Project 3 Writeup

Instructions

- This write-up is intended to be 'light'; its function is to help us grade your work.
- Please describe any interesting or non-standard decisions you made in writing your algorithm.
- Show your results and discuss any interesting findings.
- List any extra credit implementation and its results.
- Feel free to include code snippets, images, and equations.
- Use as many pages as you need, but err on the short side.
- **Please make this document anonymous.**

Project Overview

This project consisted of classifying a set of test scenes, given a labelled training set.

Instructions

1. Report your classification performance for the three recognition pipelines: tiny images + nearest neighbor, bag of words + nearest neighbor, and bag of words + one vs. all linear SVM.
2. For your best performing recognition setup, please include the full confusion matrix and the table of classifier results produced by the starter code. The submission script will include this html file.

Writeup

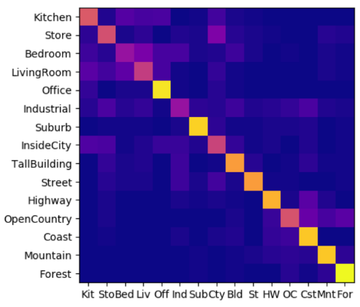
The process of scene recognition happens in two major parts.

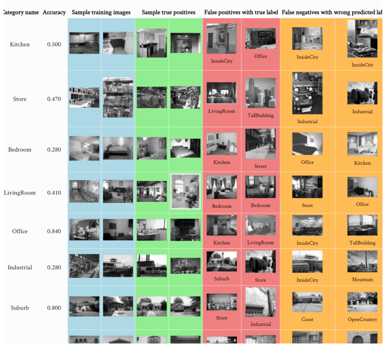
- Image feature recognition, using either tiny images or the bag of words method. In tiny images, every image is resized to 16x16, and the dominant visual characteristics are compared to a data set of 8M similarly sized images. In the bag of words model, the hog descriptor is used to describe features and add them into a vocabulary set. Then, the vocabulary is clustered with kMeans to a desired size. The labelled training images are then described with a vocabulary histogram (hog features are once again computed, and the features are binned into a histogram for each image). Test features are also described with a histogram, binned into a histogram. They are then classified.

- Image classification, using either support vector machines (SVM) or k nearest neighbors methods (knn). In SVM, a classifier is trained with the 1-vs-all method which will then classify the images! How neat! In the knn method, the distance from each test feature and each training features is computed, the top k test image neighbors for each training image are picked, and the mode label is computed, resulting in the most likely correct label for the test image.

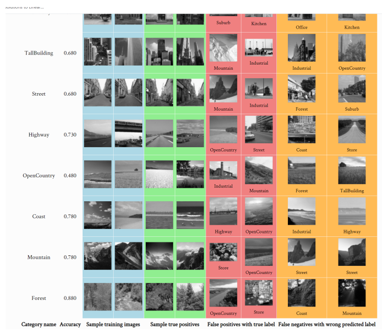
1. **Classification Performance**: I tuned k and vocab size to 17 and 200, respectively, because they were giving me the highest accuracy of the many combinations I tried:
   - Bag/SVM – 60.733%
   - Bag/KNN – 50.933%
   - Tiny/KNN – 20.256%

   Clearly, using bag of words for feature identification and SVM for classification yielded the highest accuracy of scene recognition.

2. 

3. 

ategory name	Accuracy	Sample training images		Sample true positives		False positives with true label		False negatives with wrong predicted lab	
Kitchen	0.500					InsideCity	Office	InsideCity	InsideCity
Store	0.470					LivingRoom	TallBuilding	Industrial	Industrial
Bedroom	0.280					Kitchen	Street	Office	Kitchen
LivingRoom	0.410					Bedroom	Bedroom	Store	Office
Office	0.840					Kitchen	LivingRoom	InsideCity	TallBuilding
Industrial	0.280					Suburb	Store	InsideCity	Mountain
Suburb	0.800					Store	Industrial	Coast	OpenCountry

4. 

Extra Credit

- I tried running get bag of words with different vocabulary sizes: with the following results for accuracy:
  - vocab = 10 – 47.333%
  - vocab = 20 – 52.133%
  - vocab = 50 – 55.067%
  - vocab = 100 – 54.733%
  - vocab = 200 – 60.133%
  - vocab = 500 – 52.067%
  - vocab = 2000 – 46.467%

  Since 200 vocab size gave the highest accuracy, it is likely that a vocabulary size in that range is ideal for this training set. Having too many words can make the histogram too sparse, while not having enough words will lead to a lot of random noise influencing the outcome, as there is less variety to classify features and make scenes unique. Additionally, having more than 200 vocabulary words made the program really slow.
- I also tried to use different types of SVM, and they are commented out in my code:

```
    # SVC(kernel = 'linear').fit(train_image_feats, train_labels).
                                    predict(test_image_feats)
    return SVC(kernel = 'rbf').fit(train_image_feats, train_labels
                                    ).predict(test_image_feats
                                    )
    # SVC(kernel = 'poly', degree=3).fit(train_image_feats,
                                    train_labels).predict(
                                    test_image_feats)
    # LinearSVC(penalty = 'l1', dual=False).fit(train_image_feats,
                                     train_labels).predict(
                                    test_image_feats)
    # return LinearSVC().fit(train_image_feats, train_labels).
                                    predict(test_image_feats)
```

  They all worked decently well (54%-61% accuracy) when I tuned the parameters (I didn't show me tuning them here because that would take many pages of python), but using the default SVC parameters honestly worked the best, which was surprising.