Assignment 3: Scene Recognition with Bag of Words

Overview

The goal of this project is to introduce you to image recognition. Specifically, we will examine the task of scene recognition starting with very simple methods -- tiny images and nearest neighbor classification -- and then move on to more advanced methods -- bags of quantized local features and linear classifiers learned by support vector machines.

Bag of words models are a popular technique for image classification inspired by models used in natural language processing. The model ignores or downplays word arrangement (spatial information in the image) and classifies based on a histogram of the frequency of visual words. The visual word "vocabulary" is established by clustering a large corpus of local features. See Szeliski chapter 14.4.1 for more details on category recognition with quantized features. In addition, 14.3.2 discusses vocabulary creation and 14.1 covers classification techniques.

For this project you will be implementing a basic bag of words model with many opportunities for extra credit. You will classify scenes into one of 15 categories by training and testing on the 15 scene database (introduced in **Lazebnik et al. 2006**, although built on top of previously published datasets). **Lazebnik et al. 2006** is a great paper to read, although we will be implementing the *baseline method* the paper discusses (equivalent to the zero level pyramid) and not the more sophisticated spatial pyramid (which is extra credit). For an excellent survey of pre-deep-learning feature encoding methods for bag of words models see **Chatfield et al, 2011**.

Example scenes from of each category in the 15 scene dataset. Figure from **Lazebnik et al. 2006**.

Details and Starter Code

The primary script for this project is `student_code.py`. The ipython notebook `assignment3.ipynb` runs through the function calls required to implement the project. Your task is to implement the methods within the primary script that will allow you to achieve the desired accuracies on the notebook.

You are required to implement 2 different image representations -- **tiny images and bags of SIFT features** -- and 2 different classification techniques -- **nearest neighbor and linear SVM**. In the writeup, you are specifically asked to report performance for the following combinations, and it is also highly recommended that you implement them in this order:

- Tiny images representation and nearest neighbor classifier (accuracy of about 18-25%).
- Bag of SIFT representation and nearest neighbor classifier (accuracy of about 50-60%).
- Bag of SIFT representation and linear SVM classifier (accuracy of about 60-70%).

You will start by implementing the tiny image representation and the nearest neighbor classifier. They are easy to understand, easy to implement, and runs very quickly for our experimental setup (less than 10 seconds).

The "tiny image" feature, inspired by the work of the same name by **Torralba, Fergus, and Freeman**, is one of the simplest possible image representations. One simply resizes each image to a small, fixed resolution (we recommend 16x16). It works slightly better if the tiny image is made to have 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. **Torralba, Fergus, and Freeman** propose several alignment methods to alleviate the latter drawback, but we will not worry about alignment for this project. We are using tiny images simply as a baseline.
See `get_tiny_images()` in the starter code for more details.

The nearest neighbor classifier is equally simple to understand. When tasked with classifying a test feature into a particular category, one simply finds the "nearest" training example (L2 distance is a sufficient metric) and assigns the test case the label of that nearest training example. The nearest neighbor classifier has many desirable features -- it requires no training, it can learn arbitrarily complex decision boundaries, and it trivially supports multiclass problems. It is quite vulnerable to training noise, though, which can be alleviated by voting based on the K nearest neighbors (but you are not required to do so). Nearest neighbor classifiers also suffer as the feature dimensionality increases, because the classifier has no mechanism to learn which dimensions

are irrelevant for the decision. The nearest neighbor computation also becomes slow for high dimensional data and many training examples.
See `nearest_neighbor_classify()` for more details.

Together, the tiny image representation and nearest neighbor classifier will get about 15% to 25% accuracy on the 15 scene database. For comparison, chance performance is ~7%.

After you have implemented a baseline scene recognition pipeline it is time to move on to a more sophisticated image representation -- bags of quantized SIFT features. Before we can represent our training and testing images as bag of feature histograms, we first need to establish a *vocabulary* of visual words. We will form this vocabulary by sampling many local features from our training set (10's or 100's of thousands) and then clustering them with **kmeans**. The number of kmeans clusters is the size of our vocabulary and the size of our features. For example, you might start by clustering many SIFT descriptors into k=50 clusters. This partitions the continuous, 128 dimensional SIFT feature space into 50 regions. For any new SIFT feature we observe, we can figure out which region it belongs to as long as we save the centroids of our original clusters. Those centroids are our visual word vocabulary. Because it can be slow to sample and cluster many local features, the starter code saves the cluster centroids and avoids recomputing them on future runs.
See `build_vocabulary()` for more details.

Now we are ready to represent our training and testing images as histograms of visual words. For each image we will densely sample many SIFT descriptors. Instead of storing hundreds of SIFT descriptors, we simply count how many SIFT descriptors fall into each cluster in our visual word vocabulary. This is done by finding the nearest neighbor kmeans centroid for every SIFT feature. Thus, if we have a vocabulary of 50 visual words, and we detect 220 SIFT features in an image, our bag of SIFT representation will be a histogram of 50 dimensions where each bin counts how many times a SIFT descriptor was assigned to that cluster and sums to 220. The histogram should be normalized so that image size does not dramatically change the bag of feature magnitude.
See `get_bags_of_sifts()` for more details.

You should now measure how well your bag of SIFT representation works when paired with a nearest neighbor classifier. There are *many* design decisions and free parameters for the bag of SIFT representation (number of clusters, sampling density, sampling scales, SIFT parameters, etc.) so accuracy might vary from 50% to 60%.

The last task is to train **1-vs-all linear SVMS** to operate in the bag of SIFT feature space. Linear classifiers are one of the simplest possible learning models. The feature space is partitioned by a learned hyperplane and test cases

are categorized based on which side of that hyperplane they fall on. Despite this model being far less expressive than the nearest neighbor classifier, it will often perform better. For example, maybe in our bag of SIFT representation 40 of the 50 visual words are uninformative. They simply don't help us make a decision about whether an image is a 'forest' or a 'bedroom'. Perhaps they represent smooth patches, gradients, or step edges which occur in all types of scenes. The prediction from a nearest neighbor classifier will still be heavily influenced by these frequent visual words, whereas a linear classifier can learn that those dimensions of the feature vector are less relevant and thus downweight them when making a decision. There are numerous methods to learn linear classifiers but we will find linear decision boundaries with a support vector machine. You do not have to implement the support vector machine. However, 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". E.g. 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' which controls how strongly regularized the model is. Your accuracy will be very sensitive to lambda, so be sure to test many values. See `svm_classify()` for more details.

Now you can evaluate the bag of SIFT representation paired with 1-vs-all linear SVMs. Accuracy should be from 60% to 70% depending on the parameters. You can do better still if you implement extra credit suggestions below.

The starter code, starting from `student_code.py` contains more concrete guidance on the inputs, outputs, and suggested strategies for the five functions you will
implement: `get_tiny_images()`, `nearest_neighbor_classify()`, `build_vocabulary()`, `get_bags_of_sifts()`, and `svm_classify()`. The utils folder contains code required to visualize the results on the notebook, which you are not required to edit.

Experimental Design

An important aspect of machine learning is to estimate "good" hyper-parameters. As part of this project, you will perform the following three experiments and analyze the results in your writeup:

- Use cross-validation to measure performance rather than the fixed test / train split provided by the starter code. Randomly pick 100 training and 100 testing images for each iteration and report average performance and standard deviations.
- Add a validation set to your training process to tune learning parameters. This validation set could either be a subset of the training set or some of the otherwise unused test set.
- Experiment with many different vocabulary sizes and report performance. E.g. 10, 20, 50, 100, 200, 400, 1000, 10000.

Evaluation and Visualization

The starter code builds a confusion matrix and visualizes your classification decisions by producing a **confusion matrix between the ground truth test labels and the predicted test labels** within the notebook, each time you run `student_code.py`.

Useful Functions

The starter code contains more complete discussions of useful functions from Scikit-Learn utilities to VLFeat commands.

Handing in

This is very important as you will lose points if you do not follow instructions. Every time you do not follow instructions, you will lose 5 points. The folder you hand in must contain the following:

- code/ - directory containing all your code for this assignment
- code/vocab.pkl - please make sure the vocabulary file you want us to run your code with is named vocab.pkl

Rubric

- +20 pts: Build tiny image features for scene recognition. (`get_tiny_images()`)
- +10 pts: Nearest neighbor classifier. (`nearest_neighbor_classify()`)
- +20 pts: Build a vocabulary from a random set of training features. (`build_vocabulary()`)
- +20 pts: Build histograms of visual words for training and testing images. (`get_bags_of_sifts()`)
- +10 pts: Train 1-vs-all SVMs on your bag of words model. (`svm_classify()`)
- +10 pts: Accelerate the process of generating sift features.
- +10 pts: Add cross validation and the accuracy of three combinations exceeds the average.(i.e. 21%, 55% and 65%)
- -10 pts: Bad code style.

Final Advice

- Extracting features, clustering to build a universal dictionary, and building histograms from features can be slow. A good implementation can run the entire pipeline in less than 10 minutes, but this may be at the expense of accuracy (e.g. too small a vocabulary of visual words or too sparse a sampling rate).
- Recommend you to create a conda environment using the configuration we provide(command: conda env create -f environment_assignment3.yml). Remember to activate the environment first when you open the notebook. If there are some errors in your notebook kernel, you can refer to the following configuration.