COMP3204 Computer Vision

Coursework 3: Scene Recognition

Team 33:

cc14g17 Callum Connolly

khl1g17 Kwok Hung Liu

pd2g17 Peter Deng

Run #1: K-Nearest Neighbour and Tiny Image

Firstly, we identified the classes which were needed for the code, the Main class (App.java), the K-Nearest Neighbour Classifier class (KNNClassifier.java) and a Tiny Image class (TinyImage.java).

App.java handles the data. While the training dataset is downloaded from the specification straight away using VFSGroupDataset, the test data are downloaded straight into the /data directory because the output of each image must include its respective file name which cannot be done with VFSGroupDataset (or at least we did not managed to do so). After downloading the data, the set “training” is removed as it is the parent folder which contains all the folders for the different categories. The training dataset was then fed into an instance of the KNNClassifier class along the size of the tiny images (size) using the function addTrainingData (VFSGroupDataset, int).

In the KNNClassifier class, objects of TinyImage are created from the images from the dataset and a HashMap of vector values (FloatFV) and the category names (String) are stored.

We sorted the files in the testing data folder so the output will be in numeric order. We classify each image using the instance of the classifier class, turning them into a square tiny image before calculating a vector value. The vector value will then be used to be compared to the values stored in the HashSet (categories) where the difference will be stored in a priority queue. The first k-nearest values will then be used to retrieve the original FloatFV values which are the keys of the HashSet. Using the keys retrieved, a majority voting process will take place to determine the likely category of each image. If there are more than one categories that have the same number of votes, the output will join them with the keyword “and”.

Finally, the output will be processed and stored in a text file named “run1.txt”.

The time taken for run #1 to execute is about 30 seconds to a minute.

The accuracy of the program is pretty debatable as some images have up to 3 to 4 possible guesses which are quite distinct from one another.

Run #2: Bag of Visual Words Classifier

When we first approached run #2, we tried to do it in a similar fashion to run #1 which did not work out in our favour as we realised specific methods and classes must be implemented and used. As a result, we made it in a similar style to the Tutorial of Chapter 12.

For Run #2, we have implemented a main class (App.class) which contains a class to extract patches (PatchExtractor.class) and a class to extract the bag of visual words (BOVWExtractor.class).

Firstly, we create an instance of the PatchExtractor object. Using a HardAssigner we are able to assign these features to its respective category name from the training set. The LiblinearAnnotator will then able to take both the training dataset and the PatchExtractor to train itself to identify these features from patches of an image.

Same with Run #1 we sorted the images of the testing dataset in order so they will be displayed in numeric order for the output. Using the Annotator.annotate(FImage) method, we can get an output in the form of List<ScoredAnnotation<String>> which includes the “guess” we want. As a result we converted into a list and extract the substring from it.

The time taken for Run #2 to execute is about 10-30 minutes varying from different processor power. The parts which creates the assigner and uses the training dataset to train the liblinear annotator contribute the most to the time taken when running the program. While the program is running, it uses about 1.7GB of memory. Once the training is completed, each image will take about 1-2 seconds to generate an output (00.jpg “guess”).

The accuracy of the program is way better than run #1 but still has a lot of room for improvement. The program mixes up the bedroom-livingroom pair, the OpenCountry-Coast pair, the store-kitchen pair and the Insidecity-TallBuilding pair quite often.

Run #3 – Back to DenseSIFT with added stuff

For Run 3, we use DenseSIFT, specifically PyramidDenseSIFT from chapter 12 of the OpenIMAJ Tutorial as our feature extractor. As we remembered from the tutorial, it is the most accurate feature extractor that we are able to implement. Similarly to Run #2, we used a HardAssigner but this time passing an instance of the PyramidDenseSIFT<FImage> object instead of a PatchExtractor object. In addition, we have added a HomogeneousKernelMap when creating the extractor. We then pass the new extractor to the LiblinearAnnotator which allows the program to "train" itself with the training dataset

Within the extractFeature function of the feature extractor (PHOWExtractor), we have added a PyramidSpatialAggregator instead of just aggregating the values normally.

Same with the previous runs, the images of the testing dataset have to be sorted in order so I used the same code for the previous two runs before passing the data to the annotator to determine what kind of images they are.

The time taken for Run #3 to execute is about 30-90 minutes varying from different processor power. It also uses about 2.5 GB of memory while processing the features (~10000000 total features generated for run #3). Once the training is completed, each image will take about 1-2 seconds to generate an output (00.jpg "guess").

The accuracy of the program improved a little bit (from run #2) but sometimes it still manages to mix up some categories, notably the bedroom-livingroom pair and the Opencountry-Coast pair.

Contributions

Callum - Coding and testing of Run #1. Setting up, organisation of source control. Editing and proofreading of the report

Jackey (Kwok Hung) - Coding and testing of Runs #1, #2 and #3. Writing, editing and proofreading of the report.

Peter - Testing of Runs #1, #2 and #3. Contribution to writing, editing and proofreading of the report