CSCI 5561: Assignment #3

Scene Recognition

1 Submission

- Assignment due: Nov 1 (11:55pm)
- Individual assignment
- Submission through Canvas.
- You will complete scene_recognition.py that contains the following functions:
  - get_tinyImage
  - predict_knn
  - classify_knn_tiny
  - compute_dsift
  - build_visual_dictionary
  - compute_bow
  - classify_knn_bow
  - predict_svm
  - classify_svm_bow

The code can be downloaded from https://www-users.cs.umn.edu/~hspark/csci5561_F2019/scene_recognition.py

- DO NOT SUBMIT THE PROVIDED IMAGE DATA
- The function that does not comply with its specification will not be graded.
- The code must be run with Python 3 interpreter.
- You are not allowed to use computer vision related package functions unless explicitly mentioned here. Please consult with TA if you are not sure about the list of allowed functions.
- Place code and a two-page summary write-up with resulting visualization (in pdf format; more than 2 pages will be automatically returned.) into a folder, compress it, and submit.
2 Overview

The goal of this assignment is to build a set of visual recognition systems that classify the scene categories. The scene classification dataset consists of 15 scene categories including office, kitchen, and forest as shown in Figure 1. The system will compute a set of image representations (tiny image and bag-of-word visual vocabulary) and predict the category of each testing image using the classifiers (k-nearest neighbor and SVM) built on the training data. A simple pseudo-code of the recognition system can be found below:

**Algorithm 1 Scene Recognition**

1. Load training and testing images
2. Build image representation
3. Train a classifier using the representations of the training images
4. Classify the testing data.
5. Compute accuracy of testing data classification.

For the k-nn classifier, step 3 and 4 can be combined.
3 Scene Classification Dataset

You can download the training and testing data from here:

The data folder includes two text files (train.txt and test.txt) and two folders (train and test). Each row in the text file specifies the image and its label, i.e., (label) (image_path). The text files can be used to load images. In each folder, it includes 15 classes (Kitchen, Store, Bedroom, LivingRoom, Office, Industrial, Suburb, InsideCity, TallBuilding, Street, Highway, OpenCountry, Coast, Mountain, Forest) of scene images.

Note: the image paths inside train.txt and test.txt were recorded in Windows format (use \ instead of /). You may need to use function Path and PureWindowsPath imported from pathlib to deal with that if you use Linux or Mac. But do not worry about it since we have provided a function called extract_dataset_info, which can read information from those two txt files for you.
4 Tiny Image KNN Classification

Figure 2: You will use tiny image representation to get an image feature.

```python
def get_tiny_image(img, output_size):
    ...
    return feature
```

**Input:** `img` is an gray scale image, `output_size=(w, h)` is the size of the tiny image.

**Output:** `feature` is the tiny image representation by vectorizing the pixel intensity. The resulting size will be `w×h`.

**Description:** You will simply resize each image to a small, fixed resolution (e.g., 16×16). You need to normalize the image by having zero mean and unit length. This is not a particularly good representation, because it discards all of the high frequency image content and is not especially invariant to spatial or brightness shifts.

```python
def predict_knn(feature_train, label_train, feature_test, k):
    ...
    return label_test_pred
```

**Input:** `feature_train` is a `n_{tr}×d` matrix where `n_{tr}` is the number of training data samples and `d` is the dimension of image feature, e.g., 265 for 16×16 tiny image representation. Each row is the image feature. `label_train∈[1,15]` is a `n_{tr}` vector that specifies the label of the training data. `feature_test` is a `n_{te}×d` matrix that contains the testing features where `n_{te}` is the number of testing data samples. `k` is the number of neighbors for label prediction.

**Output:** `label_test_pred` is a `n_{te}` vector that specifies the predicted label for the testing data.

**Description:** You will use a k-nearest neighbor classifier to predict the label of the testing data.
def classify_knn_tiny(label_classes, label_train_list, img_train_list, label_test_list, img_test_list):
    ...
    return confusion, accuracy

Input: label_classes is a list of all kinds of classes, img_train_list and img_test_list are lists of paths to training and test images, label_train_list and label_test_list are corresponding lists of image scene labels.

Output: confusion is a 15 × 15 confusion matrix and accuracy is the accuracy of the testing data prediction.

Description: You will combine get_tiny_image and predict_knn for scene classification. Your goal is to achieve the accuracy >18%.

Note: We have provided a function called extract_dataset_info which takes in path to dataset directory and output label_classes, label_train_list, img_train_list, label_test_list, img_test_list for you(those will be the input arguments to function classify_knn_bow and classify_svm_bow as well). To make your life and ours easier, please make sure you use that function.
def compute_dsift(img, stride, size):
    ...
    return dense_feature
Input: img is a gray scale image. stride and size are both integers controls locations on image to compute sift features and diameter of the meaningful keypoint neighborhood.
Output: dense_feature is a collection of sift features whose size is $n \times 128$. $n$ is total number of locations to compute sift features on img.
Description: Given an image, instead of detecting key points and computing sift descriptor, this function directly compute sift descriptor on a dense set of locations on image. You can use sift related functions from opencv for computing sift descriptor for each location.

def build_visual_dictionary(dense_feature_list, dic_size):
    ...
    return vocab
Input: dense_feature_list is a list of dense sift feature representation of training images (each image is represented as a $n \times 128$ array) and dic_size is the size of the dictionary (the number of visual words). Function compute_dsift is provided to extract dense sift features from an image.
Output: vocab lists the quantized visual words whose size is $dic\_size \times 128$.
Description: Given a list of dense sift feature representation of training images, you will build a visual dictionary made of quantized SIFT features. You may start
dic_size=50. You can use KMeans function imported from sklearn.cluster. You may visualize the image patches to make sense the clustering as shown in Figure 4.

**Algorithm 2 Visual Dictionary Building**

1: For each image, compute dense SIFT over regular grid
2: Build a pool of SIFT features from all training images
3: Find cluster centers from the SIFT pool using kmeans algorithms.
4: Return the cluster centers.

**Note:** It takes more than half hour to build bag-of-word visual vocabulary, if you use default parameters of KMeans function(n_init=10, max_iter=300). You may want to play around with those parameter and use np.savetxt to save current vocab if you think it is good. Then you can use np.loadtxt to load that saved vocab in the future to save your time.

```python
def compute_bow(feature, vocab):
    ...
    return bow_feature
```

**Input:** feature is a set of SIFT features for one image, and vocab is visual dictionary. **Output:** bow_feature is the bag-of-words feature vector whose size is dic_size. **Description:** Give a set of SIFT features from an image, you will compute the bag-of-words feature. The BoW feature is constructed by counting SIFT features that fall into each cluster of the vocabulary. Nearest neighbor can be used to find the closest cluster center. The histogram needs to be normalized such that BoW feature has a unit length.

Figure 5: Confusion matrix for BoW+KNN.
def classify_knn_bow(label_classes, label_train_list, img_train_list, label_test_list, img_test_list):
    ...
    return confusion, accuracy

Input: refer to function classify_knn_bow
Output: confusion is a $15 \times 15$ confusion matrix and accuracy is the accuracy of the testing data prediction.
Description: Given BoW features, you will combine build_visual_dictionary, compute_bow, and predict_knn for scene classification. Your goal is to achieve the accuracy $>50\%$. 
6 BoW+SVM

```python
def predict_svm(feature_train, label_train, feature_test):
    ...
    return label_test_pred
```

**Input:** `feature_train` is a $n_{tr} \times d$ matrix where $n_{tr}$ is the number of training data samples and $d$ is the dimension of image feature. Each row is the image feature. `label_train` ∈ $[1, 15]$ is a $n_{tr}$ vector that specifies the label of the training data. `feature_test` is a $n_{te} \times d$ matrix that contains the testing features where $n_{te}$ is the number of testing data samples.

**Output:** `label_test_pred` is a $n_{te}$ vector that specifies the predicted label for the testing data.

**Description:** You will use a SVM classifier to predict the label of the testing data. You don’t have to implement the SVM classifier. Instead, you can use e.g. function `LinearSVC` or `SVC` imported from `sklearn.svm`. Linear classifiers are inherently binary and we have a 15-way classification problem. To decide which of 15 categories a test case belongs to, you will train 15 binary, 1-vs-all SVMs. 1-vs-all means that each classifier will be trained to recognize ‘forest’ vs ‘non-forest’, ‘kitchen’ vs ‘non-kitchen’, etc. All 15 classifiers will be evaluated on each test case and the classifier which is most confidently positive “wins”. For instance, if the ‘kitchen’ classifier returns a score of -0.2 (where 0 is on the decision boundary), and the ‘forest’ classifier returns a score of -0.3, and all of the other classifiers are even more negative, the test case would be classified as a kitchen even though none of the classifiers put the test case on the positive side of the decision boundary. When learning an SVM, you have a free parameter 'lambda' (argument `C` in function `LinearSVC` and `SVC`) which controls how strongly regularized the model is. Your accuracy will be very sensitive to lambda, so be sure to test many values.

**Note:** `LinearSVC` and `SVC` can do multi-class classification if your input labels has more than 2 classes. However, you should NOT take advantage of that. Instead, you should create binary labels for each of those 15 binary 1-vs-all SVMs.
def classify_svm_bow(label_classes, label_train_list,
    img_train_list, label_test_list, img_test_list):
    ...
    return confusion, accuracy

Input: refer to function classify_knn_bow
Output: confusion is a $15 \times 15$ confusion matrix and accuracy is the accuracy of the testing data prediction.
Description: Given BoW features, you will combine build_visual_dictionary, compute_bow, predict_svm for scene classification. Your goal is to achieve the accuracy >60%.
References