Image Indexing and Query by Example

2nd Programming Task
Multimedia Retrieval
SoSe2014

*

In this document we introduce the assigned task and our solution. Following this we present the class that contains our implementation: it's elements and functions. We include a simple User Guide and conclude this document with an Appendix where, with a basis on provided code and images for testing, we discuss the features extracted by our implementation

*

The task:

Implement a simple Image Retrieval System, which makes use of information arising from dye distribution in images.
Furthermore, the following goals should be achieved:

- Setting up a feature extraction scheme (processing the raw image files & producing numeric descriptors capturing specific visual characteristics).
- Storing an index using these features for later use.
- Providing the functionality of "Query by Example" by using the created index and a similarity measure.

*

Our solution:

According to its context within the Multimedia Retrieval subject, the presented task corresponds to experimenting with aspects of a modern search engine, focusing particularly on handling images for content-based image retrieval. As opposed to annotation or keyword-based retrieval.

For this type of retrieval it's possible to do a search by sketching features, by placing icons for relative position comparison, and also by similarity searching. The later is the focus of our task.

As we discussed in class, a search engine could be modeled to consist of a crawler, an indexer and a search interface. In this task we focused specifically on adapting and using the indexer and search interfaces, as provided by functionality from Lire and Lucene libraries.

To utilize the index we provided support for **2 phases**: the **Build index phase**, when the index is created and stored in a user-selected folder. And the **Search phase**, when (only after creating the index) searches can be performed.

Our implementation makes use of 3 particular open-sourced libraries:

- Lire
- Lucene
- JavaFx, for the GUI.

Lire (http://www.semanticmetadata.net) is an extension of the Apache Lucene library, which was used for the previous programming task. Its name stands for Lucene Image Retrieval and, as this suggests, it allows the creation of Lucene indexes for images, based on extracting features from these images. Among the features which can be used, the library includes some which are compliant with the MPEG-7 standard discussed in class, such as Scalable Color, Color Layout and Edge Histogram. Additional features available include: Color histograms (HSV and RGB), Tamura, Gabor, auto color correlogram and JPEG coefficient histograms. There is also support for so called Compact Composite Descriptors such as CEDD and FCTH (http://savvash.blogspot.de/) and Visual bag of words schemes based on SIFT and SURF analysis

In our implementation we allow to use a selection of 10 of these features. These are further discussed in an Appendix, at the end section of the User guide.

From the Lire libraries we particularly made use of 3 classes:

- **DocumentBuilder:** This type of object embodies the algorithms and processing required to carry out feature extraction, and thus turn a given image file into an entity indexable by Lucene, more specifically, a specialized Document object from the Lucene library.

- **ImageSearcher:** Includes the processing required for calculating similarity measures (also called feature distances), for a given feature, between a file passed for query by example, and each entry from a loaded index passed as parameter.
- **ImageSearchHits:** Which contains an array of search results and similarity measures over a collection of these documents.

As was presented in the previous assignment, **Apache Lucene** provides a java-based indexing and searching framework.

From this framework we made use of the **Document** class, which allows to create objects, each one representing a particular file, according to the indexing domain. We also made use of its functions for the storage and retrieval of the created index. This functionality is mostly embodied in its **IndexReader** and **IndexWriter** classes.

*

Class elements:

The implementation of this assignment is contained in a java class called **ImageIndexer.**

As a design decision, our implemented class is function-oriented, therefore it is constituted only by GUI and IO related variables and there are no per-class variables related to the indexing or retrieval (apart from variables holding the paths of folders in use).

The required business layer objects are created and stored on-the-fly, from within the calling functions.

*

Class functions:

The class presented here is composed of the following functions:

1. **public static void main (String [])**: In charge of launching the application.
2. **public void start (final Stage)**: Responsible for initializing the GUI and setting some of the business logic which can be established from within the interface.

Functions used for the Build Index phase:

3. **public void selectImagesDirectory(File):** This is called from within the GUI and it implements the business logic of selecting a folder of images to be indexed.
4. **public void buildAndStoreIndex():** It implements the core of the index building process, assuming valid folders have been selected for indexing and for storing the index.

This function starts from 3 objects: a **Java array of all images** found in the selected folder, a **Lucene IndexWriter** (responsible for storing the index) and an object of the class **Lire ChainedDocumentBuilder**. This type of object serves as a sort of container for other document builders, each one supporting the extraction of a specific feature. When it is used, all of the independent features that are extracted from an image (by each document builder), are stored in a independent manner within a single Document object. In this way, the created Document can be later queried by using different features.

In this function, a loop is made over the array of images. For each one, a Document is created by using the ChainedDocumentBuilder. In this step, all features configured in the builder are extracted from the image (in our case, the 10 features selected). Then the Document is stored as an entry of the index, by the IndexWriter.

Functions used for the Search phase:

5. **public void queryByExample()**: This function assumes the proper building of the index, and a valid image passed as input. To begin with, it initializes 2 objects: a **Lucene IndexReader** that loads a stored index, and a **Lire ImageSearcher**, which is built by a **Lire ImageSearcherFactory**, containing the processing required to calculate similarity measures for querying over a given feature.

The search itself is performed over this searcher object, by passing it the selected image and the IndexReader. Its results are stored in an object of type **Lire ImageSearchHits**. This is basically an array ordered by similarity measures or scores. It stores those values, and maps them to the index entries, particularly the path of the file.

After this, the algorithm iterates over the results and displays them.

*

User Guide

Next to this documentation, you will find **5 folders**:

- **ImageIndexer-Jar file**: Containing a runnable jar file.
- **ImageIndexer-Source code:** Containing the source code of our implementation. This folder can be loaded to Eclipse as an existing Java project.

- **Sample Index folder:** To be used for storing the index.
- **Sample Images for indexing**
- **Sample Image to be used as query**

Portability Notes:
To use our program, it is necessary to have Java 8 installed. In particular, this is needed to run the attached runnable jar file.

An alternative mode of executing the program is to open it as an existing Eclipse Java project. For this it might be needed to manually add the referenced libraries (all of which are contained in the lib sufolder). If Java 8 is not available, an alternative to installing it is to download it and only add it to Eclipse by right-clicking on the JRE System Library folder in the Package Explorer. From there the properties can be configured to add this alternative JRE. By following these steps the code should be perfectly compilable and runnable.

In order to help with any difficulty for running the program, we can be contacted at: en.shadiakhras@yahoo.com and gabrielcampero@acm.org.

*

The program itself should be easy to use. At first the user can select a folder for storing the index and a folder of images to be indexed. Then she can click on the Build index button. Note that the index building might take some time, since several features are being extracted. In future versions, this might be improved.
Once the index is created, the user can click on the Search tab, input a image and then select among different features for the similarity search.

To conclude this document we will present with more detail the features made available by our implementation. We will discuss some of their characteristics while comparing them to results observed for the sample images provided.

*

Appendix:
Features extracted by our implementation

Feature extraction with Lire

Lire (http://www.itec.uni-klu.ac.at/lire/nightly/api/net/semanticmetadata/lire/) through its DocumentBuilderFactory and ImageSearcherFactory classes allows to create a matching feature extraction and similarity calculating scheme.
For this, the feature is first specified during the indexing, by adding a specific Builder to the Document Builder. An example of this can be done with the following command: builder.addBuilder(DocumentBuilderFactory.getScalableColorBuilder()).
Although it is possible to configure the builders, so the feature extraction is done with more customized parameters, in the current implementation this was not explored, instead each builder runs on default values (as defined in http://www.itec.uni-klu.ac.at/lire/nightly/api/constant-values.html).
During the searching phase, the ImageSearcherFactory class is used to create a searcher specific to the extracted feature. The only parameter that can be configured at this stage is the maximum number of hits expected during the search. By default we selected to input a value of 100. This can be changed by modifying the maxNumHits variable in our code.

Little documentation was found on the specifics of these implemented feature-extracting algorithms, nevertheless we consider that a quick analysis of their theoretical characteristics and the observed results might be of interest, serving as a concise recollection of our studies so far in the course, covering these topics .
Also, given that the focus of the homework was on dye distribution (i.e. color descriptors), we will focus our exposition on presenting them, giving less priority other types of decriptors.

Test case
Next to the documentation, we included a image folder containing 8 images:

- the famous Lena example,
- two images of birds with distinctive patterns in their feathers (one red and one blue),
- a cruise ship sailing calm yet wavy waters
- a white whale jumping on its back over a wavy sea
- two images of killer whales one in the ascending moment of a jump over calm waters, and the other in the descending moment of her junmp, over wavy waters.

Finally, as a query image we include a group of 5 killer whales showing their heads only, while performing some sort of acrobatic movement over wavy waters.

Given that we provide a small number of images, evaluating the feature extractors by precision or recall is not possible. Also, given that the query image has no specific semantic intention, the evaluation of the gap between semantics and low level descriptors is also impossible. We can however compare with a common sense color based ranking of the images according to similarity: In the highest positions we should find the killer whales, first the one in ascending jump. Following those we could expect to find the cruise (for it has more presence of the color black) or the other whale, and then the blue bird. The other examples should have small similarity to the query image, although Lena might be more likely than the red bird, by having a blue ornament in her hat.

Extracted features

Color descriptors:

Color histograms:
This implemented feature provides extraction and searching by a simple RGB color histogram, for global color distribution. The number of bins is configurable, and they are normalized to 8 bits per bin (values from 0 to 255).
The histogram could also be in HSV, HMMD and Luminance chrominance color spaces.
After calculating any histogram-based color descriptor there are several possible similarity measurements, such as the Minkowski-form metric: L1 distance (Histogram inersection), the L2 distance (Euclidean distance), Binary set Hamming distance, Histogram quadratic distance, Binary set quadratic distance, Histogram Mahalanobis distance, Histogram mean distance and Histogram moment distance.
For this particular feature, as well as others, the use of L1 and L2, and also of JSD and Tanimoto distances are available.
For the test data, the results for this feature differ from the expected in that the cruise image and Lena are given more priority, while the Blue bird is left behind. It is not easy to infer the reason for Lena given a relative priority over the Blue bird.

Scalable Color:
As explained in lectures and provided material, this is a global color descriptor, compliant with the MPEG7 standard, and based on a histogram on the Hue Saturation Value color space, with quantized bins. This histogram is then passed through a series of 1D Haar transforms, these lower the redundancy from the original image and can lead to better comparisons for similarity searching.
For the test data, the results for this feature are similar to the expected ones, even though with a slight imprecision for the blue bird takes a slight precedence over the cruise ship.

Color Layout :

This MPEG7 compliant feature, starts by dividing an image in YCbCr a grid, from there it calculates the representative colors, and uses a Discrete Cosine transform to produce the descriptor. With this it captures the spatial distribution of the representative colors. For the Lire implementation of the feature, the maximum color dimension (for selection of representative colors) is 102.
This feature achieved an optimal performance in our tests, fitting correctly the expected results for the test case.

Joint Histogram:
This feature is based on the independent calculation of different histograms based on diverse aspects, and then combining them. In the Lire implementation this is done with a 64-bin RGB representation, and using a pixel ranking scheme.
This feature displayed an optimal performance in our tests, fitting correctly the expected results for the test case.
This feature is not included in the MPEG7 standard.

Auto Color Correlogram:

This is the last of the color descriptors made available by our implementation. It is based on the AutoCorrelogram model, as described in Huang, J.; Kumar, S. R.; Mitra, M.; Zhu, W. & Zabih, R. (2007) "Image Indexing Using Color Correlograms", IEEE Computer Society. It was developed as a distilling approach for the spatial correlation of colors, with particular robustness to changes in appearance and shape due to camera angles and similar alterations.
This functionality is implemented by allowing to select among alternative proposed algorithms for the feature extraction, and configurable properties (such as modes of analysis) with different distance sets.
In our tests, this feature (used with default configuration) proved to be unsatisfactory for it gives priority to the cruise shio over the ascending whale, and to Lena over the blue bird.
This feature is not included in the MPEG7 standard.

Texture descriptors:

Gabor:
This feature is based on the use of Gabor filters, which can be considered as orientation and scale-tunable edge and line detectors. It characterizes texture orientation and scales.
Similarity measures for comparing texture descriptors include Euclidean distance, Mahalanobis distance and the L1 distance.
For our test case, the results are quite surprising, but not unexplainable. It gives priority first to the blue bird, then to the descending whale (maybe because of the wavy waters), then to the

red bird (maybe because of the background texture), then to the white whale with wavy waters, and then to the other killer whale (in aclm waters) and the cruise. Lena comes at last.

Tamura:

This is another texture descriptor, developed by Tamura, Mori and Yamawaki focusing on coarseness, contrast, directionality, line-likeness, regularity and roughness for characterising textures.
The result is also quite peculiar for our test case, which can be expected for textures seem to capture much less of the expected semantics than do color descriptors.

JPEG Coefficient Histograms:

Yet another texture descriptor. It is based on the comparison of DCT coefficients. A performance similar to other texture descriptors was observed for the test case.

Compact Composite Descriptors (using both color and texture features):

CEDD:

The CEDD feature was created, implemented and provided to Lire by Savvas A. Chatzichristofis. More information can be found in: Savvas A. Chatzichristofis and Yiannis S. Boutalis, *CEDD: Color and Edge Directivity Descriptor. A Compact Descriptor for Image Indexing and Retrieval*, A. Gasteratos, M. Vincze, and J.K. Tsotsos (Eds.): ICVS 2008, LNCS 5008, pp. 312-322, 2008.
Surprisingly good performance was observed for the test case, comparable to a slight improvement over the color histogram.
This feature is not part of the MPEG7 standard.

FCTH :

The FCTH feature was created, implemented and provided by Savvas A. Chatzichristofis. More information can be found in: Savvas A. Chatzichristofis and Yiannis S. Boutalis, *FCTH: Fuzzy Color and Texture Histogram - A Low Level Feature for Accurate Image Retrieval*, in Proceedings of the Ninth International Workshop on Image Analysis for Multimedia Interactive Services, IEEE, Klagenfurt, May, 2008.
Optimal results, comparable to the Joint Histogram were observed for this feature.
This feature is not part of the MPEG7 standard.

Further study by using our program and extending it to test the configuration possibilities of the Lire features might be of interest, but we consider that it exceeds the scope of the current assignment.

Shady Akhras
Gabriel Campero
OvGU Magdeburg, June 2014.