

电子科技大学

实验报告

学生姓名：郭宇航 学号：2016100104014 指导教师：徐行

一、实验项目名称：基于词袋模型的图像分类

二、实验原理：

BOW(Bag of words)模型最初被应用于文本分类中，将文档表示为特征矢量。它的基本思想就是假定对于一个文本，忽略其语序，语法，句法等等。仅仅看做词汇的集合，而文本中的词汇都是独立的。简单来说就是每篇文档都可以看做一个袋子里面装的是词汇。词袋模型由此得名。举个例子，假设我们有两个文档：

文档一：Bob likes to play basketball, Jim likes too.

文档二：Bob also likes to play football games.

基于这两个文本文档，构造一个词典：

Dictionary =

{1: "Bob", 2. "like", 3. "to", 4. "play", 5. "basketball", 6. "also", 7. "football", 8. "games", 9. "Jim", 10. "too" }。

这个词典一共包含 10 个不同的单词，利用词典的索引号，上面两个文档每一个都可以用一个 10 维向量表示（用整数数字 0~n（n 为正整数）表示某个单词在文档中出现的次数）：

1: [1, 2, 1, 1, 1, 0, 0, 0, 1, 1]

2: [1, 1, 1, 1, 0, 1, 1, 1, 0, 0]

向量中每个元素表示词典中相关元素在文档中出现的次数(下文中，将用单词的直方图表示)。不过，在构造文档向量的过程中可以看到，我们并没有表达单词在原来句子中出现的次序（这是本 Bag-of-words 模型的缺点之一，不过瑕不掩瑜甚至在此处无关紧要）。

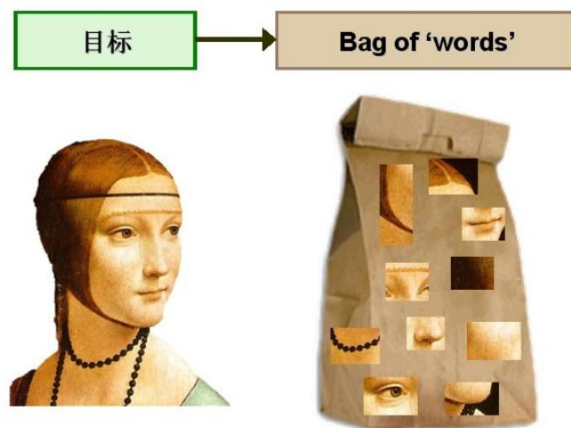


图 1：将词袋模型应用于图像分类

SIFT 特征虽然也可以描述一幅图像但是每个 SIFT 特征的维度都是 128 维，而且一幅图像的 SIFT 特征往往包含了成百上千个 SIFT 特征矢量，进行相似度的计算时计算量非常庞大。通常的做法是使用聚类算法对这些矢量进行聚类，然后用聚类的每个簇代表 BOW 模型的一个视觉词，将一幅图像的 SIFT 矢量映射到视觉词序列生成码本，从而使得图像可以通过码本矢量来表示，计算效率大大提高。

构建 BOW 模型码本的步骤如下：

- (1) 假设训练集共有 M 张图片，对训练图像进行预处理包括图像的增强，分割，图像格式的统一化等等。
- (2) 提取 SIFT 特征，对每一张图片提取 SIFT 特征，每个图像提取的 SIFT 特征数量不定。每个 SIFT 特征用一个 128 维的描述子矢量表示，假设 M 张图片共有 N 个 SIFT 特征。
- (3) 使用 K-means 算法对 (2) 中提取出来的 N 个 SIFT 特征进行聚类将 N 个向量划分为 K 个簇，码本的长度也就为 K ，计算每一张图片的 SIFT 特征到这个码本(dict)的距离并将其映射到最近的视觉词中，词频+1，完成这一步后每一张图片就成为了一个与视觉词序列相对应的词频矢量。

三、实验目的：

图像识别：特征提取+分类器的构建和使用

四、实验内容：

主要任务：

- (1) 图像词袋特征的表示：tiny 特征和 SIFT 特征
- (2) 词袋的构建：从训练数据中获取特征进行 K-means 聚类
- (3) 分类器的构建：KNN 方法（欧氏距离相似度计算归类）和 SVM（一对多策略）

需要完善的代码：

- (1) get_tiny_images()
- (2) Bulid_vocabulary() 其中包括 SIFT 特征的提取与词袋模型的建立
- (3) get_bags_of_words() 根据词袋获取测试图片的直方图特征表示
- (4) SVM_classify()和 Nearest_neighbor_classify()分类器函数的完善

五、实验步骤：

一、完善 get_tiny_images()函数：

完善代码如下：

```
def get_tiny_images(image_paths):  
    ...  
  
    This feature is inspired by the simple tiny images used as features in  
    80 million tiny images: a large dataset for non-parametric object and
```

scene recognition. A. Torralba, R. Fergus, W. T. Freeman. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol.30(11), pp. 1958-1970, 2008. <http://groups.csail.mit.edu/vision/TinyImages/>

Inputs:

`image_paths`: a 1-D Python list of strings. Each string is a complete path to an image on the filesystem.

共有 D 张图片，先将图片读入进来然后 `resize` 为行向量

Outputs:

An $n \times d$ numpy array where n is the number of images and d is the length of the tiny image representation vector. e.g. if the images are resized to 16×16 , then d is $16 * 16 = 256$.

To build a tiny image feature, resize the original image to a very small square resolution (e.g. 16×16). You can either resize the images to square while ignoring their aspect ratio, or you can crop the images into squares first and then resize evenly. Normalizing these tiny images will increase performance modestly.

As you may recall from class, naively downsizing an image can cause aliasing artifacts that may throw off your comparisons. See the docs for `skimage.transform.resize` for details:

<http://scikit-image.org/docs/dev/api/skimage.transform.html#skimage.transform.resize>

Suggested functions: `skimage.transform.resize`, `skimage.color.rgb2grey`,
`skimage.io.imread`, `np.reshape`
...

#TODO: Implement this function!

`dimension = 9`

#`dimension` 表示这里将所有的图像都 `resize` 为一个 9×9 的小图像

#构造输出的矩阵 $n \times d$

`image_tiny = np.empty((len(image_paths), dimension**2), dtype=float)`

for `i` **in** `range(len(image_paths))`:

 #读取图片数据

`image = cv2.imread(image_paths[i])`

 #转为灰度图

`image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)`

 #先进行 crop 操作

`min_dimension = np.min(image.shape)`

 #将图片规范为长宽一致

`image = image[:min_dimension, :min_dimension]`

```

image = cv2.resize(image,(dimension,dimension))
#将图片的像素点归一化
image = image/np.sum(image)
image_tiny[i,:] = np.resize(image,[dimension,dimension])
print(image_tiny.shape)
return image_tiny

```

get_tiny_images()主要目的是对图像进行简单的处理，其中包括：图像的剪裁，下采样缩放，图像的像素点归一化等等。

这里的输入参数 image_paths 是所有训练图片的路径，通过 for 循环进行遍历读取，image_tiny 用于存放已经得到的 tiny 的图像，另外其中调用了 opencv-python 中的大量 API 函数，包括 imread()以及 cvtColor()等。流程大致为：首先读入图片，对图片进行灰度化，然后进行 crop 剪裁图片，剪裁后缩放 resize 到固定的尺寸，这里给的尺寸为 9*9.最后再归一化处理，并 reshape 为行向量。

二、完善 build_vocabulary()函数：

完善代码如下：

```

def build_vocabulary(image_paths, vocab_size):
    """
    This function should sample HOG descriptors from the training images,
    cluster them with kmeans, and then return the cluster centers.

    Inputs:
        image_paths: a Python list of image path strings
        vocab_size: an integer indicating the number of words desired for
the
                    bag of words vocab set

    Outputs:
        a vocab_size x (z*z*9) (see below) array which contains the cluster
        centers that result from the K Means clustering.

    You'll need to generate HOG features using the skimage.feature.hog()
function.
    The documentation is available here:

http://scikit-image.org/docs/dev/api/skimage.feature.html#skimage.feature.hog

    However, the documentation is a bit confusing, so we will highlight some
    important arguments to consider:
        cells_per_block: The hog function breaks the image into evenly-sized
        blocks, which are further broken down into cells, each made of
        pixels_per_cell pixels (see below). Setting this parameter tells
the

```

function how many cells to include in each block. This is a tuple of width and height. Your SIFT implementation, which had a total of 16 cells, was equivalent to setting this argument to (4,4).
pixels_per_cell: This controls the width and height of each cell (in pixels). Like cells_per_block, it is a tuple. In your SIFT implementation, each cell was 4 pixels by 4 pixels, so (4,4).
feature_vector: This argument is a boolean which tells the function what shape it should use for the return array. When set to True, it returns one long array. We recommend setting it to True and reshaping the result rather than working with the default value, as it is very confusing.

It is up to you to choose your cells per block and pixels per cell. Choose values that generate reasonably-sized feature vectors and produce good classification results. For each cell, HOG produces a histogram (feature vector) of length 9. We want one feature vector per block. To do this we can append the histograms for each cell together. Let's say you set cells_per_block = (z,z). This means that the length of your feature vector for the block will be $z*z*9$.

With feature_vector=True, hog() will return one long np array containing every cell histogram concatenated end to end. We want to break this up into a list of $(z*z*9)$ block feature vectors. We can do this using a really nifty numpy function. When using np.reshape, you can set the length of one dimension to -1, which tells numpy to make this dimension as big as it needs to be to accomodate to reshape all of the data based on the other dimensions. So if we want to break our long np array (long_boi) into rows of $z*z*9$ feature vectors we can use `small_bois = long_boi.reshape(-1, z*z*9)`.

The number of feature vectors that come from this reshape is dependent on the size of the image you give to hog(). It will fit as many blocks as it can on the image. You can choose to resize (or crop) each image to a consistent size (therefore creating the same number of feature vectors per image), or

```

you
    can find feature vectors in the original sized image.

    ONE MORE THING
    If we returned all the features we found as our vocabulary, we would have
an
    absolutely massive vocabulary. That would make matching inefficient AND
    inaccurate! So we use K Means clustering to find a much smaller
(vocab_size)
    number of representative points. We recommend using
sklearn.cluster.KMeans
    to do this. Note that this can take a VERY LONG TIME to complete (upwards
    of ten minutes for large numbers of features and large max_iter), so set
    the max_iter argument to something low (we used 100) and be patient. You
    may also find success setting the "tol" argument (see documentation for
    details)
    '''

    #TODO: Implement this function!
    SIFT_features = []
    sift_model = cv2.xfeatures2d.SIFT_create()
    for i in range(len(image_paths)):
        image = cv2.imread(image_paths[i])
        image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        location, sift_features =
sift_model.detectAndCompute(image_gray, None)
        #location 表示的是 sift 特征点的数目, sift_features 是一个矩阵: X*128
        for j in sift_features:
            SIFT_features.append(j)
        SIFT_features = np.array(SIFT_features)
    kmeans_model =
KMeans(n_clusters=vocab_size, max_iter=100).fit(SIFT_features)
    #使用 Kmeans 得到的聚类的簇的中心为词袋模型中的词
    cluster_centers = kmeans_model.cluster_centers_
    return np.array(cluster_centers)

```

这一部分的代码主要流程有两项，第一项是获取每个图片的 SIFT 特征，这里使用的是 opencv-python 中的函数 `xfeatures2d.SIFT_create()`，这一函数有两个返回值，第一个是 `location` 即 SIFT 特征的位置，第二个是 SIFT 特征矢量，SIFT 特征矢量为一个 $N \times 128$ 的矩阵，其中 N 的值每一张图片都不相同；第二项是对所有图像的 SIFT 特征进行 K-means 聚类，首先先将所有的 SIFT 特征组成一个大矩阵，矩阵的行为所有的 SIFT 矢量的数量，列数为 128。通过 K-means 聚类可以得到最终我们需要的所有簇的中心向量，这些中心向量就会作为最终的字典 dict。

三、完善 get_bags_of_words()函数:

完善代码如下:

```
def get_bags_of_words(image_paths):  
    '''  
    This function should take in a list of image paths and calculate a bag  
of  
    words histogram for each image, then return those histograms in an array.  
  
    Inputs:  
        image_paths: A Python list of strings, where each string is a complete  
            path to one image on the disk.  
  
    Outputs:  
        An nxd numpy matrix, where n is the number of images in image_paths  
and  
        d is size of the histogram built for each image.  
  
    Use the same hog function to extract feature vectors as before (see  
    build_vocabulary). It is important that you use the same hog settings  
for  
    both build_vocabulary and get_bags_of_words! Otherwise, you will end up  
    with different feature representations between your vocab and your test  
    images, and you won't be able to match anything at all!  
  
    After getting the feature vectors for an image, you will build up a  
    histogram that represents what words are contained within the image.  
    For each feature, find the closest vocab word, then add 1 to the histogram  
    at the index of that word. For example, if the closest vector in the vocab  
    is the 103rd word, then you should add 1 to the 103rd histogram bin. Your  
    histogram should have as many bins as there are vocabulary words.  
  
    Suggested functions: scipy.spatial.distance.cdist, np.argsort,  
        np.linalg.norm, skimage.feature.hog  
    '''  
    vocab = np.load('vocab.npy')  
    print('Loaded vocab from file.')  
    sift_model = cv2.xfeatures2d.SIFT_create()  
    hist_vector = np.zeros([len(image_paths),len(vocab)])  
    for i in range(len(image_paths)):  
        #读取图片并进行灰度化处理  
        image = cv2.imread(image_paths[i])  
        image_gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)  
        location,sift_features =  
sift_model.detectAndCompute(image_gray,None)
```

```

        #得到每个图片的 sift 特征,下面计算每个图像的 hist 直方图统计距离选择最近
        的一个计数+1
        for j in range(len(location)):
            distance =
cdist(np.array([sift_features[j]]),vocab,metric='euclidean')
            index = np.argsort(distance)[0][0]
            hist_vector[i,index] += 1

        normalization = np.linalg.norm(hist_vector[i])
        if normalization !=0 :
            hist_vector[i] = hist_vector[i]/normalization
        #print(hist_vector)
        print(hist_vector.shape)
        #TODO: Implement this function!
        return np.array(hist_vector)

```

这一部分主要是基于得到的 SIFT 特征的 dict 得到每一个图像的特征表示。主要流程是首先读取 vocab.npy 得到之前的 dict, 然后重新计算每一张图片的 SIFT 特征向量, 计算其到 dict 每一个的矩阵, 选择距离最小那个, 并对其计数+1, 简而言之就是计算每一张图片的所有 SIFT 特征到 dict 的距离, 选择距离最小的表示次 SIFT 特征属于的类别, 对这一类别的计数器加 1, 最终得到一个图像的特征向量的表示。最后的返回值为所有图片的特征向量组成的矩阵。

四、完善 svm_classify()函数

完善代码如下:

```

def svm_classify(train_image_feats, train_labels, test_image_feats):
    """
    This function will predict a category for every test image by training
    15 many-versus-one linear SVM classifiers on the training data, then
    using those learned classifiers on the testing data.

    Inputs:
        train_image_feats: An nxd numpy array, where n is the number of
training
                                examples, and d is the image descriptor vector size.
        train_labels: An nx1 Python list containing the corresponding ground
                                truth labels for the training data.
        test_image_feats: An mxd numpy array, where m is the number of test
                                images and d is the image descriptor vector size.

    Outputs:
        An mx1 numpy array of strings, where each string is the predicted
label
        for the corresponding image in test_image_feats

```



```

We suggest you look at the sklearn.svm module, including the LinearSVC
class. With the right arguments, you can get a 15-class SVM as described
above in just one call! Be sure to read the documentation carefully.
'''

# TODO: Implement this function!
#训练 SVM 分类器
SVM_classification =
LinearSVC(penalty='l2',multi_class='ovr',random_state=0,max_iter=100)
SVM_classification.fit(train_image_feats,train_labels)
#使用 SVM 分类器得到预测结果
output_labels = []
for i in test_image_feats:
    output_labels.append(SVM_classification.predict(i.reshape(1,-1)))
return np.array(output_labels)

```

这一部分代码比较简单，输入的参数分别为训练数据和训练的标签以及测试集的数据。构造 SVM 的分类器，这里使用到的是 sklearn 这个库，调用的是线性的 LinearSVC 模型，参数选取上采用 L2 的正则化，多分类参数选取'ovr'，最大迭代次数为 100 次。将训练数据及标签带入到模型训练，最后将测试集带入模型得到最后的预测结果，保存为向量并作为函数的返回值。

五、完善 nearest_neighbor_classify()函数：

完善代码如下：

```

def nearest_neighbor_classify(train_image_feats, train_labels,
test_image_feats):
    '''
    This function will predict the category for every test image by finding
    the training image with most similar features. You will complete the given
    partial implementation of k-nearest-neighbors such that for any
    arbitrary
    k, your algorithm finds the closest k neighbors and then votes among them
    to find the most common category and returns that as its prediction.

    Inputs:
        train_image_feats: An nxd numpy array, where n is the number of
training
                           examples, and d is the image descriptor vector size.
        train_labels: An nx1 Python list containing the corresponding ground
truth labels for the training data.
        test_image_feats: An mxd numpy array, where m is the number of test
images and d is the image descriptor vector size.

    Outputs:
        An mx1 numpy list of strings, where each string is the predicted label

```

```
for the corresponding image in test_image_feats
```

The simplest implementation of k-nearest-neighbors gives an even vote to

all k neighbors found - that is, each neighbor in category A counts as one

vote for category A, and the result returned is equivalent to finding the

mode of the categories of the k nearest neighbors. A more advanced version uses weighted votes where closer matches matter more strongly than far ones.

This is not required, but may increase performance.

Be aware that increasing k does not always improve performance - even values of k may require tie-breaking which could cause the classifier to

arbitrarily pick the wrong class in the case of an even split in votes. Additionally, past a certain threshold the classifier is considering so many neighbors that it may expand beyond the local area of logical matches and get so many garbage votes from a different category that it mislabels the data. Play around with a few values and see what changes.

Useful functions:

```
scipy.spatial.distance.cdist, np.argsort, scipy.stats.mode
...
```

```
k = 9
```

```
# Gets the distance between each test image feature and each train image feature
```

```
# e.g., cdist
```

```
distances = cdist(test_image_feats, train_image_feats, 'euclidean')
```

```
#TODO:
```

```
# 1) Find the k closest features to each test image feature in euclidean space
```

```
# 2) Determine the labels of those k features
```

```
# 3) Pick the most common label from the k
```

```
# 4) Store that label in a list
```

```
labels = []
```

```
for i in range(len(test_image_feats)):
```

```
    nn = np.argsort(distances[i])
```

```
    nn_label = np.array(train_labels)[nn[:k]]
```

```
    mode, cnt = stats.mode(nn_label)
```

```
print(mode)
labels.append(mode[0])
return np.array(labels)
```

这一部分代码的构建与 SVM 分类器的构建相类似，因此不再多赘述。

六、实验数据及结果分析：

在完成 student.py 文件的完善之后，下面运行 main.py 文件测试代码和模型的效果，训练的数据共有 15 个类别，包括 bedroom, coast 等等图片。每一类别中选取了 100 张图片，共计 1500 张图片的数据。测试数据同样是 15 个类别，总数多于 1500 张。

下面给出一些模型的详细参数设置：

- 1)在 tiny 特征中给出的单个图片的尺寸为 9*9
- 2)词袋模型构建时使用 sift 特征，之后 K-means 聚类选取 cluster 的个数为 500 个
- 3)词袋模型中的视觉词的个数为 300
- 4)KNN 分类器中的 k 选取为 10

最终得到的结果如下：

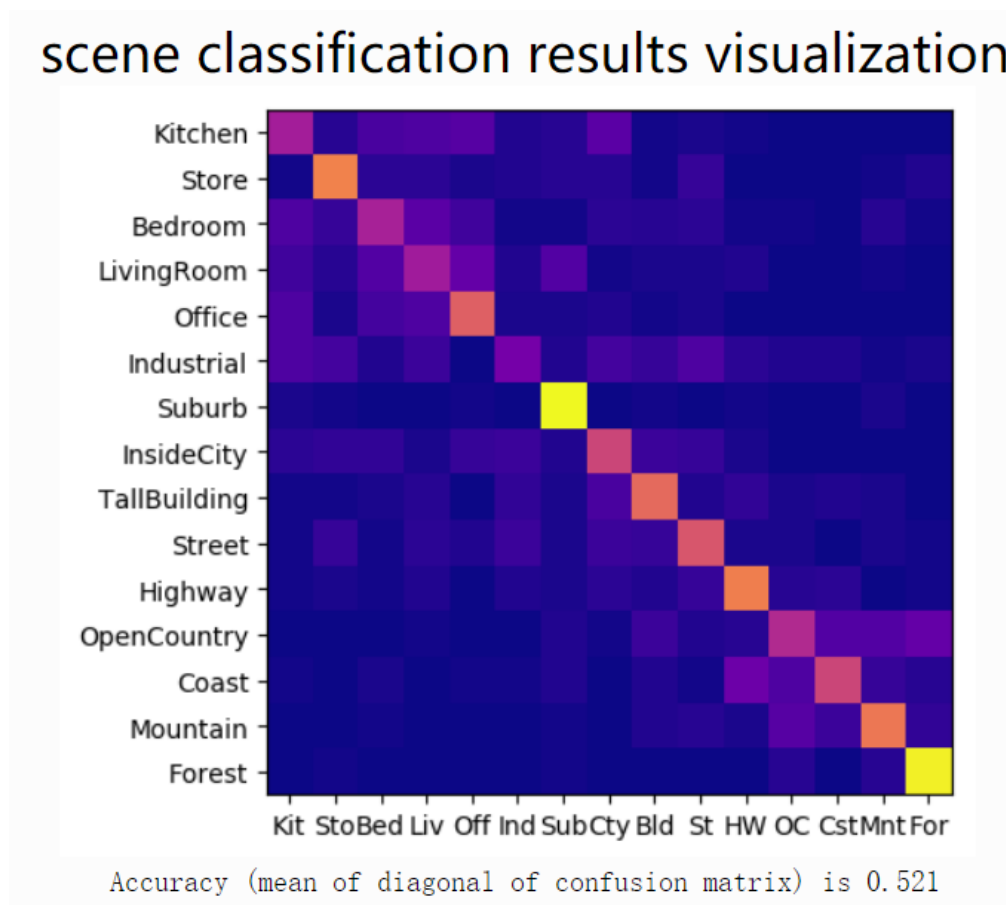


















































































































图 2：混淆矩阵以及测试准确率示意图

具体每一个类别分类的准确率如下：

Category name	Accuracy	Sample training images		Sample true positives		False positives with true label		False negatives with wrong predicted label	
Kitchen	0.330					 Bedroom	 LivingRoom	 Highway	 Mountain
Store	0.640					 Store	 Store	 Highway	 Suburb
Bedroom	0.340					 LivingRoom	 Bedroom	 Forest	 Industrial
LivingRoom	0.320					 Highway	 Kitchen	 Forest	 Office
Office	0.540					 Office	 LivingRoom	 TallBuilding	 Store
Industrial	0.210					 Industrial	 Industrial	 Office	 Suburb
Suburb	0.920					 LivingRoom	 Street	 Office	 Highway
InsideCity	0.460					 Street	 Industrial	 Suburb	 LivingRoom
TallBuilding	0.570					 OpenCountry	 TallBuilding	 Industrial	 Highway
Street	0.510					 Street	 Street	 Kitchen	 TallBuilding
Highway	0.630					 Highway	 Coast	 Forest	 Mountain
OpenCountry	0.370					 OpenCountry	 OpenCountry	 Highway	 Highway
Coast	0.460					 Coast	 Coast	 Mountain	 LivingRoom
Forest	0.900					 OpenCountry	 Forest	 Industrial	 Bedroom
Category name	Accuracy	Sample training images		Sample true positives		False positives with true label		False negatives with wrong predicted label	

整体来看，总体分类准确率较低只有 52.1%，对于 15 个具体的类别来说，其中 Suburb 和 Forest 的分类准确率较高，可以达到 90% 以上，而 kitchen, bedroom, living room, Industrial 等分类准确率较低，只有 30% 左右。整体来看自然风景图片的预测准确率高于其他的一些图片，尤其是各类房间的照片。

七、实验结论：

本次实验主要实现了对于图像的特征提取与分类，提取特征的方法主要是基于 SIFT 特征然后聚类得到的 bag of words 模型，主要实现的功能包括 SIFT 特征的提取与聚类，图片直方图特征的表示以及 SVM，KNN 两个分类器的构建。

分析最终实验得到的结果，最终预测的准确率只有 52.1%，偏低的原因分析下来，个人认为主要在于一些参数设置的问题，比如说 K-means 聚类时选取的簇的个数，考虑到 SIFT 提取了 1500 张图片的特征，每个图片有上百个 SIFT 特征，聚类时样本的数量应该达到十万到百万级别的数据，因此聚类的簇选取应该更大一些，得到的视觉词的数量可以更多则最终可以得到的特征信息阅读，更加有利于分类。

另外还有观察所有类别的分类准确率我们可以发现，其中一些自然风景的照片分类的准确率明显高一些，究其原因个人认为是自然风景图片的特征更加容易获取，特别是特定场景下的一些自然风景，特征会非常明显，而一些家庭起居或者其他工业图片往往特征不那么鲜明，而且在 BOW 这种模型忽略了图像内部之间的关联性后会使得最终的图像分类结果比较差。

八、总结及心得体会：

Project3 这个实验比较有意思，但是中途也遇到了很多的麻烦，首先第一个是在使用 opencv-python 这个库的时候发现想调用有关 SIFT 特征提取的方法时出现了问题，主要是在于版本的兼容性问题，需要使用 3.4.2.16 左右的版本。另外一个比较坑的地方是 skimage 库中的 resize 函数花费的时间是 opencv-python 中 resize 函数的很多很多倍，因此一开始使用 skimage 程序运行起来会非常的费时。当然在使用了 opencv-python 后跑程序仍然花了很多的时间，主要原因是数据量比较大，在构建 BOW 模型的视觉词 vocab 时非常的费时，因此这对于调参数带来了不小的影响。

九、对本实验过程及方法的改进建议：

- (1) 可以考虑对比使用一些其他特征提取的方法来进行图像的分类。
- (2) 可以引入一些深度网络模型来进行图像的分类识别。

报告评分：

指导教师签字：