COMP3204 - Scene Classification

Kwong Chi Tam `kcts1g13` Edward Cater `ec7g13`

December 9, 2015

Implementation Decisions

We define classes Run1, Run2 and Run3, all of which inherit from a super class "`Run`" containing a number of useful fields and methods. This includes two `GroupedDataset` instances for training and test data respectively, int variables `nTest` and `nTraining` to record the size of each data set (purely for more readable code) and some helper methods for loading/splitting data into training and test data sets. The descriptions for these methods can be found in the code comments.

Testing the Accuracy

The labeled data provided by training.zip is randomly divided into training and set sets using `GroupRandomSplitter`. The training split is used to train the classifier and the test split is used to determine the accuracy.

Classifying Unlabeled Data

Rather than splitting our data, all available labeled data from training.zip is used to train the classifier. The fully trained classifier is then used to make predictions on the unlabeled data from testing.zip, which is then written to a file.

On the Contributions of Team Members

We took steps to ensure all three runs contained significant contributions from each team member. Ultimately, both team members contributed an equal amount of work.

1 K Nearest-Neighbours

For k nearest neighbor classification, first, each image is cropped about the center to a fixed resolution of 16 by 16 pixels. After cropping, the image is normalised and each row of pixels is concatenated into a float feature vector, where each element represents a single pixel's normalised grayscale value (all images are grayscale by default). This is our "tiny-image" feature vector.

1.1 Testing the accuracy of our classifier

Feature vectors for training and test datasets are stored in separate, two dimensional float arrays; "`trData[][]`" and "`tsData[][]`", where each row corresponds to an image's tiny-image feature vector. Both training and test set data have a corresponding array of strings; "`trClass[]`" and "`tsClass[]`" where each element is the class of the image represented by a row in `trData[][]` or `tsData[][]`.

An instance of OpenIMAJ's FloatNearestNeighboursExact is created, which accepts `trData[][]` in its constructor. For every row in tsData[][], we use the `FloatNearestNeighborsExact` object to retrieve a list of the nearest neighbors found in `trData[][]`, and a is prediction made according to the mode class from that list of neighbors. We compare the actual classification of that instance of training data against the prediction and record whether or not we are correct. Once a prediction has been made for all instances of training data, we calculate the accuracy. To tune this classifier, an average of five runs was computed for k values ranging from 1-19. What was discovered that looking at only one neighbor resulted in the best classifier. Although this was still very low (approximately 21% for $k = 1$).

1.2 Classifier Configuration

Variable	Value
k	1

2 Dense Patch Sampling

In order to implement a linear classifier using the `LiblinearAnnotator`, we designed our own feature extractor that made use of a custom engine, which is responsible for sampling and normalising 8x8 patches every 4 pixels in the x and y directions. The engine inherits from OpenIMAJ's Engine class and overrides the `findFeatures()` method. This returns a `LocalFeatureList` of `FloatKeypoints`. To construct these `FloatKeypoints`, the pixel values of every element in all row within in a patch are concatenated into a single row feature vector and passed to the `FloatKeypoint` constructor along with the position of the feature. Rather than returning all the features found by the engine, the `findFeatures()` method returns a subset of key points, obtained by random selection.

The `FeatureExtractor` requires the use of a `HardAssigner` in order to generate the bag-of-visual-words. This extracts features from the densely sampled pixel patches and uses K-Means to cluster them into separate classes. Initially, the suggested number of ~ 500 clusters was used, however after testing the classifier with different numbers of clusters, it was found that using ~ 600 clusters maximized the accuracy.

Once the `HardAssigner` has been trained, the `FeatureExtractor` uses a `BlockSpatialAggregator` to append together the spatial histograms computed from the collection of visual words. The result of this is normalized and returned in the form of a feature vector.

2.1 Testing the accuracy of our classifier

An instance of `LiblinearAnnotator` is constructed using the custom feature extractor and trained using a subset of the images from training.zip data set. Internally, the data set is converted to a list containing exactly one reference to each object in the data set with (potentially) multiple annotations.

The annotator is passed to a `ClassificationEvaluator`. Calling its evaluate method retrieves the predictions in the form of a Map, and calling its analyze method calculates the accuracy and error of the classifier.

This classifier required extensive training and tuning due to the number of parameters that can be changed. Because it is so computationally expensive, the training phase can take anywhere between 15-30 minutes and therefore only a small subset from the training.zip was split into training and testing datasets. Specifically, only 5 out of the 15 scenes were used for our tuning purposes, retrieving 50 images per group for the training set and 10

images per group for the testing set. It is worth noting that accuracies may seem higher than expected due to the smaller-sized datasets being used for tuning.

- The number of clusters was increased by 100 each time, but from our results, we realised that having more than 600 clusters provided no valuable contribution to the accuracy.
- After experimenting with different step sizes, we found out that smaller values increased the accuracy of the classifier. However, this was payed with much slower performance speeds. Therefore, we considered a step size of 3 to be a reasonable balance between accuracy and performance.
- The specification suggests to start by sampling 8x8 pixel patches, so we decided to increase the bin size to figure out what effect it would have on the overall accuracy. The results show that there is no obvious pattern involving this parameter. However, using 14x14 pixel patches seemed to give the best accuracy.

2.2 Classifier Configuration

After training and tuning our classifier by modifying the mentioned parameters, we concluded that the following configuration is reasonable for our purposes, managing to achieve approximately 30% accuracy on a training set composed of 90 images per group and testing set composed of 10 images per group.

Variable	**Value**
Step size	3
Patch size	14
Number of clusters	600

3 Pyramid Dense SIFT with Homogeneous Kernel Map

For the purpose of scene classification, using SIFT descriptors for patches within regular grid spaces has shown to be more suitable than descriptors of local interest points [2, 1]. Knowing this, we opted to use a dense SIFT engine to compute and describe a collection of 8x8 patches taken at a stepsize of 3 pixels. From the patches, a subset is taken by random selection, whose SIFT descriptors are used to generate visual bags of words. A `PyramidSpatialAggregator` is used rather than the `BlockSpatialAggregator` for computing the spatial histograms across the image. Additionally, by applying a homogeneous kernel map, the non-linearly separable data is projected into space where it can be analyzed by a linear classifier. This is a highly accurate approximation of applying a nonlinear classifier.

Rather than just using dense SIFT, this classifier makes use of spatial pyramid matching, which is regarded to be "one of the most successful methods in computer vision"[1]. The `PyramidDenseSIFT` class within OpenIMAJ takes a normal `DenseSIFT` instance and applies it to different sized windows on the regular sampling grid.

The feature extractor for this classifier uses a `PyramidSpatialAggregator` to append the spatial histograms computed from the collection of visual words, returning the aggregated results in the form of a single vector. Unlike `SpatialBlockAggregator`, this approach groups the local features into fixed-size spatial blocks within a pyramid.

3.1 Training and tuning

3.2 Classifier Configuration

Variable	**Value**
Step size	3
Patch size	8
PyramidDenseSIFT scales	4, 6, 8, 10
Number of clusters	600
PyramidSpatialAggregator numBlocks	2, 4

Appendix

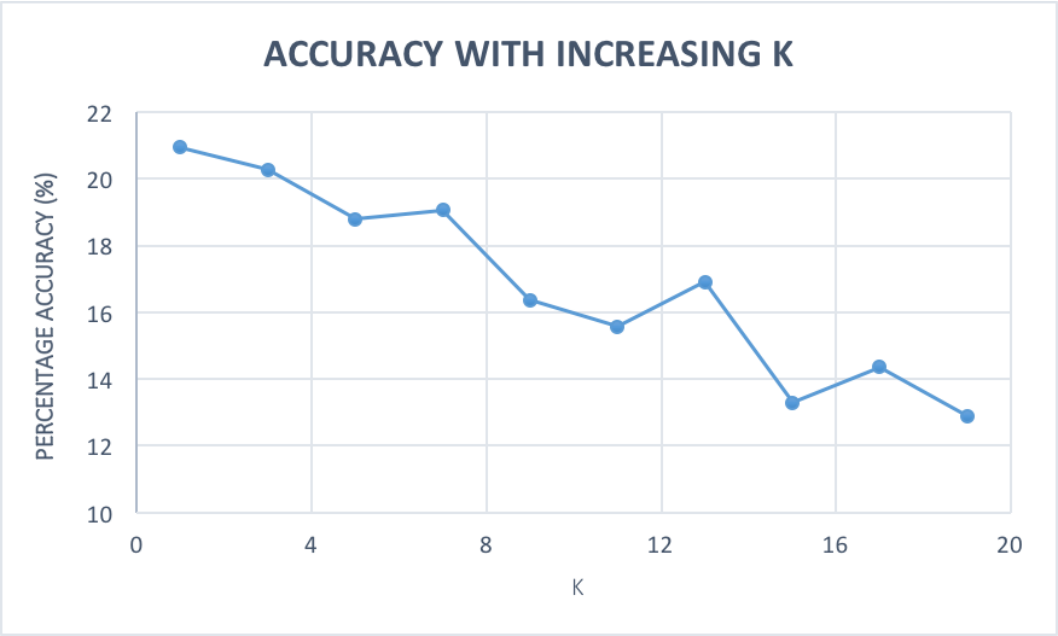

Figure 1: Accuracy with increasing k

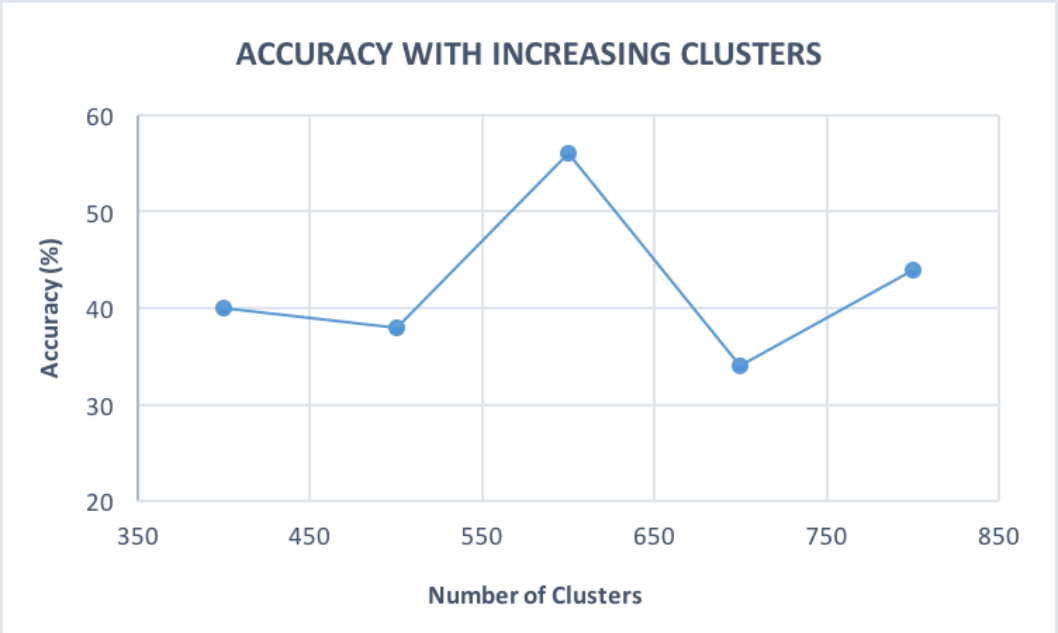

Figure 2: Accuracy with increasing number of clusters

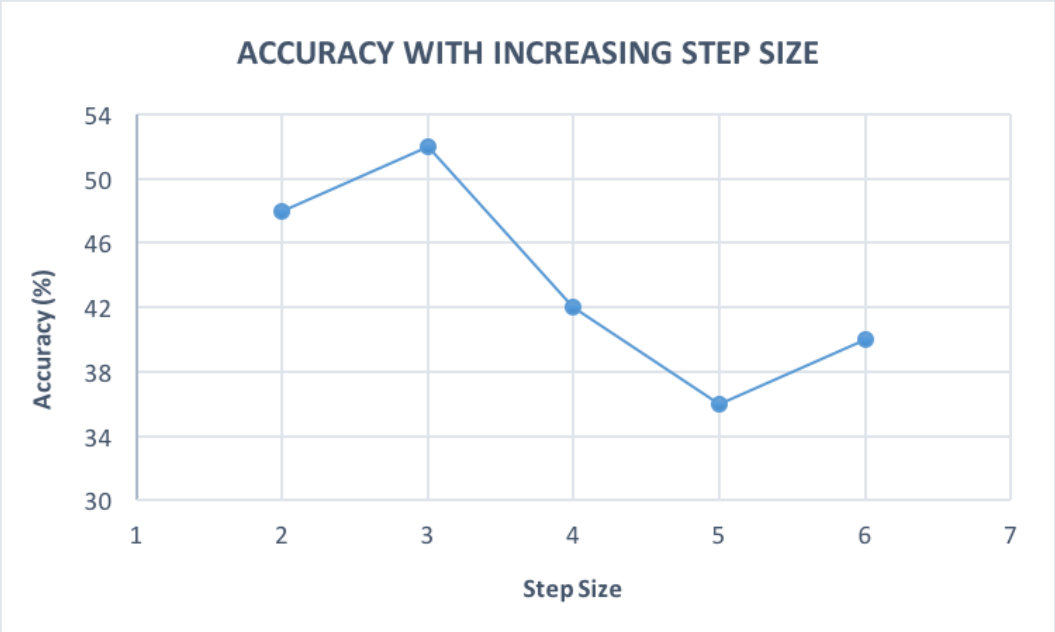

Figure 3: Accuracy with increasing step size

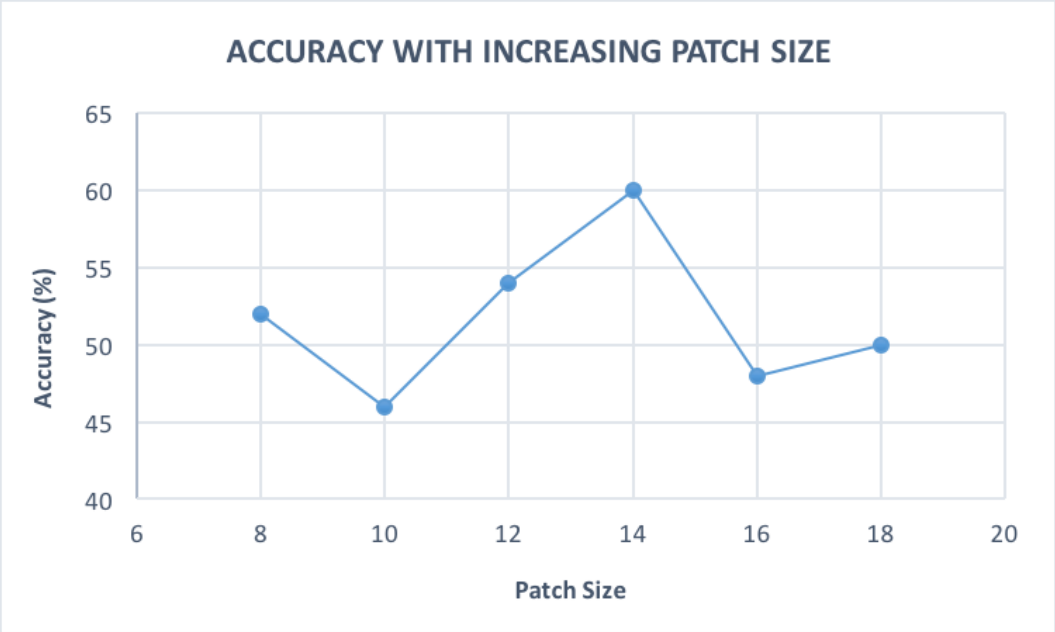

Figure 4: Accuracy with increasing patch size

References

[1] Li Fei-Fei and Pietro Perona. A bayesian hierarchical model for learning natural scene categories. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pages 524–531. IEEE, 2005.

[2] Svetlana Lazebnik, Cordelia Schmid, and Jean Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 2, pages 2169–2178. IEEE, 2006.