School of Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

Coursework 3: Scene Recognition

COMP3204 Computer Vision

Anastasov *Velimir N (vna1u19@soton.ac.uk)*
Daud *Dzhani S (dsd1u19@soton.ac.uk)*
Kavak *Dzhem T (dtk1u19@soton.ac.uk)*
Powell *Charles (cp6g18@soton.ac.uk)*
Sobczak *Konrad K (kks1g19@soton.ac.uk)*

16th December 2021

Contents

Introduction

This report provides documentation of three scene recognition classifiers developed using the OpenIMAJ framework. These classifiers are a **K-Nearest Neighbour Classifier**, a **set of Linear Classifiers**, and a **Naive Bayes Classifier**. For each classifier, descriptions of its implementation and the implementation of its feature extraction process are provided, along with justifications for any the necessary parameters. All three classifier implementations are extensions of the `MyClassifier` class, which defines some basic methods that can be used in the general classification process. A description of this general classification process is also provided.

1 Classifier Descriptions

1.1 Classifier 1: K-Nearest Neighbour

1.1.1 Overview

The K-Nearest Neighbour Classifier was implemented in the `KNNClassifier` class using the native OpenIMAJ `KNNAnnotator` class with a custom "Tiny Image" feature extractor and Euclidean space as the distance metric. During testing with the provided training data, the classifier was able to achieve an average precision of 0.33-0.34 for an optimal value of K. Instances of `KNNClassifier` are defined based on the following parameters:

- `k` : The value of K for the K-Nearest Neighbour Algorithm (`int`).
- `tinyImageRes` : The resolution of the tiny image features (`int`).
- `trainingData` : The dataset to be used for training the classifier (`GroupedDataset`).

When instantiated, the `train()` method of the `KNNAnnotator` class is called to fit the classifier to the provided training data.

1.1.2 Feature Extraction

The K-Nearest Neighbour classifier annotates images based on "Tiny Image" features. These features are squared versions of the original image reduced to a small, fixed resolution. The "Tiny Image" feature extractor is implemented within the `TinyImageFeatureExtractor` class. The `TinyImageFeatureExtractor` class implements the OpenIMAJ `FeatureExtractor` interface, and is instantiated with an integer parameter which determines the resolution of the tiny images extracted as features. Implementation for the feature extractor is provided within the `extractFeature()` method and can be broken down into three parts:

- The image is cropped to a square the size of it's original width using the `FImage.extractCenter()` method.
- The image is resized to the provided tiny image resolution using the OpenIMAJ `ResizeProcessor`.
- A single feature vector is formed by concatenating the rows of the resulting 'tiny' image.

Note that, while implementation for the mean centering and normalisation of the 'tiny image' was provided, this logic was not used in the final run of the classifier as it was not found to improve the classifier's performance.

1.1.3 Parameter Choices

The two parameters to be chosen for the K-Nearest Neighbour Classifier are the value of 'K', and the resolution of the 'tiny image' features. A value of 16 was used for the resolution of the tiny images following the recommendation of the specification, while the optimal value of K was found to be in the range of 14-19 following testing of the classifier with a range of K values (1 to 30) on the provided training data. The results of this testing are given within the report appendix (Classifier Testing Results)

1.2 Classifier 2: Linear Classifiers

1.2.1 Overview

For the second run, we developed a set of linear classifiers using OpenIMAJ's `LiblinearAnnotator`. What's more, we encapsulated Run 2's classifier into our own `LibLinearClassifier` class. We also implemented a custom "Densely-Sampled Pixel Patches" feature extractor, which we will refer to as DSPP. A sample of the training data was used to train a `K-Means Clustering` classifier. This way we obtained a dictionary of visual words, which was needed to compute each image's **bag-of-visual-words** (BoW). A BoW is simply a histogram of the occurrences of each visual-word in an image. The bags are essentially numerical vectors, which we can use in the `LiblinearAnnotator` to classify images based on the visual-words they contain.

The initial accuracy obtained when first running the classifier was in the range of 0.32 - 0.33. The application of normalisation and mean-centering to each DSPP patch improved this accuracy to 0.49-0.50. By optimising the value of K (number of clusters) to 1200, a final accuracy of 0.68-0.69 was achieved. To initialise a `LibLinearClassifier` object, the following parameters must be provided:

- `trainingData` : The dataset to be used for training the classifier (`GroupedDataset`).
- `patchSize` : The size of each DSPP patch (`int`).
- `patchEvery` : The number of pixels that we skip after extracting a DSPP patch (`int`).
- `k` : The value of K for the K-Means Clustering algorithm (`int`).
- `sampleSize` : The sample size that we use to train a `HardAssigner`, which is used as when mapping a set of patches to a BoW. (`int`).

When instantiated, the `train()` method of the `LiblinearAnnotator` class is called to fit the classifier to the given training set.

1.2.2 Feature Extraction

To extract the key points of a given image, we first scan it in a raster fashion, storing an 8x8 patch of pixels every 4 pixels. Of course, these parameters can be changed depending on the training set, but we found that they worked rather well for this run. As we discussed before, we apply normalisation and mean-centering to each patch, which improved the accuracy of our classifier. Finally, we aggregate the different extracted patches into a bag-of-visual words. This way, we can represent an image by a single feature vector based on the visual-words it contains instead of as a matrix of pixels.

We implemented our `DSPPExtractor` feature extractor class by inheriting OpenIMAJ's native `FeatureExtractor`. The class's constructor takes a `HardAssigner` as input. The `HardAssigner` is used when transforming the collected patches into a histogram of visual words. To obtain one, we train it using `K-Means Clustering` on a sample of the training data. More specifically, we extract the patches from each sampled image and group them into 1200 different clusters. The `extractFeature()` method of the `DSPPExtractor` class contains the logic for mapping an image into its histogram of words.

1.2.3 Parameter Choices

We tried tuning in different parameters to maximise the accuracy of our classifier. After changing the DSPP's `patchSize` and `sampleEvery` parameters, we observed that the resulting accuracy did not make a significant difference, and in some cases it did not change at all.

Since the testing dataset that was provided to us was unlabeled, we had to sample a part of our training set for testing purposes. The ratio that we used was 8:2 (training:testing), but we also experimented with 6:4 and 4:6, which only decreased the final accuracy.

Finally, we discovered that by increasing the number of clusters for the `K-Means clustering` algorithm the accuracy significantly improved. The effect of changing the number of clusters on the classifier performance is illustrated within the report appendix (Classifier Testing Results)

1.3 Classifier 3: Custom Classifier

1.3.1 Overview

For the third run we experimented with 3 different classifiers - **Support Vector Machine (SVM)**, **Linear Annotators** and **Naive Bayes**. When testing, all three made use of a Pyramid Histogram of Words & Dense SIFT feature extractor and a K-Means Clustering algorithm to obtain a vocabulary of visual words. After comparing all three, we observed that the Naive Bayes classifier gave the best accuracy of **0.69-0.70**. For comparison, the SVM and LiblinearAnnotator both provided accuracies of 0.49-0.55.

The final classifier was implemented in the `NaiveBayesClassifier` class, and makes use of OpenIMAJ's `NaiveBayesAnnotator` class. The class constructor requires the following parameters:

- `trainingData` : The dataset to be used for training the classifier (`GroupedDataset`).
- `k` : The value of K for the K-Means Clustering algorithm (`int`).
- `sampleSize` : The sample size that we use to train a `HardAssigner`, which is used as when mapping a set of patches to a BoW. (`int`).

When instantiated, the `train()` method of the `NaiveBayesAnnotator` class is called to fit the classifier to the given training set.

1.3.2 Feature Extraction

Since Run 2's simple feature extraction algorithm proved to be very efficient at describing the key points in an image, we decided to build upon it by replacing the DDSP patches with Dense SIFT descriptors. The reason why we choose to use Dense SIFT and not SIFT is that the former provides us with more coverage of the entire image, whereas the latter is only focused on Difference-of-Gaussian points of interest.

Nevertheless, both SIFT and Dense SIFT are not only invariant to changes in scale and rotation, but are also robust to noise and illumination. Because of that, they are widely used and applied in different areas such as feature matching, 3D modeling, object recognition etc. We can vector quantise Dense SIFT descriptors into visual words. This way we can sample a certain portion of our training set and use it to train a K-Means Clustering classifier to obtain a dictionary of visual words. The concept of BoWs was also applied in Run 2, where we explored it in more detail. Here we will use the same logic, but we expect better results as our descriptors are way more expressive.

It is important to note that BoWs are limited in the sense that they do not account for the spatial locations of occurrences of the visual words. To fix that, we can use a Pyramid Histogram of Words to build spatial histograms of the visual word occurrences. Spacial pyramid matching splits an image into different regions and weights the histograms from each level, paying more attention to deeper levels as they are more spatially focused [1].

1.3.3 Attempted Classifiers

SVM Classifier

After researching different sources online, we learned how powerful and widely-used SVMs are in the real world. However, when we initially implemented the algorithm, it seemed to be extremely inefficient - the first ever accuracy that we obtained was 0. After adjusting the classifier parameters, namely the number of clusters for the K-Means algorithm and the train:test ratio, we managed to reach an accuracy of approximately 0.2. This is when we realised that the SVM classifier had produced such poor accuracy because a standard SVM assumes that the feature space is linearly separable, which was not the case in this problem. To fix that we wrapped our feature extractor in a `HomogeneousKernelMap`, which transformed the data into a linearly separable format. As a result, the accuracy got to approximately 0.55. Whilst this accuracy is a significant improvement on the original implementation, the Naive Bayes classifier was still found to provide better performance.

LiblinearAnnotator Classifier

The second classifier that we tested was OpenIMAJ's `LiblinearAnnotator`. Here Chapter 12 from the OpenIMAJ tutorials was particularly helpful as it gave us insight into building a "near state-of-the-art image classifier". However, after numerous attempts we could not manage to achieve an accuracy greater than 0.54. We tried out different parameter values to see if we could further improve it, but we did not observe any significant changes. In fact, even increasing the number of clusters within the K-Means classifier (something which worked rather well in the previous runs) failed to change the final accuracy. We tried values for K of 200, 400, 600 and 1200, which all resulted in an accuracy of 0.51 - 0.54. For this reason, we decided to give up on this approach and search for a better alternative.

1.3.4 Final Classifier: Naive Bayes Classifier

After completing the testing described above, a Naive Bayes classifier, implemented with OpenIMAJ's `NaiveBayesAnnotator`, was chosen for the third run. This classifier is provided within the `NaiveBayesClassifier` class. Instances of the classifier take in two parameters; the feature extractor and a `NaiveBayesAnnotator.Mode`. Our implementation makes use of a pyramid histogram of words & dense SIFT feature extractor, and the `MAXIMUM_LIKLIHOOD` annotation mode, which returns only the most likely classification for a given image.

The Naive Bayes classifier is based on Bayes' Theorem [2, 3]. Bayes' Theorem states that if we have two events, we can calculate the probability of the first occurring given the second has occurred with $P(F|S) = P(F) * P(S|B)/P(S)$. It is known as "naive" because calculations for multiple features are difficult as they can cause a zero probability problem. To prevent this, we must assume that all features are independent of each other. This means that $P(F|S1, S2) = P(F) * P(S1|F) * P(S2|F)/P(S1, S2)$. Using this formula, we can calculate the probability of a scene belonging to a specific class and thus assign it to the class with the highest probability. While assuming independence doesn't seem like a sound principle, the Naive Bayes classifier was able to provide the best performance (roughly 0.7) of all of the tested classifiers. After experimenting with the value of K for the K-Means Clustering algorithm, a value of 150 was found to be optimal. The results of this testing are provided within the report appendix (Classifier Testing Results)

1.4 Overview of Classification Process

The `MyClassifier` class defines a `makeGuesses()` method which can be used to assign labels to dataset using the underlying classifier instance (e.g., instance of one of the three classifier implementations). The `makeGuesses()` method takes in the dataset to be classified (`VFSListDataset`) as input, and returns a list of tuples (`ArrayList<Tuple<String,String>>`), where the first element of a tuple is the name of the image being classified (i.e., filename), and the second element is the classification given to this image.

To annotate the provided dataset, the `makeGuesses()` method iterates over the images contained within it, and for each one, calls the `annotate()` method of the underlying classifier, passing in the image as an input parameter. The `annotate()` method returns a list of possible labels and associated scores, which is then iterated over to find the best possible annotation for the input image. The name of the image and this "best annotation" are then appended as a tuple to a list of annotations which is returned once all images have been classified.

2 Group Contributions

The below table summarises the contributions of each member of the group towards this project, reflecting the even spread of workload that was achieved.

Group Member (ID)	Contributions
Dzhem Kavak (dtk1u19)	• Worked on the implementation of the Liblinear classifier in Run 3 • Tested different parameter values to maximise accuracy for Run 3. • Worked on report of the third classifier.
Charles Powell (cp6g18)	• Implementation of K-Nearest Neighbour Classifier. • Documentation for K-Nearest Neighbour Classifier. • Proof read and amended report. • Implementation of a program for the testing and running (i.e., classifying testing images and writing classifications to a file) of all three classifiers. • Tested different parameter values for the SVM and LiblinearAnnotator classifiers in Run 3
Velimir Anastasov (vna1u19)	• Established and maintained Discord sever for group communication. • Conducted research of possible features/feature extractors and classifier to be used in the third classifier. • Worked on implementation of third classifier. • Worked on the documentation of the third classifier.
Dzhani Daud (dsd1u19)	• Worked on the implementation of Run 2 • Worked on the implementation of the SVM approach in Run 3 • Worked on the report for Run 2 • Tested different parameter values to maximise accuracy for Run 2.
Konrad Sobczak (kks1g19)	• Created and maintained the Git repositories including merges and minor fixes • Worked alongside Dzhani Daud on Run 2 • Assisted Velimir Anastasov and Dzhem Kavak on Run 3's PHOWExtractor • Refactored and maintained the code for the project

References

[1] Lazebnik, Svetlana & Schmid, Cordelia & Ponce, J.. (2006). Beyond Bags of Features: Spatial Pyramid Matching for Recognizing Natural Scene Categories. In CVPR. 2. 2169 - 2178. 10.1109/CVPR.2006.68.

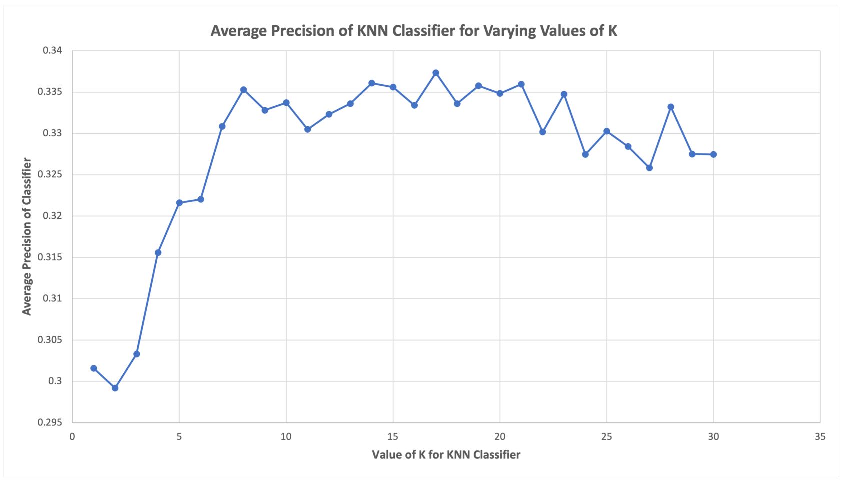
[2] Rish, I., 2001, August. An empirical study of the naive Bayes classifier. In IJCAI 2001 workshop on empirical methods in artificial intelligence (Vol. 3, No. 22, pp. 41-46).

[3] Berrar, D., 2018. Bayes' theorem and naive Bayes classifier. Encyclopedia of Bioinformatics and Computational Biology: ABC of Bioinformatics; Elsevier Science Publisher: Amsterdam, The Netherlands, pp.403-412.

A Classifier Testing Results

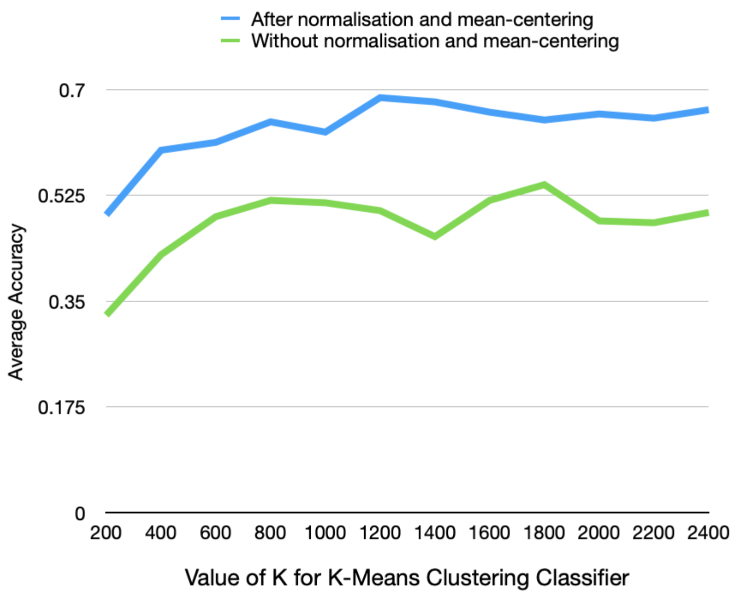
A.1 Run 1: K-Nearest Neighbour Classifier Testing

The below graph shows the average precision achieved when testing the K-Nearest Neighbour Classifier developed for Run 1 on the provided training data with a range of K values. Values 1 through to 30 were used, with each value of K being trialed 50 times, and an average of the performance score taken as the final result for this value of K.



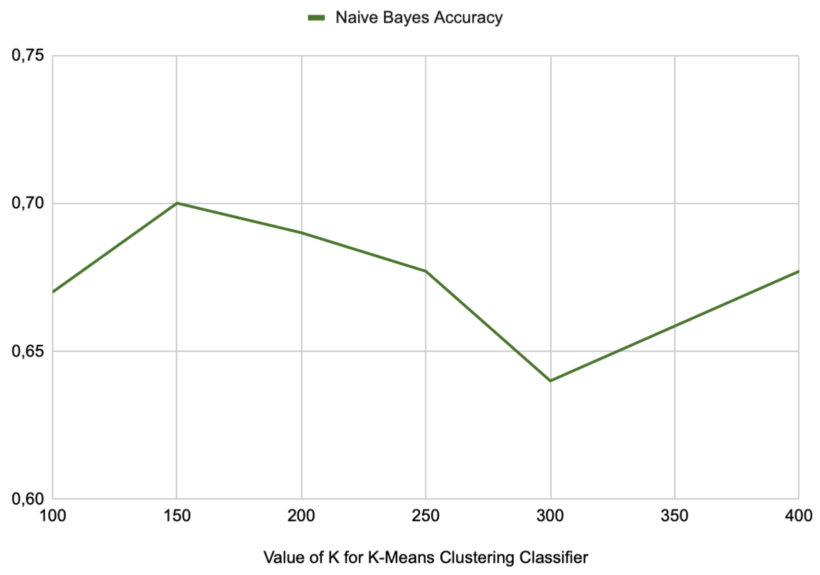
A.2 Run 2: K-Means Clustering Classifier Testing

The below graph illustrates the average accuracy that was obtained when testing the K-Means Clustering classifier with different values for K in Run 2. As you can see, an optimal performance was achieved when K is equal to 1200. The graph also shows the benefits of applying normalisation and mean-centering.



A.3 Run 3: K-Means Clustering Classifier Testing

The below graph shows the average accuracy that was obtained when testing the K-Means Clustering classifier with different values for K in Run 3. As you can see, an optimal performance of approximately 0.7 was achieved when K is equal to 150.


Naive Bayes Accuracy
Value of K for K-Means Clustering Classifier