CS 485/584
Problem Set 3

Due Nov 15th

Setup

- Download and extract the data from the link data.zip. Place the data folder inside the project folder.
- Create a conda environment for this assignment as in previous assignments. Follow the instructions below to set up the environment. If you run into import module errors, try `pip install -e .` again, and if that still doesn't work, you may have to create a fresh environment.
- Install Miniconda. It doesn't matter whether you use Python 2 or 3 because we will create our own environment that uses 3 anyway.
- Create a conda environment using the appropriate command. On Windows, open the installed **Conda prompt** to run the command. On MacOS and Linux, you can just use a terminal window to run the command. Modify the command based on your OS (linux, mac, or win):
  `conda env create -f proj3_env_<OS>.yml`
- This should create an environment named **proj3**. Activate it using the Windows command, `activate proj3` or the MacOS/Linux command, `source activate proj3`
- Install the project package, by running `pip install -e .` inside the project folder.
- Run the notebook using `jupyter notebook ./proj3_code/proj3.ipynb`
- Ensure that all sanity checks are passing by running `pytest` either inside the `proj3_unit_tests` folder, or directly in the project directory.

Dataset

The dataset to be used in this assignment is the 15-scene dataset, containing natural images in 15 possible scenarios like bedrooms and coasts. It was first introduced by Lazebnik et al, 2006. The images have a typical size of around 256 by 256 pixels, and serve as a good milestone for many vision tasks. A sample collection of the images can be found below:

Part 1: Tiny Image Representation and Nearest-Neighbor Classification

Introduction

In this part we will setup the workflow for nearest-neighbor classification. It is easy to understand, implement, and runs very quickly for our experimental setup (less than 10 seconds).

We will use tiny image representations for this part. 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. It simply resizes each image to a small, fixed resolution. The representations are then made to have zero mean and unit length for a slightly better performance. 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. The feature representaion has been implenented for you. Check `get_tiny_images()` in `utils.py` for implementation details.

The k nearest-neighbors classifier is equally simple to understand. When tasked with classifying a test feature into a particular category, one simply finds the k **nearest** training examples (L2 distance is a sufficient metric; you'll be implementing this in `pairwise_distances(X, Y)`) and assigns the test case the most common label among the k nearest neighbors. The k nearest neighbor classifier has many desirable features – it requires no training, it can learn arbitrarily complex decision boundaries, and it trivially supports multi-class problems. The voting aspect also alleviates training noise. K-nearest neighbor classifiers also suffers from the curse of

dimensionality, because the classifier has no mechanism to learn which dimensions are not relevant for the decision. The k neighbor computation also becomes slow for many training examples.

Part 1.1: Pairwise Distances (10 points)

- Implement `pairwise_distances()` in `student_code.py`.
- Use the Euclidean distance metric. You'll be using this implementation of `pairwise_distances()` in `nearest_neightbor_classify()`, `kmeans()` and `kmeans_quantize()`.

Note that you are **NOT** allowed to use any library functions like `pairwise_distances` from sklearn, pdist or scipy to help you do the calculation.

Part 1.2: K Nearest-Neighbors Classification (20 points)

- In `student_code.py`, implement the function `nearest_neightbor_classify()`.
- Given the training images features and labels, together with the testing features, classify the testing labels using the k nearest neighbors found in the training set. Your k nearest neighbors would vote on what to label the data point. The pipeline in the Jupyter Notebook will also walk you through the performance evaluation via a simple confusion matrix.

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 $\sim 7\%$. Paste the results for experiments in the report. Note that you need to tune your parameters and reach **15**% to get full credits for this part.

Part 1.3 Experiment and Report

Now we will perform experiments with the values of **tiny image size** and **k** in kNN.

(a) Report the performance of the standard parameters (image size = 16×16, k = 3).

(b) Test the following image sizes: 8×8, 16×16, 32×32, and the following **k**: 1, 3, 5, 10, 15. Report the performance

(c) In the writeup reflect on the difference in performance and processing time between the standard parameters and the experimental parameters you tried.

Part 2: Bag-of-words with SIFT Features

Introduction

Learning Objectives:

1. Understanding the concept of visual words
2. Set up the workflow for **k-means** clustering to construct the visual vocabulary.
3. Combine with the previous implemented k nearest-neighbor pipeline for classification.

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 a 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 (10s or 100s of thousands) and then clustering them with k-means. The number of k-means 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 k-means 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. A more intuitive way to think about this is through the original bag-of-words model in NLP: assume we have a vocab of ["Messi", "Obama"], and article A contains 15 occurrences of "Messi" with 1 "Obama", while article B with 2 "Messi" and 10 "Obama", then we may safely assume that article A is focusing on sports and B on politics, relatively speaking (unless Messi actually decides to run for the president).

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 k nearest-neighbor classifier. There are many design decisions and free parameters for the bag of SIFT representation (number of clusters, sampling density, SIFT parameters, etc.) so accuracy might vary from 40% to 60%.

Part 2.1: k-means (10 + 10 points)

- In `student_code.py`, implement the function `kmeans()` and `kmeans_quantize()`.
- `kmeans()` is used to perform the k-means algorithm and generate the centroids given the raw data points.
- `kmeans_quantize()` will assign the closest centroid to each of the new data entry by returning the indices.

For the `max_iter` parameter, the default value is 100, but you may start with small value like 10 to examine the correctness of your code, 10 is also sufficiently good to get you a decent result coupling with proper choice of k in kNN.

Note that in this part you are **NOT** allowed to use any clustering function from scipy or sklearn to perform the k-means algorithm.

Part 2.2: Build the Visual Vocab (10 points)

- In `student_code.py`, implement the function `build_vocabulary()`.
- For each of the **training** images, sample SIFT features uniformly using `generate_sample_points` in `utils.py` with a stride size of 20 for both horizontal and vertical directions.
- Concatenate all features from all images, and perform k-means on this collection, the resulting **centroids** will be the visual vocabulary. We have provided a working edition of SIFTNet for you to use in this part.

Part 2.3: Put it together with kNN Classification (10 points)

Now that we have obtained a set of visual vocabulary, we are now ready to quantize the input SIFT features into a bag of words for classification.

- In `student_code.py`, implement the function `get_bags_of_sift()`, and complete the classification workflow in the Jupyter Notebook.

- Sample uniformly from both training and testing images for their SIFT features (you may want to sample with a smaller stride size), and quantize the features according to the vocab we've built (which centroid should be assigned to the current feature?), and by computing the vocab histogram of all the features for a particular image, we are able to tell what's the most important aspect of the image.
- Given bag of SIFT features for both the training and testing images, since we know the ground truth for the training features, we can use nearest-neighbor to identify the labels for the testing feature. Again, in this part, you need to tune your parameters and obtain a final accuracy of at least **45**% to get full credits for this part.
- In the writeup reflect on the Tiny Image Representation vs the Bag of words with SIFT features. Why do you think that the tiny image representation gives a much worse accuracy than bag of words? Additionally why do you think Bag of Words is better in this case?

Part 2.4 Experiment and Report

(a) Using `stride(build_vocab) = 20, stride(bags_of_sift) = 5, max_iter(kmeans)=10`, play around with different values for the following parameters:

   (a) `vocab_size`: Try 50, 100 and 200
   (b) `k` in `kNN`: Try 1, 3, 5, 10, 15.

   Paste the confusion matrix with your best result onto the report, and make sure to note your parameter settings.

(b) Compare the difference here versus the k value experiment in Part 1: what can you tell from it?

Extra Credit

Part 3.1 Additional Experimentation

This is an extension of 2.4. Play around with the different values for the following parameters. Paste the confusion matrix with your best result onto the report, and make sure to note your parameter settings.

1. `max_iter` in `kmeans()`
2. `stride` or `step_size` in both `build_vocabulary()` and `get_bags_of_sifts()`

Part 3.2 Performance

Use the standard parameters and try a different distance metric for your kNN and report your accuracy. You should be able to achieve higher accuracy.

Part 3.3 Model

Set up the experiment with SVM, and reach the accuracy of 65%. Hints: you may take a look at the SVC classifier in sklearn; also note that common case of SVC classifies object following the one-against-one scheme, so you may need to do some extra work to make it fit in the multi-class problem. If you manage to get an accuracy above 60%, add that to your report!

Writeup

For this project (and all other projects), you must do a project report using the template slides provided to you. Do not change the order of the slides or remove any slides, as this will affect the grading process on Gradescope and you will be deducted points. In the report you will describe your algorithm and any decisions you made to write your algorithm a particular way. Then you will show and discuss the results of

your algorithm. The template slides provide guidance for what you should include in your report. A good writeup doesn't just show results, it also tries to draw some conclusions from the experiments. You must convert the slide deck into a PDF for your submission.

Rubric

- +70 pts: Code
  - 30 pts: Part 1
  - 40 pts: Part 2
- +30 pts: PDF report
  - 15 pts: Part 1
  - 15 pts: Part 2
- +11 pts: Extra Credit
  - 4 pts: Additional Experimentation
  - 2 pts: Performance
  - 5 pts: Model

Submission Instructions

You will have two submission files for this project:

1) proj3_code/student_code.py

2) proj3_report.pdf (change the name proj3_template.pptx to proj3_report.pdf)