电 子 科 技 大 学

实 验 报 告

学生姓名：谢卿云　　学号：2022010910017　指导教师：沈复民

一、 实验项目名称：

Scene Recognition with Bag of Words

二、 实验原理：

（一）Tiny Images 实验原理

Tiny Images 是一种简单而有效的图像特征表示方法，其核心思想是将高分辨率图像缩放到小尺寸，并将像素值展平作为特征向量。Tiny Images 常常在场景识别任务中作为基准方法或与其他特征结合使用时，仍能取得一定的效果。在本实验中，将 Tiny Images 与 KNN 分类器结合，作为探索不同特征和分类器组合的第一步。

2.1.1. 方法特点

Tiny Images 方法具有以下特点：

- **计算效率高**：特征提取过程简单快速
- **低维表示**：特征向量维度较低（如 256 维）
- **全局特征**：捕捉图像的整体低频信息
- **对细节不敏感**：对噪声和细节变化具有鲁棒性
- **对几何变换敏感**：对图像的平移、旋转和缩放敏感

2.1.2. 基本原理

Tiny Images 方法通过以下步骤将图像转换为特征向量：

1. **图像缩放**：将原始图像强制缩放到固定的小尺寸（如 16x16 像素），保留整体结构信息。
2. **灰度转换**：将彩色图像转换为灰度图像，简化特征表示。
3. **特征向量生成**：将缩放后的图像像素值按序排列，形成一维特征向量。对于 16x16 的灰度图像，得到 256 维向量。

4. **特征归一化**：对特征向量进行归一化处理，减少光照等因素的影响。

K 近邻 (K-Nearest Neighbors, KNN) 是一种简单直观的监督学习算法，主要用于分类和回归任务。在本实验中，它被用作分类器，特别是与 Tiny Images 特征结合使用。

（二）基本原理

KNN 的核心思想是：一个样本的类别由其在特征空间中的 K 个最近邻居的类别决定。

（三）算法步骤

算法流程如下伪代码所示

算法 1 Pseudocode for K-Nearest Neighbors (KNN) Classification

输入： Test data feature vector x_{test},Training data feature vectors TrainingData $= \{x_1, x_2, ..., x_n\}$,Corresponding training labels TrainingLabels $= \{y_1, y_2, ..., y_n\}$,Number of nearest neighbors K

输出： Predicted class label y_{pred} for x_{test}

Initialize an empty list: NeighborsList (to store Distance, Label tuples)
for each training sample x_i in TrainingData **do**
    Calculate the distance between x_{test} and x_i: $Distance_i \leftarrow \text{Distance}(x_{\text{test}}, x_i)$
    Get the corresponding label y_i from TrainingLabels
    Add tuple $(Distance_i, y_i)$ to NeighborsList
end for
Sort NeighborsList based on distance in ascending order
Select the first K elements from sorted NeighborsList as K_Nearest_Neighbors
Initialize an empty map: LabelCounts (to count frequency of labels)
for each neighbor $(distance, label)$ in K_Nearest_Neighbors **do**
    Increment count for label in LabelCounts
end for
Find the label with the maximum count in LabelCounts
$y_{\text{pred}} \leftarrow$ Label with highest count in LabelCounts ▷ Handle ties if necessary
return y_{pred}

（四）K 值选择

参数 K 是 KNN 算法最重要的决定因素：增大 K 可减少噪声影响，使决策边界更平滑, K 过大可能包含不相关邻居，导致性能下降, 通常通过交叉验证选择最优 K 值

（五）Bag of Words (BoW) 实验原理

Bag of Words (BoW) 模型最初用于文本分析，将文档表示为其包含的单词的频率直方图，忽略词语的顺序。将这个思想迁移到图像领域，就是将图像视为" 视觉词汇" 的集合。BoW 模型通过提取局部特征、构建视觉词汇表、生成特征直方图等步骤，实现了对图像内容的有效表示。这种方法能够捕捉图像中具有辨识度的局部信息，在许多图像分类任务中取得了不错的性能。在本实验中，我们使用 SIFT 特征实现对场景的分类；

2.5.1. 基本原理

BoW 模型通过以下步骤将图像转换为特征向量：

1. **局部特征提取**：使用 SIFT 算法检测图像中的关键点，计算每个关键点的 128 维特征描述符。特征描述符对尺度和旋转变换具有鲁棒性。
2. **构建视觉词汇表**：收集所有训练图像的 SIFT 特征描述符，使用 K-Means 聚类算法构建视觉词汇表。每个聚类中心代表一个视觉词汇，词汇表大小 (vocab_size) 需要预先设定。
3. **特征向量生成**：对每张图像提取 SIFT 特征，将特征映射到最近的视觉词汇，统计每个视觉词汇的出现频率，生成 vocab_size 维的直方图特征向量。
4. **分类**：使用 SVM 等分类器进行训练和预测，学习特征向量与图像类别的关系。

（六）支持向量机 (SVM) 实验原理

支持向量机 (Support Vector Machine, SVM) 是一种强大的监督学习模型，主要用于分类和回归任务。在本次场景分类实验中，我们将使用 SVM 作为 Bag of Words 特征的分类器。

SVM 的核心思想是寻找一个最优的超平面 (Hyperplane) 来在高维特征空间中将不同类别的样本分开。这个最优超平面不仅要能正确划分样本，还要使两类样本中的支持向量到超平面的间隔（Margin）最大化。最大化间隔可以提高分类器的

泛化能力，使其在新数据上表现更好。

2.6.1. 数学原理

对于一个二分类问题，给定一组带有标签的训练数据，SVM 的目标是找到一个决策函数，通常是线性的：

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$$

其中 $\mathbf{w}$ 是超平面的法向量，b 是偏置项。超平面由 $\mathbf{w} \cdot \mathbf{x} + b = 0$ 定义。

对于训练样本 (x_i, y_i)，其中 x_i 是特征向量，$y_i \in \{-1, 1\}$ 是类别标签，我们希望找到 $\mathbf{w}$ 和 b，使得：

$$y_i(\mathbf{w} \cdot x_i + b) \geq 1$$

同时，我们希望最小化 $\|\mathbf{w}\|^2$，等价于最大化间隔 $2/\|\mathbf{w}\|$），这是一个凸优化问题，可以通过拉格朗日乘子法等技术求解。

在实际应用中，数据往往不是完全线性可分的。SVM 引入了**软间隔 (Soft Margin)** 的概念，允许少量样本点违反间隔约束，即允许一些样本点位于间隔带内甚至错误的一侧。这通过引入松弛变量 $\xi_i \geq 0$ 和惩罚参数 C 来实现，优化目标变为最小化：

$$\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i$$

约束条件变为：

$$y_i(\mathbf{w} \cdot x_i + b) \geq 1 - \xi_i$$

2.6.2. Kernel Trick

对于非线性可分的数据，SVM 使用**核技巧 (Kernel Trick)** 将原始特征空间映射到更高维的空间，使得样本在该高维空间中变得线性可分。常用的核函数包括多项式核、径向基函数 (RBF) 核等。核函数 $K(\mathbf{x}_i, \mathbf{x}_j)$ 计算的是样本在映射后的高维空间中的内积，而无需显式计算高维映射本身，这大大提高了计算效率。

然而，在本次实验中，由于使用了 ‘sklearn.svm.LinearSVC‘，这表示我们主要考虑线性分类器，或者等价于使用了线性核 $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$。线性 SVM 在处理高维稀疏数据时通常非常高效，这与 Bag of Words 特征的特性相符。

2.6.3. 多类别分类

SVM 本身是二分类器。对于像场景分类这样的多类别任务（假设有 M 个类别），需要将二分类 SVM 扩展。常用的策略有两种：

1. **一对多 (One-vs-Rest, OvR):** 为每个类别训练一个二分类 SVM。例如，对于类别 k，训练一个分类器来区分类别 k 的样本和所有非类别 k 的样本。总共需要训练 M 个分类器。在预测时，将待分类样本输入所有 M 个分类器，选择输出分数最高（或离超平面最远）的那个类别作为预测结果。‘LinearSVC‘ 默认通常采用 One-vs-Rest 策略。
2. **一对一 (One-vs-One, OvO):** 为每一对不同的类别训练一个二分类 SVM。例如，对于类别 i 和类别 j，训练一个分类器来区分这两类样本。总共需要训练 $M(M-1)/2$ 个分类器。在预测时，将待分类样本输入所有分类器，然后使用投票机制决定最终类别。每个分类器都为其中一个类别投一票，得票最多的类别获胜。

在本次实验中，我们使用的 ‘LinearSVC‘ 通常采用 One-vs-Rest 策略来实现多类别分类。

三、 实验目的：

1. 理解并实践不同的图像全局特征和局部特征的提取与表示方法。
2. 学习并应用 K 近邻 (KNN) 和支持向量机 (SVM) 等经典分类器在图像识别任务中的原理和使用。
3. 通过比较不同特征和不同分类器组合的分类性能，分析它们对最终结果的影响。
4. 初步了解并实现基于深度神经网络 (DNN) 的场景分类方法，体验其与传统方法的差异。
5. 掌握评估分类模型性能的方法，并能够对实验结果进行分析和讨论，总结不同方法的优劣和适用场景。

四、 实验内容：

1. 分别利用 Tiny+KNN 和 Bags of Words (SIFT) + SVM 实现对场景的分类，并且比较不同特征和不同分类方法对最终精度的影响；
2. 用深度神经网络 DNN 进一步完成场景分类任务，对比其性能。

五、 实验步骤：

（一）实现 Tiny+KNN 方法

1. 实现 student.py/get_tiny_images(image_paths) 函数: 根据 Tiny Images 和 KNN 的实验原理，为输入的图像路径列表生成 Tiny Images 特征。输出一个 n x d 的 NumPy 数组，其中 n 是图像数量，d 是 Tiny Images 特征向量的长度（例如 16*16=256）；
2. 完善 ‘nearest_neighbor_classify(train_image_feats, train_labels, test_image_-feats)‘ 函数使用 KNN 分类器对测试图像进行分类。为输入的图像路径列表生成 Tiny Images 特征。输出一个 n x d 的 NumPy 数组，其中 n 是图像数量，d 是 Tiny Images 特征向量的长度（例如 16*16=256）；输出一个 m x 1 的 NumPy 数组，其中 m 是测试图像数量，每个元素是对应的预测类别标签。

（二）实现 Bag of Words (SIFT) + SVM 方法

1. 实现 student.py/build_vocabulary(image_paths, vocab_size) 函数根据 Bag of Words (SIFT) + SVM 的实验原理，从训练图像中提取 HOG 特征并构建视觉词汇，输出一个 vocab_size x (z*z*9) 的 NumPy 数组，代表聚类中心（即视觉词汇）。通常会将这个词汇保存到文件 vocab.npy。
2. 实现 get_bags_of_words(image_paths) 函数，为输入的图像路径列表生成 Bag of Words 直方图特征，加载已构建的视觉词汇 (vocab.npy)，输出一个 n x vocab_size 的 NumPy 数组，其中 n 是图像数量，vocab_size 是词汇大小，每个元素是对应的 Bag of Words 直方图特征向量。
3. 完善 svm_classify(train_image_feats, train_labels, test_image_feats) 函数；使用 SVM 分类器对测试图像进行分类输出一个 m x 1 的 NumPy 数组，其中 m 是测试图像数量，每个元素是对应的预测类别标签。

（三）比较分析

1. 在完成以上实现后，运行 proj4.ipynb 所有单元格。
2. 计算 Tiny+KNN、BoW+SVM 在测试集上的分类精度。
3. 比较这三种方法的精度, 分析不同特征表示和不同分类器对场景分类性能的影响。

六、 实验数据及结果分析：

运行第一组（Tiny+KNN）的结果如图 1 所示，可以看到整体准确率仅有 0.189，

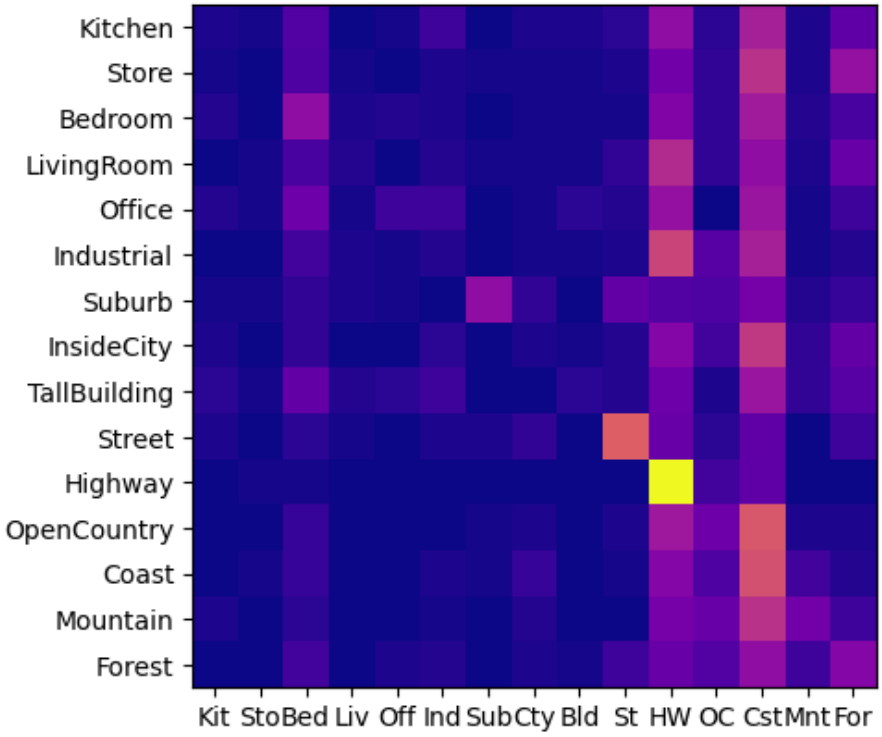

图 1: Tiny Image + KNN 混淆矩阵

接近随机猜测（理论上 15 类随机猜测准确率为 1/15 ≈ 0.067）。

运行第二组（Bags of Words + SVM）的结果如图 2 所示，整体准确率上升至 0.711，显著高于第一组。

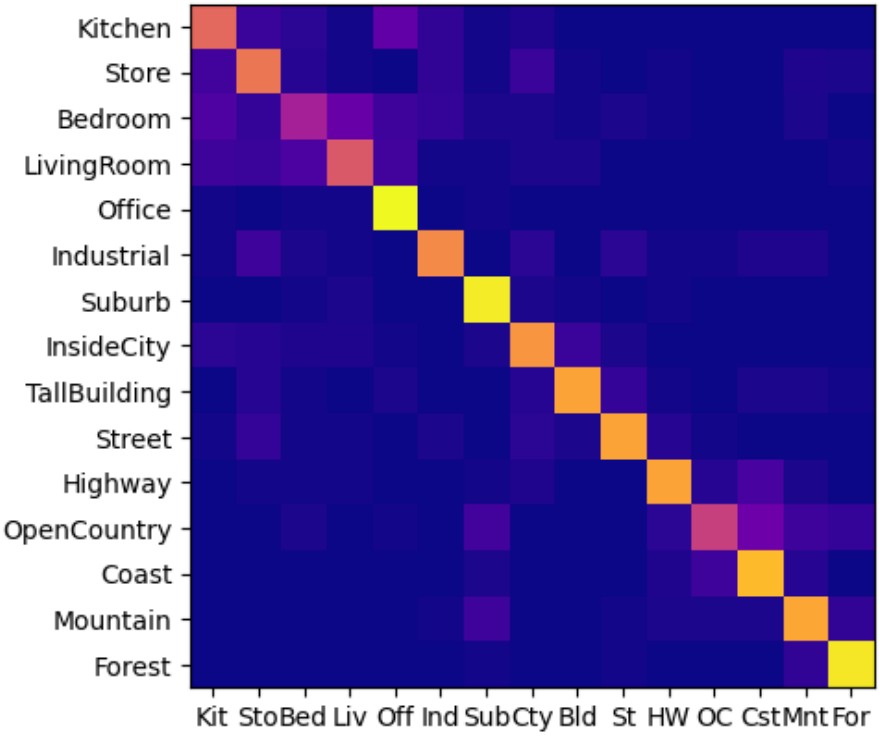

图 2: Bag of Words + SVM 混淆矩阵

七、 实验结论：

实验结果显示，第二种方法（BOW + SVM）的准确率远高于第一种方法（Tiny Image + KNN）的准确率。推测原因如下：

Tiny Image 特征是非常简单的全局特征，仅通过将图片缩放到非常小的尺寸来表示。这种方法丢失了图像的绝大部分细节信息，对图像的尺度、平移、旋转以及视角变化非常敏感。它只能捕获非常粗略的空间布局和颜色信息，难以区分细节丰富的不同场景。Bags of Words 特征是基于局部特征描述符（如 HOG 或 SIFT）构建的。这些局部描述符（如 SIFT）能够捕捉图像中具有区分性的局部模式（如边缘、角点、纹理等），并且对局部的光照、尺度和旋转变化具有一定的鲁棒性。将这些局部特征聚类形成” 视觉词汇”，并将图像表示为这些视觉词汇的直方图，能够更有效地捕获场景的构成元素及其分布，从而提供了比 Tiny Image 更丰富、更具判别力的图像表示。

KNN 分类器是一种基于实例的学习方法，它直接根据测试样本与训练样本在特征空间中的距离进行分类。虽然简单直观，但 KNN 对噪声和不相关的特征比较敏感，并且在处理高维稀疏数据（如 Bag of Words 直方图）时，” 距离” 的定义和

计算可能会受到” 维度灾难” 的影响，性能可能受限。SVM 是一种功能强大的分类器，尤其擅长处理高维数据。SVM 尝试在特征空间中找到一个最优的超平面来最大化不同类别之间的间隔。线性 SVM（LinearSVC）虽然是线性的，但它在高维空间中表现良好，并且通过合适的正则化可以有效地防止过拟合。相比于 KNN 仅依赖局部少数样本，SVM 学习的是一个全局的决策边界，这通常在高维复杂的特征空间中更具优势。

综上所述，Bags of Words 方法通过更鲁棒、更具判别力的局部特征表示捕获了场景的关键信息，而 SVM 分类器则能够有效地利用这种高维特征进行分类。因此，BOW + SVM 的组合在场景分类任务上取得了显著优于 Tiny Image + KNN 方法的性能。

八、 总结及心得体会：

我深入理解了基于传统特征和机器学习方法的图像场景分类流程，包括特征提取、特征表示（Bag of Words 模型）和分类器设计。掌握了 Tiny Image 特征和局部描述符（如 HOG/SIFT）的基本原理及其在场景识别中的应用，认识到不同特征对最终性能的影响。

我实践了聚类算法在构建视觉词汇中的作用，理解了 Bag of Words 模型如何将变长的局部特征序列转化为固定长度的图像表示向量。对比了基于实例的 KNN 分类器和基于决策边界的 SVM 分类器在处理图像特征时的特点和性能差异，认识到 SVM 在高维特征空间中的优势。

通过本次场景分类实验，我对经典计算机视觉方法有了更直观的认识，为其在现代深度学习方法中的演进奠定了基础。

九、 对本实验过程及方法、手段的改进建议及展望：

本次实验在经典场景分类方法上取得了一定的结果，但仍有许多可以改进的地方，包括但不限于：

1. 改进 Bag of Words 模型构建，除了简单的词频直方图，可以考虑使用 TF-IDF 等加权方法。
2. 本实验采取了线性核函数应用于 SVM 分类器，可以进一步尝试使用非线性核函数等。
3. 可以考虑构建深度神经网络 (DNN)，作为目前最先进的图像场景分类方法普遍基于深度学习。可以利用卷积神经网络 (CNN) 来自动学习图像的

层次化特征表示，并用于场景分类。这通常能取得比传统方法更高的性能。受限于设备，这一步未能完成。

通过以上改进，有望进一步提升场景分类的准确率和鲁棒性。

报告评分：

指导教师签字：

附录一 代码示例

核心代码如代码 1 所示。

代码 1: student.py

```
  import numpy as np
import matplotlib
from skimage.io import imread
from PIL import Image

# from skimage.color import rgb2grey
from skimage.feature import hog
from skimage.transform import resize
from scipy.spatial.distance import cdist
from sklearn.cluster import MiniBatchKMeans, KMeans
from sklearn.svm import LinearSVC
import os
import glob
from skimage.color import rgb2gray
from numpy.linalg import norm
from scipy.stats import mode


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.
    Outputs:
        An n x 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 16x16, then d is 16 * 16 = 256.

    To build a tiny image feature, resize the original image to a very small
    square resolution (e.g. 16x16). 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, PIL.Image.open, np.std
                             ...
    """

    # TODO: Implement this function!
    images = np.zeros((len(image_paths), 256))
    for i, file in enumerate(image_paths):
        img = imread(file)
        # img = rgb2gray(img)
        img = resize(img, (16, 16), anti_aliasing=True).flatten()
        images[i] = img / norm(img)
    return images


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)

    Also, you may use other feature extractor like SIFT. It's okay to use
        skimage!

    suggested func:
        skimage.hog, sklearn.cluster.MiniBatchKMeans,...
    """

    # TODO: Implement this function!
    image_list = [imread(file) for file in image_paths]
    cells_per_block = (2, 2)
    z = cells_per_block[0]
    pixels_per_cell = (4, 4)
    feature_vectors_images = []
    for image in image_list:
        feature_vectors = hog(
            image,
            feature_vector=True,
            pixels_per_cell=pixels_per_cell,
            cells_per_block=cells_per_block,
            visualize=False,
        )
        feature_vectors = feature_vectors.reshape(-1, z * z * 9)
        feature_vectors_images.append(feature_vectors)
    all_feature_vectors = np.vstack(feature_vectors_images)
    kmeans = MiniBatchKMeans(n_clusters=vocab_size, max_iter=500).fit(
        all_feature_vectors)
    vocabulary = np.vstack(kmeans.cluster_centers_)
    return vocabulary


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
    """

    # TODO: Implement this function!
    vocab = np.load("vocab.npy")
    print("Loaded vocab from file.")
```

```
    vocab_length = vocab.shape[0]
    image_list = [imread(file) for file in image_paths]

    images_histograms = np.zeros((len(image_list), vocab_length))
    cells_per_block = (2, 2)
    z = cells_per_block[0]
    pixels_per_cell = (4, 4)
    feature_vectors_images = []
    for i, image in enumerate(image_list):
        feature_vectors = hog(
            image,
            feature_vector=True,
            pixels_per_cell=pixels_per_cell,
            cells_per_block=cells_per_block,
            visualize=False,
        )
        feature_vectors = feature_vectors.reshape(-1, z * z * 9)
        histogram = np.zeros(vocab_length)
        distances = cdist(feature_vectors, vocab)
        closest_vocab = np.argsort(distances, axis=1)[:, 0]
        indices, counts = np.unique(closest_vocab, return_counts=True)
        histogram[indices] += counts
        histogram = histogram / norm(histogram)
        images_histograms[i] = histogram
    return images_histograms


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.

    suggested function: sklearn.svm.LinearSVC
    """

    # TODO: Implement this function!
    clf = LinearSVC(random_state=0, tol=1e-5)
    clf.fit(train_image_feats, train_labels)
    test_predictions = clf.predict(test_image_feats)
    return test_predictions


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.
```

```
256
257     Inputs:
258         train_image_feats: An nxd numpy array, where n is the number of
                training
259                            examples, and d is the image descriptor vector size
                               .
260         train_labels: An nx1 Python list containing the corresponding ground
261                       truth labels for the training data.
262         test_image_feats: An mxd numpy array, where m is the number of test
263                           images and d is the image descriptor vector size.
264
265     Outputs:
266         An mx1 numpy list of strings, where each string is the predicted label
267         for the corresponding image in test_image_feats
268
269     The simplest implementation of k-nearest-neighbors gives an even vote to
270     all k neighbors found - that is, each neighbor in category A counts as one
271     vote for category A, and the result returned is equivalent to finding the
272     mode of the categories of the k nearest neighbors. A more advanced version
273     uses weighted votes where closer matches matter more strongly than far
            ones.
274     This is not required, but may increase performance.
275
276     Be aware that increasing k does not always improve performance - even
277     values of k may require tie-breaking which could cause the classifier to
278     arbitrarily pick the wrong class in the case of an even split in votes.
279     Additionally, past a certain threshold the classifier is considering so
280     many neighbors that it may expand beyond the local area of logical matches
281     and get so many garbage votes from a different category that it mislabels
282     the data. Play around with a few values and see what changes.
283
284     Useful functions:
285         scipy.spatial.distance.cdist, np.argsort, scipy.stats.mode
286     """
287
288     k = 5
289
290     # Gets the distance between each test image feature and each train image
            feature
291     # e.g., cdist
292     distances = cdist(test_image_feats, train_image_feats, "euclidean")
293
294     # TODO:
295     # 1) Find the k closest features to each test image feature in euclidean
            space
296     # 2) Determine the labels of those k features
297     # 3) Pick the most common label from the k
298     # 4) Store that label in a list
299
300     sorted_indices = np.argsort(distances, axis=1)
301     knns = sorted_indices[:, 0:k]
302
303     labels = np.zeros_like(knns)
304     get_labels = lambda t: train_labels[t]
305     vlabels = np.vectorize(get_labels)
306
307     labels = vlabels(knns)
308     labels = mode(labels, axis=1)[0]
309
310     return labels
```