

## Scene Image Recognition

- a. Obtain the 15-category scenes dataset available here ([https://www.dropbox.com/s/b5rfwhezv3o6ouo/scene\\_categories.zip?dl=0](https://www.dropbox.com/s/b5rfwhezv3o6ouo/scene_categories.zip?dl=0)). The dataset consists of a mix of 15 outdoor (urban, suburban, natural) and indoor categories. Each category has between 200 to 400 images, and the average image size is  $300 \times 250$  pixels. Select the first 100 images in each category for training, the next 20 for testing, and discard the rest (after using the first 100 for training, you may choose to use all the rest for testing if your machine can handle the computation).

The first step starts with the splitting of dataset into two part i.e. testing and training. The files are not loaded at once but their paths are stored as reference to the images so they can be easily accessed, and less computation would be used.

A base directory is the one where all the scenes area categorized. We are taking first 100 images as training images and next 20 for the testing purpose, while discarding the rest of images present in each scene category.

The path is returned for both, train and testing images and labels.

```
def get_image_paths(base_dir, num_train_per_cat, num_test_per_cat):
    categories = os.listdir(base_dir)
    train_image_paths = []
    test_image_paths = []

    train_labels = []
    test_labels = []

    for category in categories:

        image_paths = glob(os.path.join(base_dir, category, '*.jpg'))
        for i in range(num_train_per_cat):
            train_image_paths.append(image_paths[i])
            train_labels.append(category)

        image_paths = glob(os.path.join(base_dir, category, '*.jpg'))
        for i in range(num_train_per_cat, num_train_per_cat + num_test_per_cat):
            test_image_paths.append(image_paths[i])
            test_labels.append(category)

    return train_image_paths, test_image_paths, train_labels, test_labels
```

- b. Next, you need to write some code to densely sample features (SIFT or any other descriptor) from each image. This can be done by extracting all possible patches of a suitable size, e.g. 16x16 pixels on a regular grid with a spacing of, say, 8 pixels. For each patch, a 128 dimensional descriptor may be computed using the VLFeat Library (<http://www.vlfeat.org/>) if you're on MATLAB, or OpenCV if using Python. You may even just make use of the dense SIFT extraction function(s) in VLFeat.

Next step comes to apply the Dense-SIFT on the images. For this purpose an open source library is being used which has the function to implement dense sift on the images and it returns us the frames, which are the key points, and descriptors for each image. While applying dsift we have given it the step size of 8 pixel and patch size of 16x16.

```
def D_SIFT(image_paths):
    bag_of_features = []
    for path in image_paths:
        img = np.asarray(Image.open(path), dtype='float32')
        frames, descriptors = dsift(img, step=8, size=[16,16], fast=True)
        bag_of_features.append(descriptors)
    bag_of_features = np.concatenate(bag_of_features, axis=0).astype('float32')
    return bag_of_features
```

- c. Learn a visual vocabulary for the scenes dataset. This can be done via k-means clustering performed on a few thousand SIFT descriptors randomly chosen in equal number from each training image in the dataset. You should use library functionality (MATLAB / OpenCV) for the clustering. Having all SIFT descriptors of all training images loaded up at the same time in memory is not feasible, hence the best way is to load each training image one at a time, perform a dense description and randomly choose a fixed number of features that should be retained in memory. Once all the images have been processed, you should have all the samples in memory upon which clustering may be carried out. Save the learned dictionary to disk (for safe keeping). You should report your classification results [see task (f) below] for dictionaries of sizes 50 and 200 (you may alternatively work with larger dictionary sizes e.g., 200 and 1000, if your machine can handle the computation, but the report should contain comparison between two different dictionary sizes).

The kmeans function used here is from the cvlfeat library (open source: github). It takes features and visual vocabulary size along with the initialization set as "PLUSPLUS". This initialization defines the initial clusters centers for k-means clustering. It minimizes the intra-class variance i.e. the sum of squared distances from each data point being clustered to its cluster center choosing a center that is closest to it.

This functions returns us a visual vocab of desired size.

```
def visual_vocab(bag_of_features, vocab_size):  
    vocab = kmeans(bag_of_features, vocab_size, initialization="PLUSPLUS")  
    return vocab
```

- d. Based on the learned visual vocabulary, construct a bag of words representation for every training and testing image.

Based on the learned visual vocabulary next step is constructing bag of words representation. For this purpose, we load the trained visual vocabulary into the memory and for each image in path, find its descriptors using dsift function. To compute the distance, we use cdist which takes the two points to compute distance along and metric, which in our case is Euclidean. Now as our requirement is choose the index with minimum distance so we find the min value among distances and allocate its index to our index variable.

Next, we compute the histogram using the built-in function of NumPy, and lastly its norm is computed for each value. This results in image features which are returned as the function terminates.

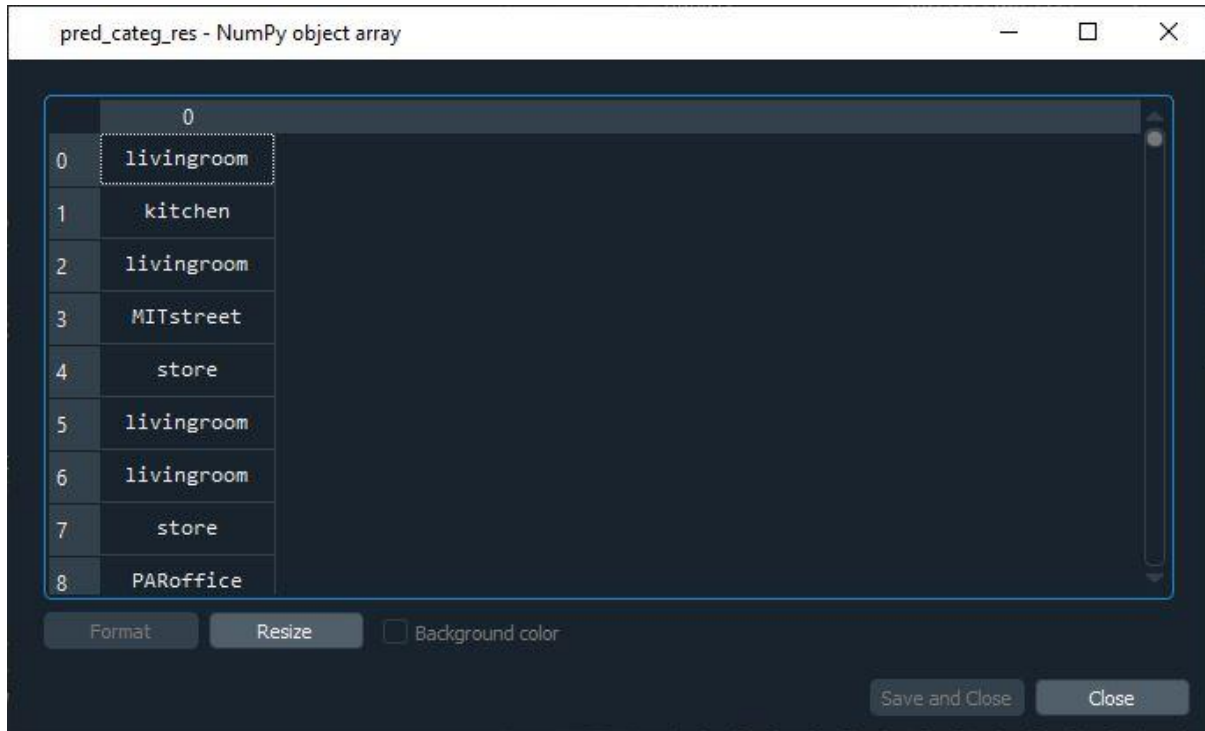
```
## Now from vocab  
def get_bags_of_word_rep(image_paths):  
    with open('vocab_train.pkl', 'rb') as handle:  
        vocab = pickle.load(handle)  
        image_feats = []  
        for path in image_paths:  
            img = np.asarray(Image.open(path), dtype='float32')  
            frames, descriptors = dsift(img, step=8, size=[16,16], fast=True)  
            dist = distance.cdist(vocab, descriptors, metric='euclidean')  
            idx = np.argmin(dist, axis=0)  
            hist, bin_edges = np.histogram(idx, bins=len(vocab))  
            hist_norm = [float(i)/sum(hist) for i in hist]  
  
            image_feats.append(hist_norm)  
        image_feats = np.asarray(image_feats)  
        return image_feats
```

- e. Train a linear SVM classifier based on the training images. Classify the test images. Try some other kernels (e.g., Gaussian, RBF, etc.). Use library functionality [MATLAB / OpenCV / VLFeat / LibSVM (<https://www.csie.ntu.edu.tw/~cjlin/libsvm/>)] for SVM classification.

From sklearn library we use the SVC for classification on training images. The SVC was trained on train image features and training



images labels. And for the case of prediction of labels pred\_labels were returned.



	0
0	livingroom
1	kitchen
2	livingroom
3	MITstreet
4	store
5	livingroom
6	livingroom
7	store
8	PARoffice

```
def svm_classify(train_image_feats, train_labels, test_image_feats):
    SVM = make_pipeline(StandardScaler(), SVC(C=1.0, kernel='rbf', degree=3,
        gamma='scale', coef0=0.0, shrinking=True,
        probability=False, tol=0.001, cache_size=200,
        class_weight=None, verbose=False, max_iter=1,
        decision_function_shape='ovr', break_ties=False,
        random_state=None))
    SVM.fit(train_image_feats, train_labels)
    pred_label = SVM.predict(test_image_feats)
    return pred_label
```

- f. Provide the confusion matrix ([https://en.wikipedia.org/wiki/Confusion\\_matrix](https://en.wikipedia.org/wiki/Confusion_matrix)) for your classification and the overall accuracy.

Lastly for the purpose of computing and plotting confusion matrix we first compute the confusion matrix using Keras built-in library for confusion matrix. It results in a matrix of size; label\_size x label\_size.

```
def plot_confusion_matrix(cm, category, title='Confusion matrix', cmap=plt.cm.rainbow):
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(category))

    plt.xticks(tick_marks, category, rotation=45)
    plt.yticks(tick_marks, category)
    plt.tight_layout()
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
```

## **Main Function**

```
base_dir="C:/EME/8th sem/CV/assignment/#3/scenes/"
train_image_paths, test_image_paths, train_labels, test_labels=get_image_paths(base_dir,100,20)
b_o_features=D_SIFT(train_image_paths)
vocab_size=200
vocab_train=visual_vocab(b_o_features, vocab_size)
with open('vocab_train.pkl', 'wb') as handle:
    pickle.dump(vocab_train, handle)
bag_of_words_train=get_bags_of_word_rep(train_image_paths)
bag_of_words_test= get_bags_of_word_rep(test_image_paths);
pred_categ_res = svm_classify(bag_of_words_train, train_labels, bag_of_words_test)

cf_matrix=build_confusion_mtx(test_labels, pred_categ_res)

print(cf_matrix)
```

## **Parameterization:**

### **Dataset Splitting**

Total	Testing	Training
400	20	100

### **Kernel**

The kernel chosen for SVC was 'rbf'

### **Visual Vocab Size**

The code was tested on two different visual vocab size i.e. 200 and 50.

### **Patch Size**

The patch size was kept 16 x 16 pixels

### **Grid Size**

Grid spacing was set to be 8

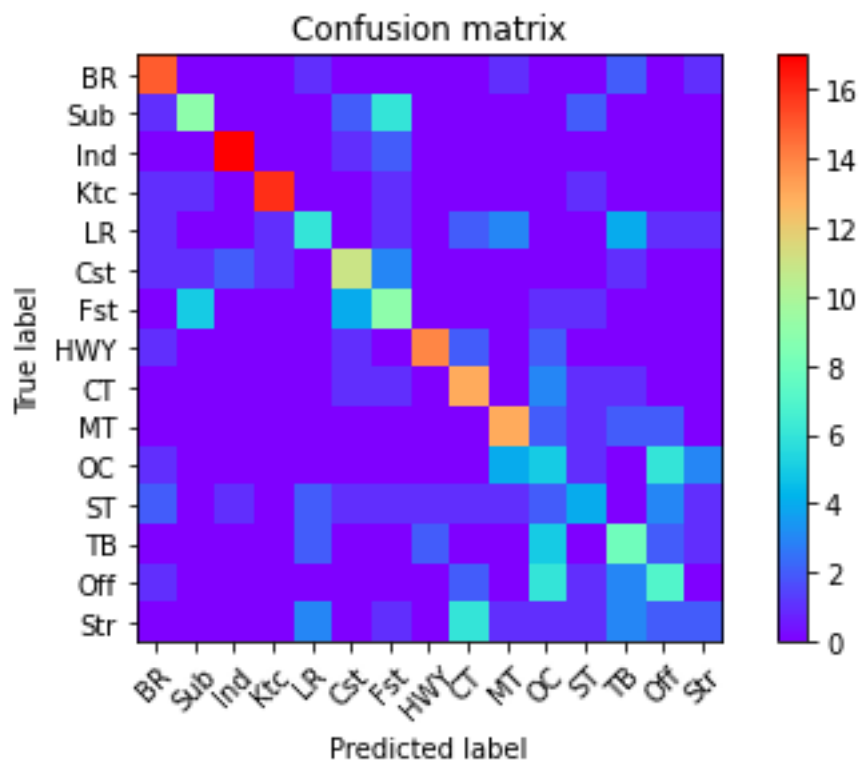
## Results

Both the results have been displayed for vocab size 50 and 200.

### Incase of 200 size vocab:

```
[[15 0 0 0 1 0 0 0 0 1 0 0 2 0 1]
 [1 9 0 0 0 2 6 0 0 0 0 2 0 0 0]
 [0 0 17 0 0 1 2 0 0 0 0 0 0 0 0]
 [1 1 0 16 0 0 1 0 0 0 0 1 0 0 0]
 [1 0 0 1 6 0 1 0 2 3 0 0 4 1 1]
 [1 1 2 1 0 11 3 0 0 0 0 0 1 0 0]
 [0 5 0 0 0 4 9 0 0 0 1 1 0 0 0]
 [1 0 0 0 0 1 0 14 2 0 2 0 0 0 0]
 [0 0 0 0 0 1 1 0 13 0 3 1 1 0 0]
 [0 0 0 0 0 0 0 0 0 13 2 1 2 2 0]
 [1 0 0 0 0 0 0 0 0 4 5 1 0 6 3]
 [2 0 1 0 2 1 1 1 1 1 2 4 0 3 1]
 [0 0 0 0 2 0 0 2 0 0 5 0 8 2 1]
 [1 0 0 0 0 0 0 0 2 0 6 1 3 7 0]
 [0 0 0 0 3 0 1 0 6 1 1 1 3 2 2]]
```

## Matrix plot



## Incase of 50 vocab size:

```
[[18 0 0 0 0 0 0 0 0 0 1 0 1 0 0]
 [ 1 9 0 0 0 2 7 0 0 0 1 0 0 0 0]
 [ 0 0 15 0 0 3 2 0 0 0 0 0 0 0 0]
 [ 0 2 0 18 0 0 0 0 0 0 0 0 0 0 0]
 [ 1 1 0 0 3 0 0 1 1 1 1 1 10 0 0]
 [ 1 1 1 0 0 13 2 0 0 0 0 0 0 0 2]
 [ 0 5 0 2 0 2 9 0 0 0 0 2 0 0 0]
 [ 0 0 0 0 0 0 0 14 1 0 0 2 0 1 2]
 [ 1 0 0 0 0 0 0 0 11 0 3 3 2 0 0]
 [ 0 0 0 0 0 0 0 0 1 11 1 0 3 4 0]
 [ 1 0 0 0 1 0 0 2 0 2 1 0 1 10 2]
 [ 1 0 0 0 2 0 1 0 1 0 1 8 1 2 3]
 [ 0 0 0 0 3 0 0 0 0 0 1 1 7 7 1]
 [ 1 0 0 0 0 0 0 0 2 0 1 2 4 9 1]
 [ 0 0 1 0 2 0 0 1 4 0 0 0 1 3 8]]
```

## Matrix Plot

