SYDE 671 Advanced Image Processing

David Abou Chacra University of Waterloo `Fall 2021`

Assignment 3 - Coding Part
Scene Recognition with Bag of Words

1 Instructions

- Deliverables: a zip file containing the folder *results_webpage*, your students.py file and (optionally) a Jupyter notebook with your write-up and any additional experiments/notes.
- Feel free to include images and equations and comment on both your codes and results meticulously.

You will perform scene recognition with three different methods, where you will classify scenes into one of 15 categories by training and testing on the 15 scene database Lazebnik et al. 2006.
The things that you need to implement are as usual in student.py.
The three classifiers are:

- Tiny images representation (get_tiny_images()) and nearest neighbor classifier (nearest_neighbor_classify()).
- Bag of words representation (build_vocabulary(), get_bags_of_words()) and nearest neighbor classifier.
- Bag of words representation and linear SVM classifier (svm_classify()).

Potentially useful functions:

- skimage.feature.hog() and similar feature extractors
- sklearn.cluster.KMeans()
- scipy.stats.mode()
- sklearn.svm.LinearSVC()
- skimage.transform.resize()
- skimage.util.crop()
- scipy.spatial.distance.cdist()

Forbidden functions: sklearn.neighbors.KNeighborsClassifier

2 Starter code details

Initial performance: Running main.py unchanged will randomly guess the category of every test image. This achieves 7% accuracy as, by chance with 15 classes, 1 out of every 15 guesses is correct.
Data: The starter code trains and tests on 100 images from each category (i.e. 1500 training examples total and 1500 test cases total). In a real research paper, one would be expected to test performance on random splits of the data into training and test sets, but the starter code does not do this to ease debugging.
Evaluation and Visualization: The starter code function create_results_webpage() in create_results_webpage.py builds a confusion matrix and visualizes your classification decisions by producing a table of true positives, false positives, and false negatives as a webpage each time you run main.py. You do not need to edit this function.

3 Rubric

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.

- 5 pts: Build tiny image features for scene recognition. (get_tiny_images())
- 10 pts: Nearest neighbor classifier. (nn_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_words())
- 20 pts: Train 1-vs-all SVMs on your bag of words model. (svm_classify())
- 5 pts: Discussion of your design decisions and evaluation.
- 10 pts: Extra credit (up to ten points total)

4 Extra credit

For all extra credit, please include quantitative analysis showing the impact of the particular method you've implemented. Each item is "up to" some amount of points because trivial implementations may not be worthy of full extra credit. Most of the extra credit focuses on the final bag of words + SVM pipeline of the project, not the baseline tiny image and nearest neighbor methods.

Feature representation extra credit:

- 3 pts: Experiment with features at multiple scales. E.g., sampling features from different levels of a Gaussian pyramid.
- 5 pts: Add additional complementary features (e.g., GIST descriptors and self-similarity descriptors) and have the classifier consider them all.

Feature quantization and bag of words extra credit:

- 3 pts: Use "soft assignment" to assign visual words to histogram bins. Each visual word will cast a distance-weighted vote to multiple bins. This is called "kernel codebook encoding" by Chatfield et al..
- 5 pts: Use one of the more sophisticated feature encoding schemes analyzed in the comparative study of Chatfield et al. (GMM, Fisher, Super Vector, or LLC). scikit-learn offers a helpful function for GMM fitting: sklearn.mixture.GaussianMixture.fit().

Classifier extra credit:

- 3 pts: Train the SVM with more sophisticated kernels such as Gaussian/RBF, L1, or chi-sqr.
- 5 pts: Try and improve the nearest neighbor classifier to be competitive or better than the linear SVM using the method of Boiman, Schechtman, and Irani, CVPR 2008.

Spatial Pyramid representation and classifier:

- 4 pts: Add spatial information to your features by implementing the spatial pyramid feature representation described in Lazebnik et al 2006.
- 3 pts: Additionally implement the pyramid match kernel described in Lazebnik et al 2006.

Experimental design extra credit:

- 3 pts: 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.
- 3 pts: 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.
- 3 pts: Experiment with many different vocabulary sizes and report performance. E.g. 10, 20, 50, 100, 200, 400, 1000, 10000.
- 5 pts: Report performance on the 397-category SUN database. This involves more than 100x as many training and testing examples as the base project, so it is not trivial to do.

5 Notes

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 modern feature encoding methods for bag of words models, please see Chatfield et al. 2011.

Tiny Images and Nearest Neighbor Classification

Start by implementing the tiny image representation and the nearest neighbor classifier. They are (relatively) easy to understand, (relatively) easy to implement, and run very quickly for our experimental setup (less than 10 seconds). The tiny image feature, by Torralba et al., 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 et al. 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 of performance with which to compare more sophisticated representations. See get_tiny_images() for more details.
The nearest neighbor classifier is equally simple to understand. When tasked with classifying a test feature into a particular category, we simply find the "nearest" training example (L2 distance is a sufficient metric) and assigns to the test case the label of the nearest training example. The nearest neighbor classifier has many desirable features: it requires no training, it can represent arbitrarily complex decision boundaries, and it trivially supports multiclass problems. However, it is vulnerable to training noise, 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. See nearest_neighbor_classify() for more details.
Together, the tiny image representation and nearest neighbor classifier will achieve 15-25% accuracy on the 15 scene database. For comparison, chance performance is 7%.

Bag of Words

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 6.2 for more details.
After we have implemented a baseline scene recognition pipeline, we shall move on to a more sophisticated image representation: bags of quantized feature descriptors. Before we can represent our training and testing images as bags of feature descriptors, we first need to establish a vocabulary of visual words, which will represent the similar local regions across the images in our training database. We will form this vocabulary by clustering with k-means the many thousands of local feature vectors from our training set.
Note: This is not the same as finding the k-nearest neighbor feature descriptors! For example, given the training database, we might start by clustering our local feature vectors into k=50 clusters. The centroids of these clusters are our visual word vocabulary. For any new feature point we observe, we can find the nearest cluster to know its visual word. As 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. Be careful of this if you re-run with different parameters. 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 feature descriptors. Instead of storing hundreds of feature descriptors, we simply count how many feature descriptors fall into

each cluster in our visual word vocabulary. This is done by finding the nearest neighbor k-means centroid for every feature descriptor. Thus, if we have a vocabulary of 50 visual words, and we detect 220 features in an image, then our bag of words representation will be a histogram of 50 dimensions where each bin counts how many times a feature 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_words() for more details.
You should now measure how well your bag of words representation works when paired with a nearest neighbor classifier. There are many design decisions and free parameters for the bag of words representation (number of clusters, sampling density, sampling scales, feature descriptor parameters, etc.) so performance might vary from 50

Multi-class SVM

The last task is to train 1-vs-all linear SVMs to classify the bag of words feature space. Linear classifiers are a simple learning model. 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 words 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 need to 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. Luckily for us, sklearn.svm.LinearSVC() handles a lot of the heavy lifting for doing mutliclass classification for us in a single function call! This function will automatically infer that it needs to do multiclass classification if it's given an training dataset with multiple output labels. See svm_classify() for more details.
Now you can evaluate the bag of words representation paired with 1-vs-all linear SVMs. Accuracy should be from 50% to 60% depending on the parameters. You can do better still if you implement extra credit suggestions.

Computation Speed

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). Saving intermediate results (see usage of np.save and np.load in the stencil) can be really helpful for debugging different parts of the pipeline and prevent you from continually re-running really time consuming functions.

Credits

Adapted for python by Brendan Walsh and Michael Chen, with help from Jamie DeMaria and Anna Sabel. Original project description and code by James Hays and Sam Birch.