divided into training, validation and test sets, the vocabulary building can be performed on the training samples. The first step is to obtain the dense response for each image. For this purpose a SIFT descriptor is applied densely on the basis of grid points, instead of keypoints (`bow.py, obtain_dense_features()` method). The class `DsiftExtractor` (Appendix A, `bow.py`) provided in [2] is used for this purpose. The `GRIDSPACING` and `PATCHSIZE` chosen is 8 and 16 respectively. This means that for every 8 points of an image: its 16×16 neighborhood is divided into a 4×4 matrix of cells, the orientation is quantized into 8 bins in each cell and a vector of $4 \times 4 \times 8 = 128$ dimensions is obtained as the SIFT representation for a pixel.

In addition, although it is not necessary for the correct functioning of the algorithm, as the images are of very different sizes it was decided to rescale them all to a fixed size of 250x250. This saves computation time (time increases with image size) and ensures that all images have the same influence when building the vocabulary (otherwise larger images would result in more feature vectors). To sum up, the result of this step is *no. imgs (800)* × *no. grid points per image (900) = 72000* feature vectors of size *128*.

Once the SIFT dense features are extracted, they are clustered (`bow.py, build_bow()` method). To to this, the k-means class of scikit-learn is used. The many thousands of local feature vectors from the previous step are grouped around 100 clusters, leading to a 100-size dimensional vocabulary. Each centroid represent a visual word.

Finally, each of the images can be described using the constructed vocabulary (`bow.py, extractFeatures()` method). Since the size of the vocabulary has been set to 100 visual words, the bag-of-words representation of the image is a 100-dimensional histogram. This histogram is build by classifying each of the image features and counting how many fall into each cluster.

At this point, each image is encoded and described only by a 100-dimensional compact vector.

4 Image description based on LBP

The `computeLBP()` method of `lbp.py` file computes the LBP response for a set of images using the `local_binary_pattern` functionality of the scikit-image library. In this case, no direct local binary patterns, but uniform local binary patterns (ULBP) are used.

ULBP over circular neighborhoods provides rotation invariance and low dimensionality feature vectors. It consist of computing the LBP response on a circle of radius r centered at point c. Neighbor values that are not exactly at the center of the pixels are estimated by bilinear interpolation. After the computation, only the uniforms patterns are selected. An LBP is uniform when it contains at most two transitions from 1 to 0 and/or from 0 to 1. When computing the LBP histogram, a separate value is assigned for each uniform pattern and all non-uniform patterns are assigned a single value. Furthermore, the uniforms patterns which differ only in their degree of rotation are also considered equivalent. As a result, the final dimension of the descriptor is $n + 2$ (where n is the number of neighbors).

Since in this work we use $radius = 3$ and $neighbors = 8 \times radius$, each image is described by a vector of size 26. The concatenation between the BoW vector and the LBP vector gives the final image description.

5 Classification

The classification task is performed with a Support Vector Machine (SVM) with Radial Basis Function (RBF) kernel (`classifier.py, train_test()` method). The regularization factor (λ) and the kernel parameter (γ) are tuned over the validation set. Since we have 10 classes, a 1 vs. all configuration is used.

Once the model has been trained with the best-fit parameters, the accuracy and confusion matrix metrics are used to measure the effectiveness of the approach.

6 Results

The first experiment is dedicated to analyse the effect of the vocabulary size (number of clusters). For this purpose, only the BoW description of the image is considered, excluding the LBP descriptor (i.e. the combination of the two features is not taken into account, only BoW).

Figure 1 shows the evolution of the accuracy as a function of the number of clusters. It is observed that, as the vocabulary size increases, the results improve up to saturation at 150. From this point on, the accuracy starts to decrease again. Thus, for this particular case, the optimal result is achieved with 150 clusters (48.8% acc.)

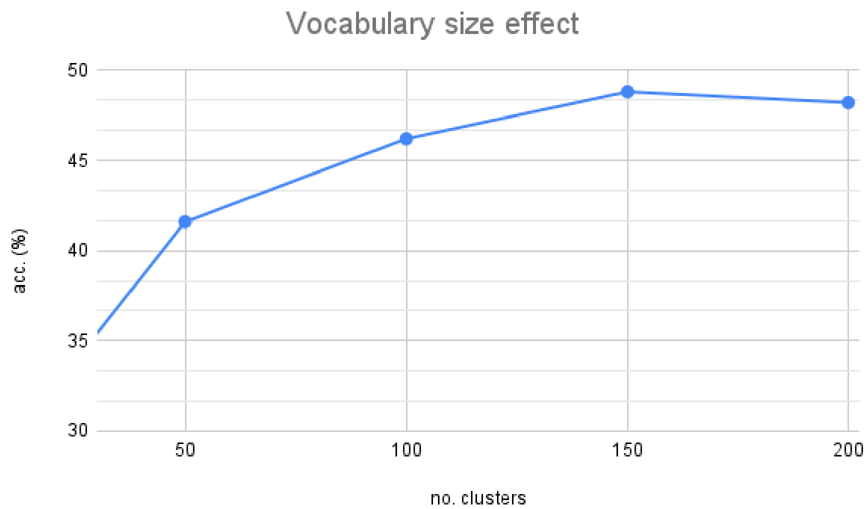
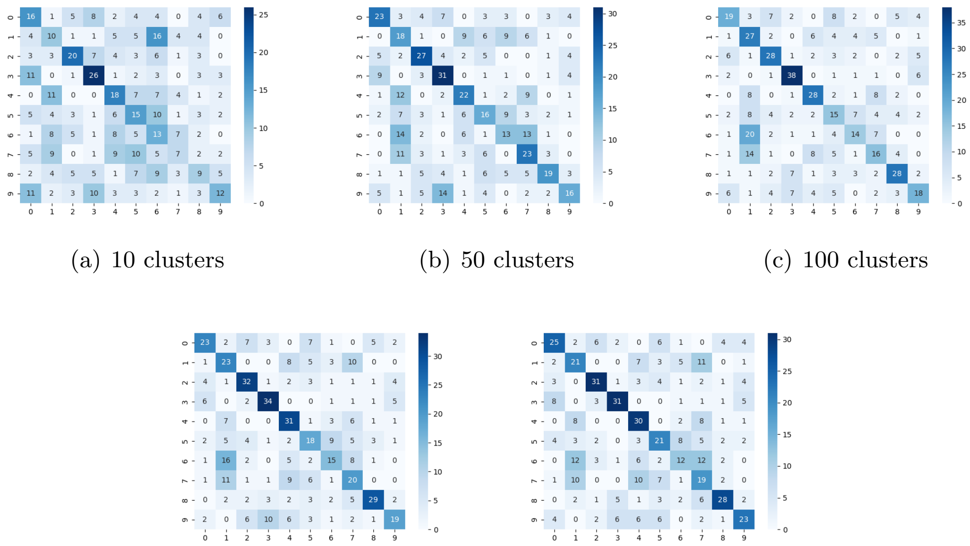


Figure 1: Evolution of acc. with no. clusters. Only the BoW descriptor is taken into account.

Figure 2 shows the corresponding confusion matrices. For 10 clusters, errors are observed to occur frequently among all classes. As the vocabulary size increases, these are concentrated around classes 5 (gym), 6 (locker_room), and 9 (winecellar). For instance, confusions between gym-kitchen, kitchen-bathroom or locker_room-bathroom are common.



(a) 10 clusters (b) 50 clusters (c) 100 clusters

(d) 150 clusters (e) 200 clusters

Figure 2: Confusion matrices. *bakery: 0, bathroom: 1, bookstore: 2, casino: 3, corridor: 4, gym: 5, kitchen: 6, locker_room: 7, subway: 8, winecellar: 9*

The second experiment analyzes the effect of the combination of LBP and BoW. Figure 3 shows the accuracy of the 3 methods: BoW alone, LBP alone and both combined. Figure 4 shows the corresponding confusion matrices.

It can be seen how the incorporation of texture information into the bag of visual words approach by calculating local binary patterns significantly improves the results. Accuracy increases from 48.8% to 57.9%.

It is also worth mentioning that, on its own, LBP is a worse descriptor than the BoW approach for this particular scene classification problem (48.8% vs 29.6%).

Finally, regarding the results between classes, the errors with the combined LBP + BoW method continue to occur mainly in the classes 5 (gym) and 6 (locker_room). However, for class 9 (winecellar) the addition of LBP greatly improved the results.

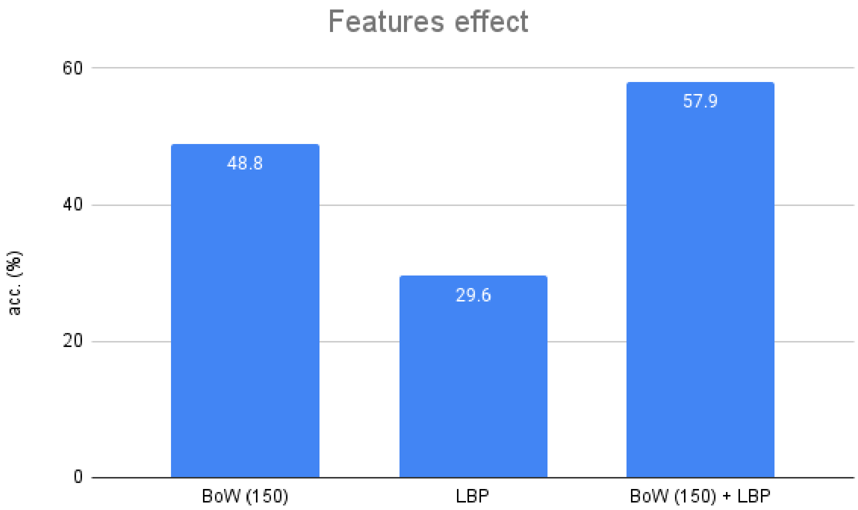
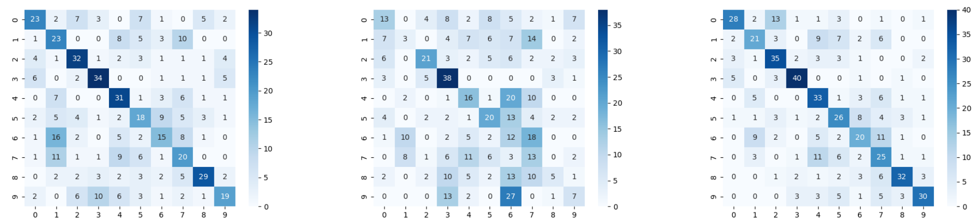


Figure 3: asd



(a) BoW (150) (b) LBP (c) BoW (150) + LBP

Figure 4: Confusion matrices. *bakery: 0, bathroom: 1, bookstore: 2, casino: 3, corridor: 4, gym: 5, kitchen: 6, locker_room: 7, subway: 8, winecellar: 9*

7 Conclusion

In this work the scene classification problem was addressed using classical techniques. It has been observed that the method based on the construction of bags of visual words, although it does not give excellent results, is better than other descriptors such as LBP (48.8% vs 29.6%). It has also been observed that increasing the vocabulary size improves the results up to a saturation threshold (for this particular problem located at 150 clusters).

Finally, and most importantly, it should be noted that the results improve considerably when the BoW approach is enriched with texture information (LBP). In doing so, the best results are obtained: **57.9% acc**, 150 clusters and SVM tuned parameters $\lambda = 8$, $\gamma = 0.0078125$.

Appendix A - Code

Main.py

```
import argparse
from modules.data import *
from modules.bow import *
from modules.classifier import *
from modules.lbp import *
from config import *
import matplotlib.pyplot as plt
import pdb
import pickle

def execute(data_path):
    # STEP 1: data acquisition. 150 images per class
    print("Taking and splitting images...")
    images_names = find_images(data_path, LABEL_MAPPER)
    train_names, val_names, test_names = train_val_test_split(images_names,
        TRAIN_N_IMAGES, VAL_N_IMAGES, TEST_N_IMAGES)

    # STEP 2: BOW construction + STEP 3: Describe each image by its histogram
         of visual features ocurrences.
    if os.path.exists(TRAIN_IMAGES_FEATURES_PATH) and os.path.exists(
        VAL_IMAGES_FEATURES_PATH) and os.path.exists(
        TEST_IMAGES_FEATURES_PATH):
        print("Loading final imgs. features...")
        train_features = np.loadtxt(TRAIN_IMAGES_FEATURES_PATH)
        val_features = np.loadtxt(VAL_IMAGES_FEATURES_PATH)
        test_features = np.loadtxt(TEST_IMAGES_FEATURES_PATH)
    else:
        sift_des = DsiftExtractor(GRIDSPACING, PATCHSIZE, 1)
        # STEP 2: BOW construction
        print("Building BOW...")
        # Get dense response for each image
        train_descriptor_list, train_labels = obtain_dense_features(
            train_names, sift_des)
        # Build BOW
        kmeans_bow = build_bow(train_descriptor_list, N_CLUSTERS)

        pickle.dump(kmeans_bow, open(BOW_PATH, "wb")) # save bow

        # STEP 3: Describe each image by its histogram of visual features
            ocurrences.
        print("TRAIN IMAGES...")
        # BOW features
        train_im_features = extractFeatures(kmeans_bow, train_descriptor_list
            , train_labels, N_CLUSTERS)
```

```
        # LBP features
        train_lbp_features = computeLBP(train_names)
        train_features = np.concatenate((train_im_features[:,:-1],
            train_lbp_features, train_im_features[:,-1,np.newaxis]), axis=1)
        # Save features
        np.savetxt(TRAIN_IMAGES_FEATURES_PATH, train_features)

        # Same with val and test
        print("TEST IMAGES...")
        val_descriptor_list, val_labels = obtain_dense_features(val_names,
            sift_des)
        # BOW features
        val_im_features = extractFeatures(kmeans_bow, val_descriptor_list,
            val_labels, N_CLUSTERS)
        # LBP features
        val_lbp_features = computeLBP(val_names)
        val_features = np.concatenate((val_im_features[:,:-1],
            val_lbp_features, val_im_features[:,-1,np.newaxis]), axis=1)
        np.savetxt(VAL_IMAGES_FEATURES_PATH, val_features)

        print("VAL IMAGES...")
        test_descriptor_list, test_labels = obtain_dense_features(test_names,
             sift_des)
        # BOW features
        test_im_features = extractFeatures(kmeans_bow, test_descriptor_list,
            test_labels, N_CLUSTERS)
        # LBP features
        test_lbp_features = computeLBP(test_names)
        test_features = np.concatenate((test_im_features[:,:-1],
            test_lbp_features, test_im_features[:,-1,np.newaxis]), axis=1)
        np.savetxt(TEST_IMAGES_FEATURES_PATH, test_features)

    # STEP 4: Train and TEST
    print("Training and testing SVC model...")
    model = train_test(train_features, val_features, test_features)
    pickle.dump(model, open(MODEL_PATH, "wb")) # save model


if __name__ == "__main__":
    # Read user arguments: input path to the images folder.
    parser = argparse.ArgumentParser(description='Scene Classification
        Project')
    parser.add_argument('-i', '--input', help='<Required> Path to the images'
                        ' directory. Must be a folder', default=None)
    args = parser.parse_args()
    images_folder = args.input

    if not images_folder:
```

```
        images_folder = DATA_PATH

    np.random.seed(88)
    execute(DATA_PATH)
```

config.py

```
import numpy as np
import os
# DATA
DATA_PATH = 'data'
RESULT_PATH = 'results'
BOW_PATH = os.path.join(RESULT_PATH, 'BOW.pkl')
MODEL_PATH = os.path.join(RESULT_PATH, 'model.pkl')
TRAIN_IMAGES_FEATURES_PATH = os.path.join(RESULT_PATH, '
    train_images_features.txt')
VAL_IMAGES_FEATURES_PATH = os.path.join(RESULT_PATH, 'val_images_features.
    txt')
TEST_IMAGES_FEATURES_PATH = os.path.join(RESULT_PATH, 'test_images_features.
    txt')

MAX_IMAGES_PER_CLASS = 150
TRAIN_N_IMAGES = 80
VAL_N_IMAGES = 20
TEST_N_IMAGES = 50


LABEL_MAPPER = {
    'bakery': 0,
    'bathroom': 1,
    'bookstore': 2,
    'casino': 3,
    'corridor': 4,
    'gym': 5,
    'kitchen': 6,
    'locker_room': 7,
    'subway': 8,
    'winecellar': 9
}


# SIFT DENSE APLICATION
GRIDSPACING = 8
PATCHSIZE = 16

NANGLES = 8
NBINS = 4
NSAMPLES = NBINS**2
```

```
ALPHA = 9.0
ANGLES = np.array(range(NANGLES))*2.0*np.pi/NANGLES

# BOW
N_CLUSTERS = 100
RESIZE_SIZE = (250, 250)

# LBP
RADIUS = 3
N_POINTS = 8 * RADIUS
METHOD = 'uniform'
BINS = 25
```

bow.py

```
import numpy as np
from scipy import signal
from config import *
import pdb
import cv2
from time import perf_counter
from sklearn.cluster import KMeans, MiniBatchKMeans

def gen_dgauss(sigma):
    '''
    generating a derivative of Gauss filter on both the X and Y direction.
    '''
    fwid = np.int(2*np.ceil(sigma))
    G = np.array(range(-fwid,fwid+1))**2
    G = G.reshape((G.size,1)) + G
    G = np.exp(- G / 2.0 / sigma / sigma)
    G /= np.sum(G)
    GH,GW = np.gradient(G)
    GH *= 2.0/np.sum(np.abs(GH))
    GW *= 2.0/np.sum(np.abs(GW))
    return GH,GW

class DsiftExtractor:
    '''
    The class that does dense sift feature extractor.
    Sample Usage:
        extractor = DsiftExtractor(gridSpacing,patchSize,[optional params])
        feaArr,positions = extractor.process_image(Image)
    '''
    def __init__(self, gridSpacing, patchSize,
                 nrml_thres = 1.0,\
                 sigma_edge = 0.8,\
                 sift_thres = 0.2):
```

```
        '''
        gridSpacing: the spacing for sampling dense descriptors
        patchSize: the size for each sift patch
        nrml_thres: low contrast normalization threshold
        sigma_edge: the standard deviation for the gaussian smoothing before
            computing the gradient
        sift_thres: sift thresholding (0.2 works well based on Lowe's SIFT
            paper)
        '''
        self.gS = gridSpacing
        self.pS = patchSize
        self.nrml_thres = nrml_thres
        self.sigma = sigma_edge
        self.sift_thres = sift_thres
        # compute the weight contribution map
        sample_res = self.pS / np.double(NBINS)
        sample_p = np.array(range(self.pS))
        sample_ph, sample_pw = np.meshgrid(sample_p,sample_p)
        sample_ph.resize(sample_ph.size)
        sample_pw.resize(sample_pw.size)
        bincenter = np.array(range(1,NBINS*2,2)) / 2.0 / NBINS * self.pS -
            0.5
        bincenter_h, bincenter_w = np.meshgrid(bincenter,bincenter)
        bincenter_h.resize((bincenter_h.size,1))
        bincenter_w.resize((bincenter_w.size,1))
        dist_ph = abs(sample_ph - bincenter_h)
        dist_pw = abs(sample_pw - bincenter_w)
        weights_h = dist_ph / sample_res
        weights_w = dist_pw / sample_res
        weights_h = (1-weights_h) * (weights_h <= 1)
        weights_w = (1-weights_w) * (weights_w <= 1)
        # weights is the contribution of each pixel to the corresponding bin
            center
        self.weights = weights_h * weights_w
        #pyplot.imshow(self.weights)
        #pyplot.show()

    def process_image(self, image, positionNormalize = True, verbose = False)
        :
        '''
        processes a single image, return the locations and the values of
            detected SIFT features.
        image: a M*N image which is a numpy 2D array. Color images will
            automatically be converted to grayscale.
        positionNormalize: whether to normalize the positions to [0,1]. If
            False, the pixel-based positions of the
            top-right position of the patches is returned.
```

```
    Return values:
    feaArr: the feature array, each row is a feature positions: the
        positions of the features
    '''

    image = image.astype(np.double)
    if image.ndim == 3:
        # we do not deal with color images.
        image = np.mean(image,axis=2)
    # compute the grids
    H,W = image.shape
    gS = self.gS
    pS = self.pS
    remH = np.mod(H-pS, gS)
    remW = np.mod(W-pS, gS)
    offsetH = int(remH/2)
    offsetW = int(remW/2)
    gridH,gridW = np.meshgrid(range(offsetH,H-pS+1,gS), range(offsetW,W-
        pS+1,gS))
    gridH = gridH.flatten()
    gridW = gridW.flatten()
    if verbose:
        print('Image: w {}, h {}, gs {}, ps {}, nFea {}'.format(W,H,gS,pS
            ,gridH.size))
    feaArr = self.calculate_sift_grid(image,gridH,gridW)
    feaArr = self.normalize_sift(feaArr)
    if positionNormalize:
        positions = np.vstack((gridH / np.double(H), gridW / np.double(W)
            ))
    else:
        positions = np.vstack((gridH, gridW))
    return feaArr, positions

def calculate_sift_grid(self,image,gridH,gridW):
    '''
    This function calculates the unnormalized sift features
    It is called by process_image().
    '''
    H,W = image.shape
    Npatches = gridH.size
    feaArr = np.zeros((Npatches,NSAMPLES*NANGLES))

    # calculate gradient
    GH,GW = gen_dgauss(self.sigma)
    IH = signal.convolve2d(image,GH,mode='same')
    IW = signal.convolve2d(image,GW,mode='same')
    Imag = np.sqrt(IH**2+IW**2)
    Itheta = np.arctan2(IH,IW)
```

```
        Iorient = np.zeros((NANGLES,H,W))
        for i in range(NANGLES):
            Iorient[i] = Imag * np.maximum(np.cos(Itheta - ANGLES[i])**ALPHA
                ,0)
            #pyplot.imshow(Iorient[i])
            #pyplot.show()
        for i in range(Npatches):
            currFeature = np.zeros((NANGLES,NSAMPLES))
            for j in range(NANGLES):
                currFeature[j] = np.dot(self.weights,\
                        Iorient[j,gridH[i]:gridH[i]+self.pS, gridW[i]:gridW[i]+
                            self.pS].flatten())
            feaArr[i] = currFeature.flatten()
        return feaArr

    def normalize_sift(self,feaArr):
        '''
        This function does sift feature normalization following David Lowe's
            definition
         (normalize length -> thresholding at 0.2 -> renormalize length)
        '''
        siftlen = np.sqrt(np.sum(feaArr**2,axis=1))
        hcontrast = (siftlen >= self.nrml_thres)
        siftlen[siftlen < self.nrml_thres] = self.nrml_thres
        # normalize with contrast thresholding
        feaArr /= siftlen.reshape((siftlen.size,1))
        # suppress large gradients
        feaArr[feaArr>self.sift_thres] = self.sift_thres
        # renormalize high-contrast ones
        feaArr[hcontrast] /= np.sqrt(np.sum(feaArr[hcontrast]**2,axis=1)).\
                reshape((feaArr[hcontrast].shape[0],1))
        return feaArr

def obtain_dense_features(images_names, des):
    """Apply descriptor to each image.

    Parameters
    ----------
    images_names : dict
        Images groupped by class
    des : feature dense descriptor

    Returns
    -------
    descriptor_list: list (len = no. images)
        Raw features of each image.
    labels : nd.array
        True output of each image. Same order than descriptor_list
```

```python
    """

    # ESTADISTICAS ###
    init_time = perf_counter()
    no_images = 0
    _aux_ = list(images_names.keys())[0]
    n_totales = len(images_names.keys()) * len(images_names[_aux_])
    print(f'Computing dense response... 0/{n_totales}')

    descriptor_list = [] # len = no. images
    labels = np.array([]) # labels. Same order than descriptor_list

    for class_name in images_names.keys():
        labels = np.concatenate([labels, np.repeat(int(LABEL_MAPPER[
            class_name]), len(images_names[class_name]))])
        for img_name in images_names[class_name]:
            img = cv2.imread(img_name, 0)
            img = cv2.resize(img, RESIZE_SIZE) # CONSIDERAR
            feaArr, _ = des.process_image(img)
            descriptor_list.append(feaArr)

            # ESTADISTICAS ###
            no_images += 1
            if (no_images % 50) == 0:
                actual_time = perf_counter() - init_time
                print('Computing dense response... %d/%d ( eta: %.1f s )' % (
                    no_images, n_totales, (n_totales - no_images) *
                    actual_time / no_images))

    return descriptor_list, labels

def _vstackDenseFeatures(descriptor_list):
    """ Vstack the list of features by image

    Parameters
    ----------
    descriptor_list : list (len = no. images)
        Raw features of each image.

    Returns
    -------
    descriptors : nd.array
        1 row per feature
    """
    descriptors = np.array(descriptor_list[0])
    for descriptor in descriptor_list[1:]:
        descriptors = np.vstack((descriptors, descriptor))
    return descriptors
```

```
def build_bow(descriptor_list, n_clusters):
    """ Build bow clustering features

    Parameters
    ----------
    descriptor_list : list (len = no. images)
        Raw features of each image.
    n_clusters : int

    Returns
    -------
    kmeans: <class 'sklearn.cluster._kmeans.KMeans'>
        Bag of visual words
    """
    # Vstack descriptor_list
    print("Vstacking results....")
    descriptors = _vstackDenseFeatures(descriptor_list)

    # Clustering to obtain bow
    print("Clustering....")
    kmeans_bow = KMeans(n_clusters=n_clusters).fit(descriptors)
    # Faster computation
    #kmeans_bow = MiniBatchKMeans(n_clusters=n_clusters).fit(descriptors)

    return kmeans_bow


def extractFeatures(kmeans, descriptor_list, labels, no_clusters):
    """ Describe each image by its histogram of visual features ocurrences.

    Parameters
    ----------
    kmeans: <class 'sklearn.cluster._kmeans.KMeans'>
        Bag of visual words
    descriptor_list : list (len = no. images)
        Raw features of each image.
    labels : nd.array
        True output of each image. Same order than descriptor_list
    n_clusters : int

    Returns
    -------
    im_features: np.array
        Matrix of size N x n+1 where:
        - N is the number of patterns, 1 row for each image.
        - n is the number of atributes (computed shape measures). The last
            column codify the class.
```

```
    """
    image_count = len(descriptor_list)
    im_features = np.array([np.zeros(no_clusters+1) for i in range(
        image_count)])
    no_images = 0
    init_time = perf_counter()
    for i in range(image_count):
        predictions = kmeans.predict(descriptor_list[i])
        unique, counts = np.unique(predictions, return_counts=True)
        for idx_, sum_ in zip(unique, counts):
            im_features[i][idx_] += sum_
        # append label
        im_features[i][-1] = labels[i]

        no_images += 1
        if (no_images % 50) == 0:
                actual_time = perf_counter() - init_time
                print('Encoding images... %d/%d ( eta: %.1f s )' % (no_images,
                     image_count, (image_count - no_images) * actual_time /
                    no_images))

    return im_features
```

lbp.py

```
from time import perf_counter
import cv2
from skimage.feature import local_binary_pattern
from config import *
import pdb

def computeLBP(images_names):
    """ Compute LBP descriptor for gray image.

    Parameters
    ----------
    images_names : dict
        Images groupped by class

    Returns
    -------
    feature_matrix_base : np.ndarray
        Matrix of size N x n where:
        - N is the number of patterns, 1 row for each image.
        - n is the number of atributes.
    """

    # ESTADISTICAS ###
```

```
    init_time = perf_counter()
    no_images = 0
    _aux_ = list(images_names.keys())[0]
    n_totales = len(images_names.keys()) * len(images_names[_aux_])
    print(f'Computing lbp... 0/{n_totales}')

    matrix_features = []

    for class_name in images_names.keys():
        for img_name in images_names[class_name]:
            img = cv2.imread(img_name, 0)
            lbp = local_binary_pattern(img, N_POINTS, RADIUS, METHOD)
            # Histogram
            hist, _ = np.histogram(lbp, bins=BINS)
            matrix_features.append(hist)

            # ESTADISTICAS ###
            no_images += 1
            if (no_images % 50) == 0:
                actual_time = perf_counter() - init_time
                print('Computing LBP... %d/%d ( eta: %.1f s )' % (no_images,
                    n_totales, (n_totales - no_images) * actual_time /
                    no_images))
    matrix_features = np.array(matrix_features)
    return matrix_features
```

classifier.py

```
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
import numpy as np
import matplotlib.pyplot as plt
from config import *
from sklearn.svm import LinearSVC
from sklearn.metrics import *
import seaborn as sns


def plotHistogram(im_features, no_clusters):
    x_scalar = np.arange(no_clusters)
    y_scalar = np.array([abs(np.sum(im_features[:,h], dtype=np.int32)) for h
        in range(no_clusters)])

    plt.bar(x_scalar, y_scalar)
    plt.xlabel("Visual Word Index")
    plt.ylabel("Frequency")
    plt.title("Complete Vocabulary Generated")
    plt.xticks(x_scalar + 0.4, x_scalar)
```

```
    plt.savefig(os.path.join(RESULT_PATH, "histogram.png")); plt.clf()

def train_test(train_im_features, val_im_features, test_im_features):
    """Obtain SVC model. Evaluate it

    Parameters
    ----------
    train_im_features : np.ndarray
        Matrix of size N x n+1 where:
        - N is the number of patterns, 1 row for each image.
        - n is the number of atributes (computed shape measures).
          The last column codify the class.r
    val_im_features : same for val images
    test_im_features : same for test images

    Returns
    -------
    """

    y_train = train_im_features[:, -1]
    x_train = train_im_features[:, 0:-1]

    y_val = val_im_features[:, -1]
    x_val = val_im_features[:, 0:-1]

    y_test = test_im_features[:, -1]
    x_test = test_im_features[:, 0:-1]

    # Normalize features
    scale = StandardScaler().fit(x_train)
    x_train = scale.transform(x_train)
    x_val = scale.transform(x_val)
    x_test = scale.transform(x_test)

    plotHistogram(x_train, N_CLUSTERS) # Guardar en vez de plot

    # TRAIN
    # Sintonizacion lambda y sigma
    print("Sintonizacion:")
    vL=2.**np.arange(-5, 16, 2)
    vG=2.**np.arange(-7, 8, 2)

    kappa_sintonizacion=np.zeros((len(vL), len(vG)));
    kappa_mellor=-np.Inf; L_mellor=vL[0]; G_mellor=vG[0]
    print('%10s %15s %10s %10s'%('Lambda','Gamma', 'Kappa (%)', 'Mejor'))

    for i,L in enumerate(vL):
        for j,G in enumerate(vG):
```

```
            modelo=SVC(C=L, kernel ='rbf', gamma=G, verbose=False).fit(
                x_train, y_train)
            z = modelo.predict(x_val)
            kappa = cohen_kappa_score(y_val, z) * 100
            kappa_sintonizacion[i,j] = kappa
            if kappa>kappa_mellor:
                kappa_mellor=kappa; L_mellor=L; G_mellor = G
            print('%.2f %15g %10.1f %10.1f'%(L,G, kappa, kappa_mellor))
    print('L_mejor=%g, G_mejor=%g, kappa=%.2f%%\n'%(L_mellor, G_mellor,
        kappa_mellor))

    # MODELO CON MEJOR PARAMS
    X = np.vstack((x_train, x_val))
    Y = np.concatenate((y_train, y_val))
    model = SVC(C=L_mellor, kernel ='rbf', gamma=G_mellor, verbose=False).fit
        (X,Y)

    z = model.predict(x_test)
    acc = 100 * accuracy_score(y_test, z)
    print(acc)
    cf = confusion_matrix(y_test, z)
    cf_image = sns.heatmap(cf, cmap='Blues', annot=True, fmt='g')
    figure = cf_image.get_figure()
    figure.savefig(os.path.join(RESULT_PATH, "cf.png")); plt.clf()

    return model
```

data.py

```
import os
from config import *
import pdb
from re import search

def find_images(dirpath, label_mapper):
    """Detect image files contained in a folder.

    Parameters
    ----------
    dirpath : string
        Path name of the folder that contains the images.
    label_mapper : dict
        It associates a class, identified by its name before '-' character
        with a particular integer label.

    Returns
    -------
    imgfiles : dict
```

```
        Full path names of all the image files in ‘dirpath‘ (and its
        subfolders) grouped by class name.
    """
    pattern = '_gif'
    images_dictionary = {}
    for class_name in label_mapper.keys():
        images_aux = []
        root_path = os.path.join(dirpath, class_name)
        for img_name in os.listdir(root_path):
            # _gif images tienen problemas al cargarse con openCV
            if search(pattern, img_name):
                continue
            images_aux.append(os.path.join(root_path, img_name))
            # Stop at MAX_IMAGES_PER_CLASS images
            if len(images_aux) == MAX_IMAGES_PER_CLASS:
                break
        images_dictionary[class_name] = images_aux
    return images_dictionary

def train_val_test_split(imgfiles, train_n, val_n, test_n):
    """Train-val-test split.

    Parameters
    ----------
    imgfiles : dict
        Full path names of all the image files in ‘dirpath‘ (and its
        subfolders) grouped by class name.

    Returns
    -------
    {train}{va}{test}_names : dict
    """
    train_names = {}
    val_names = {}
    test_names = {}

    for class_name in imgfiles:
        train_names[class_name] = imgfiles[class_name][:train_n]
        val_names[class_name] = imgfiles[class_name][train_n : train_n +
            val_n]
        test_names[class_name] = imgfiles[class_name][train_n + val_n :
            train_n + val_n + test_n]
    return train_names, val_names, test_names

def get_class(filename, label_mapper):
    """Extract the class integer label from the path of the image.

    Parameters
```

```
    ----------
    filename : string
        Filename (including path) of a shape sample.
    label_mapper : dict

    Returns
    -------
    class_name : integer
        Class integer to which the shape sample belongs.

    """
    label = os.path.split(os.path.split(filename)[0])[1]
    return label_mapper[label]
```

References

[1] A. Quattoni, and A.Torralba. Recognizing Indoor Scenes. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2009.

[2] Yangqing Jia. Dense SIFT implementation. Available in `https://github.com/Yangqing/dsift-python/blob/master/dsift.py` at May 14, 2022.