OpenIMAJ
Intelligent Multimedia Analysis In Java

Jonathon Hare
Sina Samangooei
David Dupplaw

The OpenIMAJ Tutorial

The OpenIMAJ Tutorial

Jonathon Hare, Ph.D.

Electronics and Computer Science
The University of Southampton
http://users.ecs.soton.ac.uk/jsh2

Sina Samangooei, Ph.D.

Electronics and Computer Science
The University of Southampton
http://users.ecs.soton.ac.uk/ss

David Dupplaw, Ph.D.

Electronics and Computer Science
The University of Southampton
http://users.ecs.soton.ac.uk/dpd

The OpenIMAJ Tutorial

Jonathon Hare
Sina Samangooei
David Dupplaw

Table of Contents

Foreword: 1.3.5

The latest version of this book is always available on the OpenIMAJ site: http://www.openimaj.org

We welcome your feedback. Please do not hesitate to contact the OpenIMAJ team with any questions or ideas you may have about the OpenIMAJ.

If you have any feedback or questions, you are encouraged to email us at : openimaj@gmail.com[1] or to file an issue on the OpenIMAJ issue tracker at http://jira.openimaj.org.

OpenIMAJ has a long history, starting well before its public release; Parts of the original codebase come from projects funded by the European Union's 6th Framework Programme[2]), the Engineering and Physical Sciences Research Council[3] the Arts and Humanities Research Council[4], Ordnance Survey[5] and the BBC[6].

Current development of the OpenIMAJ and ImageTerrier software is primarily funded by the European Union Seventh Framework Programme[7] under the ARCOMEM[8] project (grant agreement 270239). The initial public releases were also funded by the European Union Seventh Framework Programme[9] under the LivingKnowledge[10] (grant agreement 231126) together with the LiveMemories[11] project, graciously funded by the Autonomous Province of Trento[12].

Edition: 1.3.5 (2016-06-21)

[1] mailto:openimaj@gmail.com
[2] http://cordis.europa.eu/fp6/
[3] http://www.epsrc.ac.uk
[4] http://www.ahrc.ac.uk
[5] http://www.ordnancesurvey.co.uk/
[6] http://www.bbc.co.uk
[7] http://cordis.europa.eu/fp7/home_en.html
[8] http://www.arcomem.eu
[9] http://cordis.europa.eu/fp7/home_en.html
[10] http://www.livingknowledge-project.eu
[11] http://livememories.org/
[12] http://www.provincia.tn.it

Preface

OpenIMAJ is a set of libraries and tools for multimedia content analysis and content generation. OpenIMAJ is very broad and contains everything from state-of-the-art computer vision (e.g. SIFT descriptors, salient region detection, face detection, etc.) and advanced data clustering, through to software that performs analysis on the content, layout and structure of webpages.

OpenIMAJ is primarily written in pure Java and, as such, is completely platform independent. The video capture and hardware libraries contain some native code but Linux (x86, x86_64 and ARM CPUs are supported), OSX and Windows are supported out of the box (under both 32 and 64 bit JVMs). It is possible to write programs that use the libraries in any JVM language that supports Java interoperability, such as Groovy, Jython, JRuby or Scala. OpenIMAJ can even be run on Android phones and tablets.

The OpenIMAJ software is structured into a number of modules. The modules can be used independently, so if, for instance, you were developing data clustering software using OpenIMAJ you wouldn't need to acquire the modules related to images or text. The list on the following page illustrates the modules and summarises the functionality in each component.

This tutorial aims to instruct the reader on how to get up and running writing code using OpenIMAJ. Currently the tutorial covers the following areas:

1. Getting started with OpenIMAJ using Maven
2. Processing your first image
3. Introduction to clustering, segmentation and connected components
4. Processing video
5. Finding faces
6. Global image features
7. SIFT and feature matching

In the future we hope to add more content to the tutorial covering the following:

- Basic text analysis
- Image and video indexing using ImageTerrier
- Compiling OpenIMAJ from source
- Tracking features in video
- Audio processing
- Speech recognition
- Hardware interfaces
- Advanced local features
- Scalable processing with OpenIMAJ/Hadoop
- Machine learning
- Building a bibliography of the techniques used in your code.

The OpenIMAJ Modules

`archetypes`
: Maven archetypes for OpenIMAJ.

  `openimaj-quickstart-archetype`
  : Maven quickstart archetype for OpenIMAJ.

  `openimaj-subproject-archetype`
  : Maven archetype for creating OpenIMAJ subprojects with the most of the standard configuration completed automatically.

`core`
: Submodule for modules containing functionality used across the OpenIMAJ libraries.

  `core`
  : Core library functionality concerned with general programming problems rather than multimedia specific functionality. Includes I/O utilities, randomisation, hashing and type conversion.

  `core-image`
  : Core definitions of images, pixels and connected components. Also contains interfaces for processors for these basic types. Includes loading, saving and displaying images.

  `core-video`
  : Core definitions of a video type and functionality for displaying and processing videos.

  `core-audio`
  : Core definitions of audio streams and samples/chunks. Also contains interfaces for processors for these basic types.

  `core-math`
  : Mathematical implementations including geometric, matrix and statistical operators.

  `core-feature`
  : Core notion of features, usually denoted as arrays of data. Definitions of features for all primitive types, features with location and lists of features (both in memory and on disk).

  `core-experiment`
  : Classes to formally describe experiments and evaluations, with support for automatically evaluating their results.

  `core-citation`
  : Tools for annotating code with publication references and automatically generating bibliographies for your code.

  `core-aop-support`
  : Core support for Aspect Oriented Programming and Bytecode manipulation as used in core-citation and core-experiment.

`image`
: Submodule for image related functionality.

  `image-processing`
  : Implementations of various image, pixel and connected component processors (resizing, convolution, edge detection, ...).

`image-local-features`
: Methods for the extraction of local features. Local features are descriptions of regions of images (SIFT, ...) selected by detectors (Difference of Gaussian, Harris, ...).

`image-feature-extraction`
: Methods for the extraction of low-level image features, including global image features and pixel/patch classification models.

`faces`
: Implementation of a flexible face-recognition pipeline, including pluggable detectors, aligners, feature extractors and recognisers.

`image-annotation`
: Methods for describing automatic image annotators.

`object-detection`
: Support for object detection, including a haar-cascade implementation.

`image-indexing-retrieval`
: Subproject for CBIR related components.

`vector-image`
: Support for vector images using the Batik SVG library.

`camera-calibration`
: Camera calibration techniques and associated code.

`multiview`
: OpenIMAJ Multiview reconstruction and 3D imaging library.

`video`
: Sub-modules containing support for analysing and processing video.

`video-processing`
: Various video processing algorithms, such as shot-boundary detection.

`xuggle-video`
: Plugin to use Xuggler as a video source. Allows most video formats to be read into OpenIMAJ.

`gstreamer-video`
: ...

`audio`
: Submodule for audio processing and analysis related functionality.

`audio-processing`
: Implementations of various audio processors (e.g. multichannel conversion, volume change, ...).

`machine-learning`
: Sub-module for machine-learning libraries.

`clustering`
: Various clustering algorithm implementations for all primitive types including random, random forest, K-Means (Exact, Hierarchical and Approximate), ...

`nearest-neighbour`
: Implementations of K-Nearest-Neighbour methods, including approximate methods.

`machine-learning`
: The OpenIMAJ Machine Learning Library contains implementations of optimised machine learning techniques that can be applied to OpenIMAJ structures and features.

`text`
: Text Analysis functionality for OpenIMAJ.

`nlp`
: The OpenIMAJ NLP Library contains a text pre-processing pipeline which goes from raw unstructured text to part of speech tagged stemmed text.

`thirdparty`
: Useful third-party libraries (possibly originally written in other languages) that have been ported to Java and integrated with OpenIMAJ. Not all modules have the same license.

`klt-tracker`
: A port of Stan Birchfield's Kanade-Lucas-Tomasi tracker to OpenIMAJ. See http://www.ces.clemson.edu/~stb/klt/.

`tld`
: A port of Georg Nebehay's tracker https://github.com/gnebehay/OpenTLD originally created by Zdenek Kalal https://github.com/zk00006/OpenTLD.

`ImprovedArgs4J`

`IREval`
: A modified version of the IREval module (version 4.12) from the lemur project with extensions to better integrate with OpenIMAJ. See http://www.lemurproject.org.

`FaceTracker`
: Port of Jason Mora Saragih's FaceTracker to Java using OpenIMAJ. FaceTracker is an implementation of a facial model tracker using a Constrained Local Model.

`processing-core`

`MatrixLib`

`JTransforms`
: JTransforms is the first, open source, multithreaded FFT library written in pure Java. Currently, four types of transforms are available: Discrete Fourier Transform (DFT), Discrete Cosine Transform (DCT), Discrete Sine Transform (DST) and Discrete Hartley Transform (DHT). The code is derived from General Purpose FFT Package written by Takuya Ooura and from Java FFTPack written by Baoshe Zhang. This version has been modified to daemonize threads and stop any application using the library waiting after execution has finished, and is based on revision 29 of the svn version of the code from 2014-05-18.

`demos`
: Demos showing the functionality of OpenIMAJ.

`demos`
: Demos showing the use of OpenIMAJ.

`sandbox`
: A project for various tests that don't quite constitute demos but might be useful to look at.

`touchtable`
: Playground for the ECS touchtable.

`SimpleMosaic`
: Demo showing SIFT matching with a Homography model to achieve image mosaicing.

`CampusView`
: Demo showing how we used OpenIMAJ to create a Street-View-esq capture system.

`ACMMM-Presentation`
: The OpenIMAJ presentation for ACMMM 2011. Unlike a normal presentation, this one isn't PowerPoint, but is actually an OpenIMAJ Demo App!.

`examples`
: Example code snippets showing specific functionalities.

`SimpleOCR`
: Simple (numerical) OCR using template matching to extract the timestamps embedded in GlacsWeb time-lapse videos.

`knowledge`
: Submodule for modules related to knowledge representation and reasoning.

`core-rdf`
: Core support for the Resource Description Framework (RDF), including object-rdf mapping.

`ontologies`
: Submodule for modules which each represent a specific ontology represented in java.

`sioc`

`geocoder`
: Geocoding and reverse geocoding utilities.

`test-resources`
: Resources for running OpenIMAJ JUnit tests.

`tools`
: Sub-modules containing commandline tools exposing OpenIMAJ functionality.

`core-tool`
: Core library for building openimaj tools.

`GlobalFeaturesTool`
: A tool for extracting various global features from images.

`ClusterQuantiserTool`
: Tool for clustering and quantising features.

`LocalFeaturesTool`
: Tool for extracting local image features.

`FaceTools`
: Tools for detecting, extracting and comparing faces within images.

`FeatureVisualisation`
: Tools for visualising certain types of image feature.

`CityLandscapeClassifier`
: Tool for classifying images as cityscapes (or images containing man-made objects) or landscapes. Based on the edge direction coherence vector.

`WebTools`
: Tools and utilities for extracting info from web-pages.

`OCRTools`
: Tools for training and testing OCR.

`ImageCollectionTool`
: Tool for extracting images from collections (zip, gallery, video etc.).

`SimilarityMatrixTool`
: A tool for performing operations on Similarity Matrices.

`TwitterPreprocessingTool`
: Tool for applying a text preprocessing pipeline to twitter tweets.

`RedditHarvester`
: A tool for harvesting posts and comments from reddit.

`picslurper`
: Tool for grabbing all the images from a social media stream and holding on to some simple stats.

`ReferencesTool`
: Tool for running an OpenIMAJ program and extracting the references for used methods and classes.

`openimaj-processing`
: Set of tools for integrating OpenIMAJ with processing.

`CBIRTools`
: Tools for content-based image indexing and retrieval.

`hadoop`
: Sub-modules for integrating OpenIMAJ with Apache Hadoop to allow Map-Reduce style distributed processing.

`core-hadoop`
: Reusable wrappers and helpers to access and create sequence-files and map-reduce jobs.

`tools`
: Tools that provide multimedia analysis algorithms expressed as Map-Reduce jobs that can be run on a Hadoop cluster.

`core-hadoop-tool`
: Tool for clustering and quantising features using Map-Reduce jobs on a Hadoop cluster.

`HadoopFastKMeans`
: Distributed feature clustering tool.

`HadoopDownloader`
: Distributed download tool.

`HadoopLocalFeaturesTool`
: Distributed local image feature extraction tool.

`SequenceFileTool`
: Tool for building, inspecting and extracting Hadoop SequenceFiles.

`HadoopGlobalFeaturesTool`
: Distributed global image feature extraction tool.

`HadoopClusterQuantiserTool`
: Distributed feature quantisation tool.

`HadoopTwitterPreprocessingTool`
: Tool for clustering and quantising features using Map-Reduce jobs on a Hadoop cluster.

`HadoopTwitterTokenTool`
: Tool for clustering and quantising features using Map-Reduce jobs on a Hadoop cluster.

`SequenceFileIndexer`
: Tool for building an index of the keys in a Hadoop SequenceFile.

`HadoopEXIFTool`
: Tool for extracting EXIF information from images on a Hadoop cluster.

`SequenceFileMerger`
: Base for tools built on hadoop.

`HadoopImageIndexer`
: ...

`streams`
: Sub-modules for stream provision, processing and analysis.

`storm`
: Sub-modules for integrating OpenIMAJ with Storm to allow distrbuted stream processing.

`core-storm`
: The main Storm dependency and some extra utility classes specific to OpenIMAJ.

`tools`
: Tools that provide multimedia analysis algorithms expressed as Storm topology that can be run locally or on a Storm cluster.

`StormTwitterPreprocessingTool`
: Tool for processing tweets on Storm.

`core-storm-tool`
: Base for tools built on top of storm.

`common-stream`
: Classes providing access to common types of streaming data (twitter, irc, wikimedia edits, etc) as well as providing methods and techniques for working with streams.

`web`
: Sub-modules containing support for analysing and processing web-pages.

  `core-web`
  : Implementation of a programatic offscreen web browser and utility functions.

  `webpage-analysis`
  : Utilities for analysing the content and visual layout of a web-page.

  `readability4j`
  : Readability4J is a partial re-implementation of the original readability.js script in Java. Many modifications have been made however.

  `twitter`
  : The twitter project contains tools with which to read JSON data from the twitter API and process the data.

  `data-scraping`
  : Utility methods and classes for extracting data and information from the web.

`hardware`
: Sub-modules containing interfaces to hardware devices that we've used in projects built using OpenIMAJ.

  `core-video-capture`
  : Cross-platform video capture interface using a lightweight native interface. Supports 32 and 64 bit JVMs under Linux, OSX and Windows.

  `serial-driver`
  : Interface to hardware devices that connect to serial or USB-serial ports.

  `gps`
  : Interface to GPS devices that support the NMEA protocol.

  `compass`
  : Interface to an OceanServer OS5000 digital compass.

  `nmea-parser`
  : Contains a parser for NMEA sentences written in Groovy.

  `kinect`
  : The OpenIMAJ Kinect Library contains the core classes and native code required interface with the Kinect device.

  `turntable`
  : Integration with our serially controlled turntable.

  `core-gpgpu`

`content`
: Libraries for multimedia content creation.

  `slideshow`
  : A library for creating slideshows and presentations that can contain interactive demos that utilise all OpenIMAJ components.

  `animation`
  : Code to help make an animation of data/models/etc.

`visualisations`
: A library that contains classes for visualising various different features, such as audio and video.

`ide-integration`
: Plugins to aid OpenIMAJ development in various IDE's.

`documentation`
: Submodule for documentation.

`tutorial`
: The OpenIMAJ tutorial.

`tutorial-content`
: The content of the OpenIMAJ tutorial.

`tutorial-pdf`
: OpenIMAJ tutorial in PDF format.

`tutorial-html`
: OpenIMAJ Tutorial in HTML format.

`tutorial-code`
: The source-code for the OpenIMAJ Tutorial.

Part I. OpenIMAJ Fundamentals

Chapter 1. Getting started with OpenIMAJ using Maven

Apache Maven is a project management tool. Maven performs tasks such as automatic dependency management, project packaging and more. We **strongly** encourage anyone using OpenIMAJ to use Maven to get their own project started. We've even provided a Maven **archetype** for OpenIMAJ (basically a project template) that lets you get started programming with OpenIMAJ quickly.

Tip

You can find out more about Apache Maven at http://maven.apache.org.

OpenIMAJ requires Maven 2 or 3; if you want to build OpenIMAJ from source you will need Maven 3. You can check if you have Maven installed already by opening a terminal (or DOS command prompt) and typing:

```
mvn -version
```

If Maven is found the, version will be printed. If the version is less than 2.2.1, or Maven was not found, go to http://maven.apache.org to download and install it. Once you've installed Maven try the above command to test that it is working.

To create a new OpenIMAJ project, run the following command:

```
mvn -DarchetypeCatalog=http://maven.openimaj.org/archetype-catalog.xml archetype:generate
```

Maven will then prompt you for some input. Firstly, when prompted, choose the `openimaj-quickstart-archetype` and choose the latest version. For the `groupId`, enter something that identifies you or a group that you belong to (for example, I might choose `uk.ac.soton.ecs.jsh2` for personal projects, or `org.openimaj` for OpenIMAJ sub-projects). For the `artifactId` enter a name for your project (for example, `OpenIMAJ-Tutorial01`). The version can be left as `1.0-SNAPSHOT`, and the default package is also OK. Finally enter `Y` and press return to confirm the settings. Maven will then generate a new project in a directory with the same name as the `artifactId` you provided.

Overriding the OpenIMAJ version

Versions of the archetype after `1.0.5` automatically select the corresponding OpenIMAJ version. With all versions of the archetype, you can override this by setting the `openimajVersion` on the command-line with the `-D` argument.

The project directory contains a file called `pom.xml` and a directory called `src`. The `pom.xml` file describes all of the dependencies of the project and also contains instructions for packaging the project into a fat jar that contains all your project code and resources together with the dependencies. If you find that you need to add another library to your project, you should do so by editing the `pom.xml` file and adding a new dependency. The `src` directory contains the code for your project. In particular, `src/main/java` contains your java application code and `src/test/java` contains unit tests.

The default project created by the archetype contains a small "hello world" application. To compile and assemble the "hello world" application you `cd` into the project directory from the command line (replacing ***`OpenIMAJ-Tutorial01`*** with the name of your project):

```
cd OpenIMAJ-Tutorial01
```

and run the command:

```
mvn assembly:assembly
```

This will create a new directory called target that contains the assembled application jar (the assembled jar is the one whose name ends with `-jar-with-dependencies.jar`). To run the application, enter:

```
java -jar target/OpenIMAJ-Tutorial01-1.0-SNAPSHOT-jar-with-dependencies.jar
```

The application will then run, and a window should open displaying a picture with the text "hello world". Closing the window, or **ctrl-c** on the command line, will quit the application.

1.1. Integration with your favourite IDE

We could now go ahead and start playing with the code in a text editor, however this really isn't recommended! Using a good Integrated Development Environment (IDE) with auto-completion will make your experience much better.

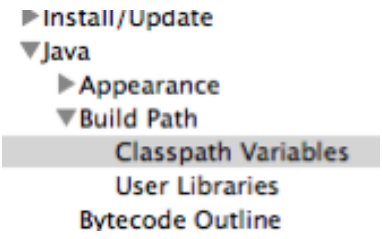
Maven integrates with all the popular IDEs. The OpenIMAJ developers all use Eclipse[1] so that is what we're most familiar with, however we should be able to help getting it set up in a different IDE if you wish.

Integration with Eclipse is quite simple. From the command line, inside the project directory, issue the command:

```
mvn eclipse:eclipse
```

This will generate Eclipse project files in the same directory. In Eclipse you can then import the project into the Eclipse workspace (File > import..., choose Existing projects into workspace, select the project directory, make sure Copy projects into workspace is **unchecked**, then click Finish). The project should then appear in the workspace and you'll be able to look at the `App.java` file that was generated by the archetype.

IMPORTANT By default Eclipse doesn't know about Maven and its repositories of jars. When you first import an OpenIMAJ project into Eclipse it will have errors. You can fix this by adding a new Java classpath variable (Eclipse > Preferences > Java > Build Path > Classpath Variables) called `M2_REPO`. The value of this variable is the location of your `.m2/repository` directory. For Unix systems this is usually found in your home directory, for Windows systems it is found in `C:\Documents and Settings\<user>\`.



Once you've opened the `App.java` file in Eclipse, you can right-click on it and select Run as > Java Application to run it from within Eclipse.

[1] http://www.eclipse.org

1.2. Exercises

1.2.1. Exercise 1: Playing with the sample application

Take a look at the `App.java` from within your IDE. Can you modify the code to render something other than "hello world" in a different font and colour?

Part II. Image Fundamentals

OpenIMAJ started life as a set of classes for analysing image content. The chapters in Part II present a number of practical examples and exercises that will get you up an running manipulating images and performing real-world image analysis tasks.

Chapter 2. Processing your first image

In this section we'll start with the "hello world" app and show you how you can load an image, perform some basic processing on the image, draw some stuff on your image and then display your results.

Loading images into Java is usually a horrible experience. Using Java `ImageIO`, one can use the `read()` method to create a `BufferedImage` object. Unfortunately the `BufferedImage` object hides the fact that it is (and in fact all digital raster images are) simply arrays of pixel values. A defining philosophy of OpenIMAJ is to *keep things simple* which in turn means in OpenIMAJ images are as close as one can get to being **just arrays of pixel values**.

To read and write images in OpenIMAJ we use the `ImageUtilities` class. In the `App.java` class file remove the sample code within the main method and add the following line:

```
MBFImage image = ImageUtilities.readMBF(new File("file.jpg"));
```

For this tutorial, read the image from the following URL:

```
MBFImage image = ImageUtilities.readMBF(new URL("http://static.openimaj.org/media/tutorial/sinaface.jpg"));
```

The `ImageUtilities` class provides the ability to read `MBFImages` and `FImages`. An `FImage` is a greyscale image which represents each pixel as a value between `0` and `1`. An `MBFImage` is a multi-band version of the `FImage`; under the hood it actually contains a number `FImage` objects held in a list each representing a band of the image. What these bands represent is given by the image's public `colourSpace` field, which we can print as follows:

```
System.out.println(image.colourSpace);
```

If we run the code, we'll see that the image is an RGB image with three `FImages` representing the red, blue and green components of the image in that order.

You can display any OpenIMAJ image object using the `DisplayUtilities` class. In this example we display the image we have loaded then we display the red channel of the image alone:

```
DisplayUtilities.display(image);
DisplayUtilities.display(image.getBand(0), "Red Channel");
```

In an image-processing library, images are no good unless you can do something to them. The most basic thing you can do to an image is fiddle with its pixels. In OpenIMAJ, as an image is just an array of floats, we make this is quite easy. Let's go through our colour image and set all its blue and green pixels to black:

```
MBFImage clone = image.clone();
for (int y=0; y<image.getHeight(); y++) {
    for(int x=0; x<image.getWidth(); x++) {
        clone.getBand(1).pixels[y][x] = 0;
        clone.getBand(2).pixels[y][x] = 0;
    }
}
DisplayUtilities.display(clone);
```

Note that the first thing we do here is to `clone` the image to preserve the original image for the remainder of the tutorial. The pixels in an `FImage` are held in a 2D float array. The rows of the image are held in the first array that, in turn, holds each of the column values for that row: `[y][x]`. By displaying this image we should see an image where two channels are black and one channel is not. This results in an image that looks rather red.

Though it is helpful to sometimes get access to individual image pixels, OpenIMAJ provides a lot of methods to make things easier. For example, we could have done the above like this instead:

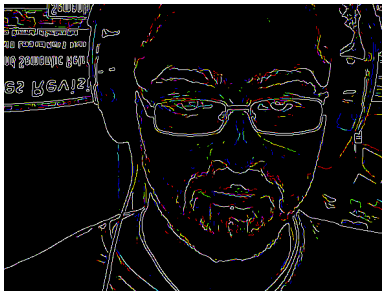
```
clone.getBand(1).fill(0f);
clone.getBand(2).fill(0f);
```

More complex image operations are wrapped up by OpenIMAJ processor interfaces: `ImageProcessors`, `KernelProcessors`, `PixelProcessors` and `GridProcessors`. The distinction between these is how their algorithm works internally. The overarching similarity is that an image goes in and a processed image (or data) comes out. For example, a basic operation in image processing is **edge detection**. A popular edge detection algorithm is the *Canny edge detector*. We can call the Canny edge detector like so:

```
image.processInplace(new CannyEdgeDetector());
```

When applied to a colour image, each pixel of each band is replaced with the edge response at that point (for simplicity you can think of this as the difference between that pixel and its neighbouring pixels). If a particular edge is only strong in one band or another then that colour will be strong, resulting in the psychedelic colours you should see if you display the image:

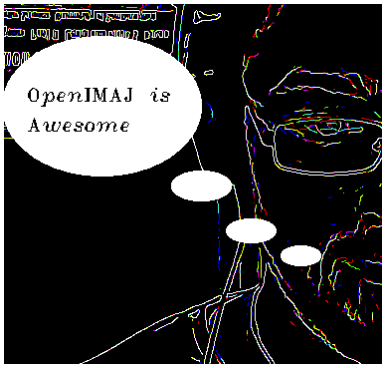
```
DisplayUtilities.display(image);
```



Finally, we can also draw on our image in OpenIMAJ. On every `Image` object there is a set of drawing functions that can be called to draw points, lines, shapes and text on images. Let's draw some speech bubbles on our image:

```
image.drawShapeFilled(new Ellipse(700f, 450f, 20f, 10f, 0f), RGBColour.WHITE);
image.drawShapeFilled(new Ellipse(650f, 425f, 25f, 12f, 0f), RGBColour.WHITE);
image.drawShapeFilled(new Ellipse(600f, 380f, 30f, 15f, 0f), RGBColour.WHITE);
image.drawShapeFilled(new Ellipse(500f, 300f, 100f, 70f, 0f), RGBColour.WHITE);
image.drawText("OpenIMAJ is", 425, 300, HersheyFont.ASTROLOGY, 20, RGBColour.BLACK);
image.drawText("Awesome", 425, 330, HersheyFont.ASTROLOGY, 20, RGBColour.BLACK);
DisplayUtilities.display(image);
```

Here we construct a series of ellipses (defined by their centre [x, y], axes [major, minor] and rotation) and draw them as white filled shapes. Finally, we draw some text on the image and display it.



2.1. Exercises

2.1.1. Exercise 1: DisplayUtilities

Opening lots of windows can waste time and space (for example if you wanted to view images on every iteration of a process within a loop). In OpenIMAJ we provide a facility to open a *named display* so that was can reuse the display referring to it by name. Try to do this with all the images we display in this tutorial. Only 1 window should open for the whole tutorial.

2.1.2. Exercise 2: Drawing

Those speech bubbles look rather plain; why not give them a nice border?

Chapter 3. Introduction to clustering, segmentation and connected components

In this tutorial we'll create an application that demonstrates how an image can be broken into a number of regions. The process of separating an image into regions, or segments, is called **segmentation**. Segmentation is a widely studied area in computer vision. Researchers often try to optimise their segmentation algorithms to try and separate the **objects** in the image from the **background**.

To get started, create a new OpenIMAJ project using the Maven archetype, import it into your IDE, and delete the sample code from within the generated `main()` method of the `App` class. In the `main()` method, start by adding code to load an image (choose your own image):

```
MBFImage input = ImageUtilities.readMBF(new URL("http://..."));
```

To segment our image we are going to use a machine learning technique called **clustering**. Clustering algorithms automatically group similar things together. In our case, we'll use a popular clustering algorithm called **K-Means** clustering to group together all the similar colours in our image. Each group of similar colours is known as a **class**. The K-means clustering algorithm requires you set the number of classes you wish to find **a priori** (i.e. beforehand).

K-means Clustering

K-Means initialises cluster centroids with randomly selected data points and then iteratively assigns the data points to their closest cluster and updates the centroids to the mean of the respective data points.

Colours in our input image are represented in **RGB colour space**; that is each pixel is represented as three numbers corresponding to a red, green and blue value. In order to measure the similarity of a pair of colours the "distance" between the colours in the colour space can be measured. Typically, the distance measured is the **Euclidean** distance. Unfortunately, distances in RGB colour space do not reflect what humans perceive as similar/dissimilar colours. In order to work-around this problem it is common to transform an image into an alternative colour space. The **Lab colour space** (pronounced as separate letters, L A B) is specifically designed so that the Euclidean distance between colours closely matches the perceived similarity of a colour pair by a human observer.

Euclidean Distance

The Euclidean distance is the straight-line distance between two points. It is named after the "Father of Geometry", the Greek mathematician Euclid.

To start our implementation, we'll first apply a colour-space transform to the image:

```
input = ColourSpace.convert(input, ColourSpace.CIE_Lab);
```

We can then construct the K-Means algorithm:

```
FloatKMeans cluster = FloatKMeans.createExact(2);
```

The parameter (2) is the number of clusters or classes we wish the algorithm to generate. We can optionally provide a second integer argument that controls the maximum number of iterations of the algorithm (the default is 30 iterations if we don't specify otherwise).

Tip

There are a number of different static factory methods on the `FloatKMeans` class, as well as constructors that allow various flavours of the K-Means algorithm to be instantiated. In particular, the `FloatKMeans.createKDTreeEnsemble(int)` method creates an *approximate* K-means implementation using a technique based on an ensemble of KD-Trees. The approximate algorithm is much faster than the exact algorithm when there is very high-dimensional data; in this case, with only three dimensions, the approximate algorithm is not required.

> All the OpenIMAJ K-Means implementations are multithreaded and automatically takes advantage of all the processing power they can obtain by default. This behaviour can of course be controlled programatically however.

The `FloatKMeans` algorithm takes its input as an array of floating point vectors (`float[][]`). We can flatten the pixels of an image into the required form using the `getPixelVectorNative()` method:

```
float[][] imageData = input.getPixelVectorNative(new float[input.getWidth() * input.getHeight()][3]);
```

The K-Means algorithm can then be run to group all the pixels into the requested number of classes:

```
FloatCentroidsResult result = cluster.cluster(imageData);
```

Each class or cluster produced by the K-Means algorithm has an index, starting from 0. Each class is represented by its centroid (the average location of all the points belonging to the class). We can print the coordinates of each centroid:

```
float[][] centroids = result.centroids;
for (float[] fs : centroids) {
    System.out.println(Arrays.toString(fs));
}
```

Now is a good time to test the code. Running it should print the (`L, a, b`) coordinates of each of the classes.

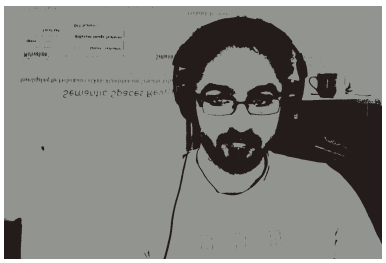
We can now use a `HardAssigner` to assign each pixel in our image to its respective class using the centroids learned during the `FloatKMeans`. This is a process known as **classification**. There are a number of different `HardAssigners`, however, `FloatCentroidsResult` has a method called `defaultHardAssigner()` which will return an assigner fit for our purposes. `HardAssigners` have a method called `assign()` which takes a vector (the `L, a, b` value of a single pixel) and returns the index of the class that it belongs to. We'll start by creating an image that visualises the pixels and their respective classes by replacing each pixel in the input image with the centroid of its respective class:

```
HardAssigner<float[],?,?> assigner = result.defaultHardAssigner();
for (int y=0; y<input.getHeight(); y++) {
    for (int x=0; x<input.getWidth(); x++) {
        float[] pixel = input.getPixelNative(x, y);
        int centroid = assigner.assign(pixel);
        input.setPixelNative(x, y, centroids[centroid]);
    }
}
```

We can then display the resultant image. Note that we need to convert the image back to RGB colour space for it to display properly:

```
input = ColourSpace.convert(input, ColourSpace.RGB);
DisplayUtilities.display(input);
```

Running the code will display an image that looks a little like the original image but with as many colours as there are classes.



To actually produce a segmentation of the image we need to group together all pixels with the same class that are touching each other. Each set of pixels representing a segment is often referred to as a **connected component**. Connected components in OpenIMAJ are modelled by the `ConnectedComponent` class.

The `GreyscaleConnectedComponentLabeler` class can be used to find the connected components:

```
GreyscaleConnectedComponentLabeler labeler = new GreyscaleConnectedComponentLabeler();
List<ConnectedComponent> components = labeler.findComponents(input.flatten());
```

Note that the `GreyscaleConnectedComponentLabeler` only processes greyscale images (the `FImage` class) and not the colour image (`MBFImage` class) that we created. The `flatten()` method on `MBFImage` merges the colours into grey values by averaging their RGB values.

Tip

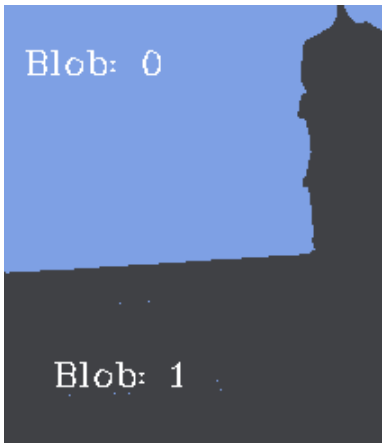
OpenIMAJ also contains a class called `ConnectedComponentLabeler` which can only be used on binary (pure black and white) `FImages`.

The `ConnectedComponent` class has many useful methods for extracting information about the shape of the region. Lets draw an image with the components numbered on it. We'll use the centre of mass of each region to position the number and only render numbers for regions that are over a certain size (50 pixels in this case):

```
int i = 0;
for (ConnectedComponent comp : components) {
    if (comp.calculateArea() < 50)
        continue;
    input.drawText("Point:" + (i++), comp.calculateCentroidPixel(), HersheyFont.TIMES_MEDIUM, 20);
}
```

Finally, we can display the image with the labels:

```
DisplayUtilities.display(input);
```



3.1. Exercises

3.1.1. Exercise 1: The PixelProcessor

Rather than looping over the image pixels using two for loops, it is possible to use a `PixelProcessor` to accomplish the same task:

```
image.processInplace(new PixelProcessor<Float[]>() {
    Float[] processPixel(Float[] pixel) {
        ...
    }
});
```

Can you re-implement the loop that replaces each pixel with its class centroid using a `PixelProcessor`?

What are the advantages and disadvantages of using a `PixelProcessor`?

3.1.2. Exercise 2: A real segmentation algorithm

The segmentation algorithm we just implemented can work reasonably well, but is rather naïve. OpenIMAJ contains an implementation of a popular segmentation algorithm called the `FelzenszwalbHuttenlocherSegmenter`.

Try using the `FelzenszwalbHuttenlocherSegmenter` for yourself and see how it compares to the basic segmentation algorithm we implemented. You can use the `SegmentationUtilities.renderSegments()` static method to draw the connected components produced by the segmenter.

Chapter 4. Global image features

The task in this tutorial is to understand how we can extract numerical representations from images and how these numerical representations can be used to provide similarity measures between images, so that we can, for example, find the most similar images from a set.

As you know, images are made up of pixels which are basically numbers that represent a colour. This is the most basic form of numerical representation of an image. However, we can do calculations on the pixel values to get other numerical representations that mean different things. In general, these numerical representations are known as **feature vectors** and they represent particular **features**.

Let's take a very common and easily understood type of feature. It's called a colour histogram and it basically tells you the proportion of different colours within an image (e.g. 90% red, 5% green, 3% orange, and 2% blue). As pixels are represented by different amounts of red, green and blue we can take these values and accumulate them in our histogram (e.g. when we see a red pixel we add 1 to our "red pixel count" in the histogram).

A histogram can accrue counts for any number of colours in any number of dimensions but the usual is to split the red, green and blue values of a pixel into a smallish number of "bins" into which the colours are thrown. This gives us a three-dimensional cube, where each small cubic bin is accruing counts for that colour.

OpenIMAJ contains a multidimensional `MultidimensionalHistogram` implementation that is constructed using the number of bins required in each dimension. For example:

```
MultidimensionalHistogram histogram = new MultidimensionalHistogram( 4, 4, 4 );
```

This code creates a histogram that has 64 (4 × 4 × 4) bins. However, this data structure does not do anything on its own. The `HistogramModel` class provides a means for creating a `MultidimensionalHistogram` from an image. The `HistogramModel` class assumes the image has been normalised and returns a normalised histogram:

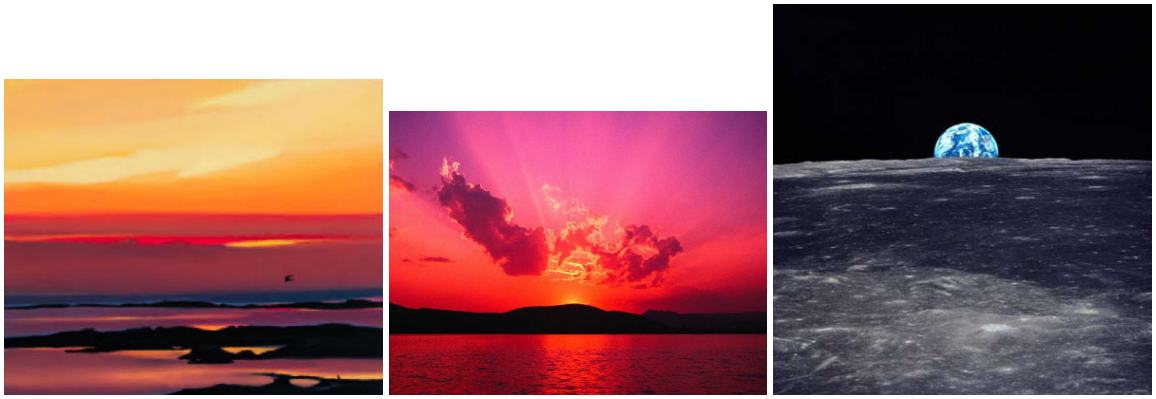
```
HistogramModel model = new HistogramModel( 4, 4, 4 );
model.estimateModel( image );
MultidimensionalHistogram histogram = model.histogram;
```

You can print out the histogram to see what sort of numbers you get for different images. Note that the you can re-use the `HistogramModel` by applying it to different images. If you do reuse the `HistogramModel` the `model.histogram` will be the same object, so you'll need to `clone()` it if you need to keep hold of its values for multiple images. Let's load in 3 images then generate and store the histograms for them:

```
URL[] imageURLs = new URL[] {
   new URL( "http://users.ecs.soton.ac.uk/dpd/projects/openimaj/tutorial/hist1.jpg" ),
   new URL( "http://users.ecs.soton.ac.uk/dpd/projects/openimaj/tutorial/hist2.jpg" ),
   new  URL( "http://users.ecs.soton.ac.uk/dpd/projects/openimaj/tutorial/hist3.jpg" )
};

List<MultidimensionalHistogram> histograms = new ArrayList<MultidimensionalHistogram>();
HistogramModel model = new HistogramModel(4, 4, 4);

for( URL u : imageURLs ) {
    model.estimateModel(ImageUtilities.readMBF(u));
    histograms.add( model.histogram.clone() );
}
```



We now have a list of histograms from our images. The `Histogram` class extends a class called the `MultidimensionalDoubleFV` which is a feature vector represented by multidimensional set of double precision numbers. This class provides us with a `compare()` method which allows comparison between two multidimensional sets of doubles. This method takes the other feature vector to compare against and a comparison method which is implemented in the `DoubleFVComparison` class.

So, we can compare two histograms using the Euclidean distance measure like so:

```
double distanceScore = histogram1.compare( histogram2, DoubleFVComparison.EUCLIDEAN );
```

This will give us a score of how similar (or dissimilar) the histograms are. It's useful to think of the output score as a **distance** apart in space. Two very similar histograms will be very close together so have a small distance score, whereas two dissimilar histograms will be far apart and so have a large distance score.

The Euclidean distance measure is symmetric (that is, if you compare `histogram1` to `histogram2` you will get the same score if you compare `histogram2` to `histogram1`) so we can compare all the histograms with each other in a simple, efficient, nested loop:

```
for( int i = 0; i < histograms.size(); i++ ) {
    for( int j = i; j < histograms.size(); j++ ) {
        double distance = histograms.get(i).compare( histograms.get(j), DoubleFVComparison.EUCLIDEAN );
    }
}
```

4.1. Exercises

4.1.1. Exercise 1: Finding and displaying similar images

Which images are most similar? Does that match with what you expect if you look at the images? Can you make the application display the two most similar images that are not the same?

4.1.2. Exercise 2: Exploring comparison measures

What happens when you use a different comparison measure (such as `DoubleFVComparison.INTERSECTION`)?

Chapter 5. SIFT and feature matching

In this tutorial we'll look at how to compare images to each other. Specifically, we'll use a popular **local feature descriptor** called **SIFT** to extract some *interesting points* from images and describe them in a standard way. Once we have these local features and their descriptions, we can match local features to each other and therefore compare images to each other, or find a visual query image within a target image, as we will do in this tutorial.

Firstly, lets load up a couple of images. Here we have a magazine and a scene containing the magazine:

```
MBFImage query = ImageUtilities.readMBF(new URL("http://static.openimaj.org/media/tutorial/query.jpg"));
MBFImage target = ImageUtilities.readMBF(new URL("http://static.openimaj.org/media/tutorial/target.jpg"));
```

The first step is feature extraction. We'll use the **difference-of-Gaussian** feature detector which we describe with a **SIFT descriptor**. The features we find are described in a way which makes them invariant to size changes, rotation and position. These are quite powerful features and are used in a variety of tasks. The standard implementation of SIFT in OpenIMAJ can be found in the `DoGSIFTEngine` class:

```
DoGSIFTEngine engine = new DoGSIFTEngine();
LocalFeatureList<Keypoint> queryKeypoints = engine.findFeatures(query.flatten());
LocalFeatureList<Keypoint> targetKeypoints = engine.findFeatures(target.flatten());
```

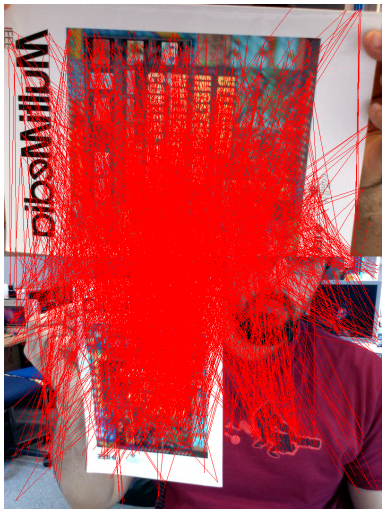
Once the engine is constructed, we can use it to extract `Keypoint` objects from our images. The `Keypoint` class contain a public field called `ivec` which, in the case of a standard SIFT descriptor is a 128 dimensional description of a patch of pixels around a detected point. Various distance measures can be used to compare `Keypoints` to `Keypoints`.

The challenge in comparing `Keypoints` is trying to figure out which `Keypoints` match between `Keypoints` from some query image and those from some target. The most basic approach is to take a given `Keypoint` in the query and find the `Keypoint` that is closest in the target. A minor improvement on top of this is to disregard those points which match well with MANY other points in the target. Such point are considered non-descriptive. Matching can be achieved in OpenIMAJ using the `BasicMatcher`. Next we'll construct and setup such a matcher:

```
LocalFeatureMatcher<Keypoint> matcher = new BasicMatcher<Keypoint>(80);
matcher.setModelFeatures(queryKeypoints);
matcher.findMatches(targetKeypoints);
```

We can now draw the matches between these two images found with this basic matcher using the `MatchingUtilities` class:

```
MBFImage basicMatches = MatchingUtilities.drawMatches(query, target, matcher.getMatches(), RGBColour.RED);
DisplayUtilities.display(basicMatches);
```



As you can see, the basic matcher finds many matches, many of which are clearly incorrect. A more advanced approach is to filter the matches based on a given geometric model. One way of achieving this in OpenIMAJ is to use a `ConsistentLocalFeatureMatcher` which given an internal matcher and a model fitter configured to fit a geometric model, finds which matches given by the internal matcher are consistent with respect to the model and are therefore likely to be correct.

To demonstrate this, we'll use an algorithm called Random Sample Consensus (RANSAC) to fit a geometric model called an **Affine transform** to the initial set of matches. This is achieved by iteratively selecting a random set of matches, learning a model from this random set and then testing the remaining matches against the learnt model.

Tip

An Affine transform models the transformation between two parallelograms.

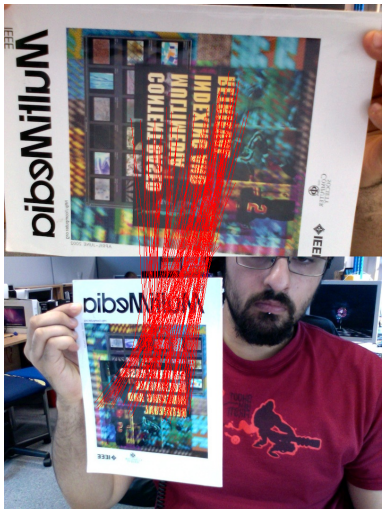
We'll now set up a RANSAC model fitter configured to find Affine Transforms (using the `RobustAffineTransformEstimator` helper class) and our consistent matcher:

```
RobustAffineTransformEstimator modelFitter = new RobustAffineTransformEstimator(5.0, 1500,
  new RANSAC.PercentageInliersStoppingCondition(0.5));
matcher = new ConsistentLocalFeatureMatcher2d<Keypoint>(
  new FastBasicKeypointMatcher<Keypoint>(8), modelFitter);

matcher.setModelFeatures(queryKeypoints);
matcher.findMatches(targetKeypoints);

MBFImage consistentMatches = MatchingUtilities.drawMatches(query, target, matcher.getMatches(),
  RGBColour.RED);

DisplayUtilities.display(consistentMatches);
```



The `AffineTransformModel` class models a two-dimensional Affine transform in OpenIMAJ. The `RobustAffineTransformEstimator` class provides a method `getModel()` which returns the internal Affine Transform model whose parameters are optimised during the fitting process driven by the `ConsistentLocalFeatureMatcher2d`. An interesting byproduct of using the `ConsistentLocalFeatureMatcher2d` is that the `AffineTransformModel` returned by `getModel()` contains the best transform matrix to go from the query to the target. We can take advantage of this by transforming the bounding box of our query with the transform estimated in the `AffineTransformModel`, therefore we can draw a polygon around the estimated location of the query within the target:

```
target.drawShape(
  query.getBounds().transform(modelFitter.getModel().getTransform().inverse()), 3,RGBColour.BLUE);
DisplayUtilities.display(target);
```

5.1. Exercises

5.1.1. Exercise 1: Different matchers

Experiment with different matchers; try the `BasicTwoWayMatcher` for example.

5.1.2. Exercise 2: Different models

Experiment with different models (such as a `HomographyModel`) in the consistent matcher. The `RobustHomographyEstimator` helper class can be used to construct an object that fits the `HomographyModel` model. You can also experiment with an alternative robust fitting algorithm to RANSAC called Least Median of Squares (LMedS) through the `RobustHomographyEstimator`.

Tip

A HomographyModel models a planar Homography between two planes. Planar Homographies are more general than Affine transforms and map quadrilaterals to quadrilaterals.

Chapter 6. Image Datasets

Datasets are an important concept in OpenIMAJ. Fundamentally, a dataset is a collection of data items. OpenIMAJ supports two types of dataset: `ListDataset`s and `GroupedDataset`s. As the name suggests, a `ListDataset` is basically like a list of data items and indeed the `ListDataset` class extends the java `List` interface. A `GroupedDataset` is essentially a keyed map of `Dataset`s and is an extension of the Java `Map` interface. The datasets classes are designed to provide a useful way of manipulating collections of items, and are particularly useful for applying machine-learning techniques to data as we'll see later in the tutorial.

This tutorial explores the use of datasets that contain images. OpenIMAJ contains methods and classes to help you efficiently deal with the construction and manipulation of image datasets (and indeed datasets of other types). To get started, create a new project using the Maven archetype, or add a new class to an existing OpenIMAJ Maven project and add a main method.

We'll start by looking at how you can create a simple list dataset from a directory of images you have on your computer's disk. If you don't have a directory of images to hand, create an empty one somewhere on your computer and add a couple of images to it. Now, add some code to your main method to construct an instance of a `VFSListDataset` as follows:

```
VFSListDataset<FImage> images =
  new VFSListDataset<FImage>("/path/to/image_dir", ImageUtilities.FIMAGE_READER);
```

In your code you'll need to replace the `/path/to/image_dir` string with the path to your directory of images. Notice that the dataset we've created is typed on the `FImage` class, and in the constructor we've passed a reference to `ImageUtilities.FIMAGE_READER`. This means that this dataset will contain grey-scale versions of the images on the disk (irrespective of whether they are actually colour images). The `ImageUtilities.FIMAGE_READER` is a special object called an `ObjectReader`. If you wanted to load colour images in your dataset, you would just need to change the type to `MBFImage`, and use the `ImageUtilities.MBFIMAGE_READER` `ObjectReader` instead.

As we mentioned earlier, a `ListDataset` extends a normal Java `List`, so you can do standard things like getting the number of items in the dataset:

```
System.out.println(images.size());
```

The dataset interface also allows you to easily get a random item from the dataset. As we're dealing with images, we can display a random image as follows:

```
DisplayUtilities.display(images.getRandomInstance(), "A random image from the dataset");
```

Also, because we're dealing with a list of images, we can display them all in a window as follows:

```
DisplayUtilities.display("My images", images);
```

The `VFSListDataset` class is very powerful. It can be used to create datasets from any kinds of data given an appropriate `ObjectReader` implementation. Beyond this, it is also able to create datasets from other sources, such as compressed archives containing data items, and even from remote data that is not stored on the local disk. Try running the following code which creates an image dataset from images in a zip file which is hosted on a web-server:

```
VFSListDataset<FImage> faces =
  new VFSListDataset<FImage>("zip:http://datasets.openimaj.org/att_faces.zip", ImageUtilities.FIMAGE_READER);
DisplayUtilities.display("ATT faces", faces);
```

As was mentioned in the introduction to this chapter, a grouped dataset maps a set of keys to sub-datasets. Grouped datasets are useful for things like machine-learning when you want to train classifiers to distinguish between groups. If you download and unzip the faces dataset that we used above (http://datasets.openimaj.org/att_faces.zip), you'll see that the images are actually grouped into directories, with all the images of a single individual stored in the same directory. When we loaded the list dataset from the zip file, we lost the associations between images of each individual. Using a `VFSGroupDataset` we can maintain the associations:

```
VFSGroupDataset<FImage> groupedFaces =
 new VFSGroupDataset<FImage>( "zip:http://datasets.openimaj.org/att_faces.zip", ImageUtilities.FIMAGE_READER);
```

Using the grouped dataset, we can iterate through the keys, which are actually created from the names of the directories containing the images, and display all the images from each individual in a window:

```
for (final Entry<String, VFSListDataset<FImage>> entry : groupedFaces.entrySet()) {
 DisplayUtilities.display(entry.getKey(), entry.getValue());
}
```

Sometimes, it can be useful to be able to dynamically create a dataset of images from the web. In the image analysis community, Flickr is often used as a source of tagged images for performing activities such as training classifiers. The `FlickrImageDataset` class makes it easy to dynamically construct a dataset of images from a Flickr search:

```
FlickrAPIToken flickrToken = DefaultTokenFactory.get(FlickrAPIToken.class);
FlickrImageDataset<FImage> cats =
  FlickrImageDataset.create(ImageUtilities.FIMAGE_READER, flickrToken, "cat", 10);
DisplayUtilities.display("Cats", cats);
```

The Flickr website requires you authenticate to use its API. The first time you run the above code, you will see instructions on obtaining a Flickr API key and secret, which you then have to enter at the prompt. Once you've done this once, the key and secret will be stored and automatically retrieved in the future by the `DefaultTokenFactory`. It is also possible to for-go the `DefaultTokenFactory` and construct a `FlickrAPIToken` and fill in the api key and secret field manually.

6.1. Exercises

6.1.1. Exercise 1: Exploring Grouped Datasets

Using the faces dataset available from `http://datasets.openimaj.org/att_faces.zip`, can you display an image that shows a randomly selected photo of each person in the dataset?

6.1.2. Exercise 2: Find out more about VFS datasets

`VFSListDatasets` and `VFSGroupDatasets` are based on a technology from the Apache Software Foundation called Commons Virtual File System (Commons VFS). Explore the documentation of the Commons VFS to see what other kinds of sources are supported for building datasets.

6.1.3. Exercise 3: Try the BingImageDataset dataset

The `BingImageDataset` class allows you to create a dataset of images by performing a search using the Bing search engine. The `BingImageDataset` class works in a similar way to the `FlickrImageDataset` described above. Try it out!

6.1.4. Exercise 4: Using MapBackedDataset

The `MapBackedDataset` class provides a concrete implementation of a `GroupedDataset`. See if you can use the static `MapBackedDataset.of` method to construct a grouped dataset of images of some famous people. Use a `BingImageDataset` to get the images of each person.

Part III. Video Fundamentals

Video analysis is a natural extension to image analysis. OpenIMAJ contains a number of classes and methods to help you work with video, covering everything from the application of image processing to individual frames to feature tracking across entire video sequences. Part III of the tutorial will provide you with the knowledge need to construct video objects from files and capture hardware, and demonstrate the basic foundations of applying processing and analysis algorithms to videos.

Chapter 7. Processing video

In this section we'll show you how to deal with videos using OpenIMAJ. We provide a set of tools for loading, displaying and processing various kinds of video.

All videos in OpenIMAJ are subtypes of the `Video` class. This class is typed on the type of underlying frame. In this case, let's create a video which holds coloured frames:

```
Video<MBFImage> video;
```

Exactly what kind of video is loaded depends on what you want to do. To load a video from a file we use the **Xuggle** library which internally uses `ffmpeg`. Let's load a video from a file (which you can download from here: http://static.openimaj.org/media/tutorial/keyboardcat.flv).

If we want to load a video from a file we use a `XuggleVideo` object:

```
video = new XuggleVideo(new File("/path/to/keyboardcat.flv"));
```

Tip

The `XuggleVideo` class also has constructors that let you pass a URL to a video on the web without downloading it first:

```
video = new XuggleVideo(new URL("http://static.openimaj.org/media/tutorial/keyboardcat.flv"));
```

If your computer has a camera, OpenIMAJ also supports live video input. These are called capture devices and you can use one through the `VideoCapture` class:

```
video = new VideoCapture(320, 240);
```

This will find the first video capture device attached to your system and render it as closely to 320 × 240 pixels as it can. To select a specific device you can use the alternative constructors and use the `VideoCapture.getVideoDevices()` static method to obtain the available devices.

To see if either of these kinds of video work, we can use `VideoDisplay` to display videos. This is achieved using the static function calls in `VideoDisplay` (which mirror those found in `DisplayUtilities` for images) like so:

```
VideoDisplay<MBFImage> display = VideoDisplay.createVideoDisplay(video);
```

Simply by creating a display, the video starts and plays. You can test this by running your app.

As with images, displaying them is nice but what we really want to do is process the frames of the video in some way. This can be achieved in various ways; firstly videos are `Iterable`, so you can do something like this to iterate through every frame and process it:

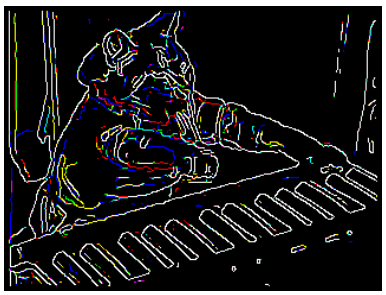
```
for (MBFImage mbfImage : video) {
    DisplayUtilities.displayName(mbfImage.process(new CannyEdgeDetector()), "videoFrames");
}
```

Here we're applying a Canny edge detector to each frame and displaying the frame in a named window. Another approach, which ties processing to image display automatically, is to use an event driven technique:

```
VideoDisplay<MBFImage> display = VideoDisplay.createVideoDisplay(video);
display.addVideoListener(
  new VideoDisplayListener<MBFImage>() {
    public void beforeUpdate(MBFImage frame) {
        frame.processInplace(new CannyEdgeDetector());
    }

    public void afterUpdate(VideoDisplay<MBFImage> display) {
    }
  });
```

These `VideoDisplayListener`s are given video frames before they are rendered and they are handed the video display after the render has occurred. The benefit of this approach is that functionality such as looping, pausing and stopping the video is given to you for free by the `VideoDisplay` class.



7.1. Exercises

7.1.1. Exercise 1: Applying different types of image processing to the video

Try a different processing operation and see how it affects the frames of your video.

Chapter 8. Finding faces

OpenIMAJ contains a set of classes that contain implementations of some of the state-of-the-art face detection and recognition algorithms. These classes are provided as a sub-project of the OpenIMAJ code-base called `faces`. The OpenIMAJ maven archetype adds the face library as a dependency and so we can start building face detection applications straight away.

Create a new application using the quick-start archetype (see tutorial 1) and import it into your IDE. If you look at the `pom.xml` file you will see that the `faces` dependency from OpenIMAJ is already included. As you've already done the video-processing tutorial, we'll try to find faces within the video that your cam produces. If you don't have a cam, follow the video tutorial on how to use video from a file instead.

Start by removing the code from the main method of the `App.java` class file. Then create a video capture object and a display to show the video. Create a listener on the video display to which we can hook our face finder. The code is below, but check out the previous tutorial on video processing if you're not sure what it means.

```
VideoCapture vc = new VideoCapture( 320, 240 );
VideoDisplay<MBFImage> vd = VideoDisplay.createVideoDisplay( vc );
vd.addVideoListener(
  new VideoDisplayListener<MBFImage>() {
    public void beforeUpdate( MBFImage frame ) {
    }

    public void afterUpdate( VideoDisplay<MBFImage> display ) {
    }
  });
```

For finding faces in images (or in this case video frames) we use a face detector. The `FaceDetector` interface provides the API for face detectors and there are currently two implementations within OpenIMAJ - the `HaarCascadeDetector` and the `SandeepFaceDetector`. The `HaarCascadeDetector` is considerably more robust than the `SandeepFaceDetector`, so we'll use that.

In the `beforeUpdate()` method, instantiate a new `HaarCascadeDetector`. The constructor takes the minimum size in pixels that a face can be detected at. For now, set this to `40` pixels:

```
FaceDetector<DetectedFace,FImage> fd = new HaarCascadeDetector(40);
```

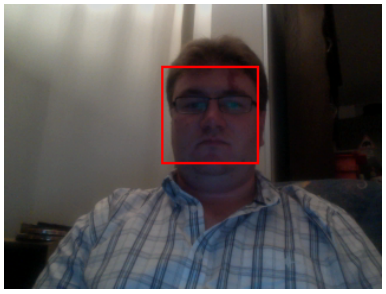
Like all `FaceDetector` implementations, the `HaarCascadeDetector` has a method `detectFaces()` which takes an image. Because the `HaarCascadeDetector` uses single band images, we must convert our multi-band colour image into a single band image. To do this we can use the `Transforms` utility class that contains some static methods for converting images. The `calculateIntensity()` method will do just fine. Note that functionally the `calculateIntensity()` method does the same thing as the `flatten()` method we used earlier when used on RGB images.

```
List<DetectedFace> faces = fd.detectFaces(Transforms.calculateIntensity(frame));
```

The `detectFaces()` method returns a list of `DetectedFace` objects which contain information about the faces in the image. From these objects we can get the rectangular bounding boxes of each face and draw them back into our video frame. As we're doing all this in our `beforeUpdate()` method, the video display will end up showing the bounding boxes on the displayed video. If you run the code and you have a cam attached, you should see yourself with a box drawn around your face. The complete code is shown below:

```
FaceDetector<DetectedFace,FImage> fd = new HaarCascadeDetector(40);
List<DetectedFace> faces = fd.detectFaces( Transforms.calculateIntensity(frame));

for( DetectedFace face : faces ) {
    frame.drawShape(face.getBounds(), RGBColour.RED);
}
```



OpenIMAJ has other face detectors which go a bit further than just finding the face. The `FKEFaceDetector` finds facial keypoints (the corners of the eyes, nose and mouth) and we can use this detector instead simply by instantiating that object instead of the `HaarCascadeDetector`. The `FKEFaceDetector` returns a slightly different object for each detected face, called a `KEDetectedFace`. The `KEDetectedFace` object contains the extra information about where the keypoints in the face are located. The lines of our code to instantiate the detector and detect faces can now be changed to the following:

```
FaceDetector<KEDetectedFace,FImage> fd = new FKEFaceDetector();
List<KEDetectedFace> faces = fd.detectFaces( Transforms.calculateIntensity( frame ) );
```

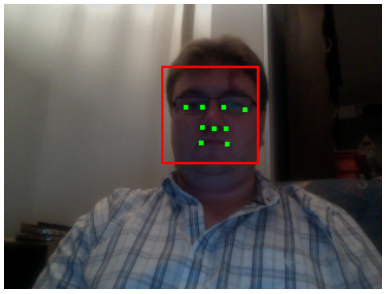
If you run the demo now, you will see exactly the same as before, as the `FKEFaceDetector` still detects bounding boxes. It may be running a bit slower though, as there is much more processing going on - we're just not seeing the output of it! So, let's plot the facial keypoints.

To get the keypoints use `getKeypoints()` on the detected face. Each keypoint has a position (public field) which is relative to the face, so we'll need to translate the point to the position of the face within the video frame before we plot the points. To do that we can use the `translate()` method of the `Point2d` class and the `minX()` and `minY()` methods of the `Rectangle` class.

8.1. Exercises

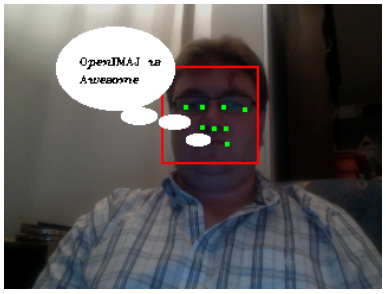
8.1.1. Exercise 1: Drawing facial keypoints

Use the information above to plot the facial keypoints on the video.



8.1.2. Exercise 2: Speech bubbles

Try and take the speech bubble from the previous image tutorial and make it come from the mouth in the video. **Hints:** use `getKeypoint(FacialKeypointType)` to get the keypoint of the left corner of the mouth and plot the ellipses depending on that point. You may need to use smaller ellipses and text if your video is running at 320x240.


OpenIMAJ is
Awesome

Part IV. Audio Fundamentals

OpenIMAJ started as a library for image processing, then expanded to video processing. The natural extension to dealing with videos is audio processing and has an important role in many applications of multimedia understanding. In this chapter we'll go through the basic audio subsystem in OpenIMAJ and show how to extract features from audio that can then be used in the various other tools in OpenIMAJ.

Chapter 9. Getting Audio and Basic Processing

Let's dive straight in and get something going before we look at some audio theory and more complex audio processing. One of the most useful things to do with audio is to listen to it! Playing an audio file in OpenIMAJ is very easy: simply create your audio source and pass it to the audio player.

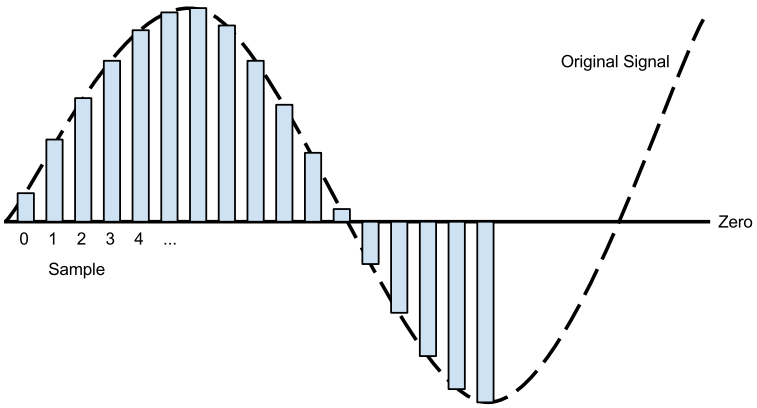
```
XuggleAudio xa = new XuggleAudio( new File( "myAudioFile.mp3" ) );
AudioPlayer.createAudioPlayer( xa ).run();
```

If you run these 2 lines of code you should hear audio playing. The `XuggleAudio` class uses the **Xuggler** library to decode the audio from the file. The audio player ()that's constructed using a static method as with the video player) returns an audio player instance which we set running straight away.

Tip

The `XuggleAudio` class also has constructors for reading audio from a URL or a stream.

What's happening underneath is that the Xuggler library decodes the audio stream into chunks of audio (called *frames*) each of which has many *samples*. A sample represents a level of sound pressure and the more of these there are within one second, the better the representation of the original continuous signal. The number of samples in one second is called the sample rate and you may already know that audio on CDs are encoded at 44,100 samples per second (or 44.1KHz). The maximum frequency that can be encoded in a digital signal is half of the sample rate (e.g. an estimate of a 22.05KHz sine wave with a 44.1KHz sampled signal will be {1,-1,1,-1...}). This is called the Nyquist frequency (named after Swedish-American engineer Harry Nyquist).

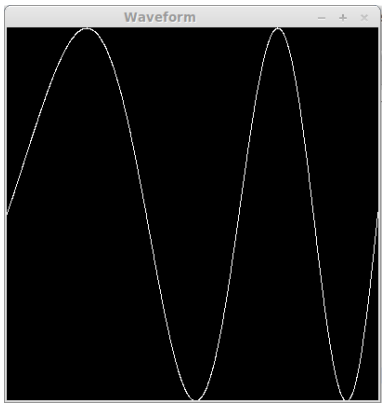


Let's have a look at the audio waveform. This is easy to do with OpenIMAJ as we have a subproject that contains various visualisers for various data including audio data. The `AudioWaveform` visualisation acts as a very basic oscilloscope for displaying the audio data. We'll use a file from `http://audiocheck.net/` as it's good for understanding some of the audio functions we're about to describe. We can link directly to the file by passing a URL to `XuggleAudio` (as in the code snippet below) or you can download the 20Hz-20KHz sweep and use it by passing a `File` to the `XuggleAudio`.

```
final AudioWaveform vis = new AudioWaveform( 400, 400 );
vis.showWindow( "Waveform" );

final XuggleAudio xa = new XuggleAudio(
    new URL( "http://www.audiocheck.net/download.php?" +
        "filename=Audio/audiocheck.net_sweep20-20klin.wav" ) );

SampleChunk sc = null;
while( (sc = xa.nextSampleChunk()) != null )
    vis.setData( sc.getSampleBuffer() );
```



So, the first two lines above create the visualisation. We open the file and then we iterate through the audio stream (with `xa.nextSampleChunk()`) and send that data to the visualisation (we'll cover the `getSampleBuffer()` method later).

The audio subsystem in OpenIMAJ has been designed to match the programming paradigm of the image and video subprojects. So, all classes providing audio extend the `Audio` class. Currently all implementations also extend the `AudioStream` class which defines a method for getting frames of audio from the stream which we call `SampleChunks` in OpenIMAJ. A `SampleChunk` is a wrapper around an array of bytes. Understanding what those bytes mean requires knowledge of the format of the audio data and this is given by the `AudioFormat` class.

Audio data, like image data, can come in many formats. Each digitised reading of the sound pressure (the sample) can be represented by 8 bits (1 byte, signed or unsigned), 16 bits (2 bytes, little or big endian, signed or unsigned), or 24 bits or more. The sample rate can be anything, although 22.05KHz or 44.1KHz is common for audio (48KHz for video). The audio data can also represent one (mono), two (stereo) or more channels of audio, which are interleaved in the sample chunk data.

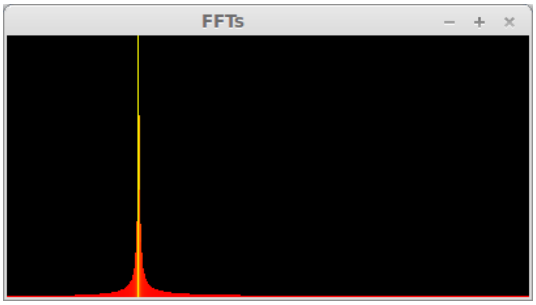
To make code agnostic to the audio format, OpenIMAJ has a API that provides a means for accessing the sample data in a consistent way. This class is called a `SampleBuffer`. It has a `get(index)` method which returns a sample as a value between `0..1` whatever the underlying size of the data. It also provides a `set(index,val)` method which provides the opposite conversion. Multichannel audio is still interleaved in the `SampleBuffer`, however, it does provide various accessors for getting data from specific channels. An appropriate `SampleBuffer` for the audio data in a `SampleChunk` can be retrieved using `SampleChunk.getSampleBuffer()`.

Ok, enough theory for the moment. Let's do something interesting that will help us towards understanding what we're getting in.

An algorithm called the Fourier Transform converts a time-domain signal (i.e. the signal you're getting from the audio file) into a frequency-domain signal (describing what pitches or frequencies contribute towards the final signal). We can see what frequencies are in our signal by applying the transform and visualising the results.

Take the previous code and change the visualisation to be a `BarVisualisation`. Next, we'll create a `FourierTransform` object and take our stream of data from there.

```
FourierTransform fft = new FourierTransform( xa );
...
while( (sc = fft.nextSampleChunk()) != null )
{
    float[][] fftData = fft.getMagnitudes();
    vis.setData( fftData[0] );
}
```

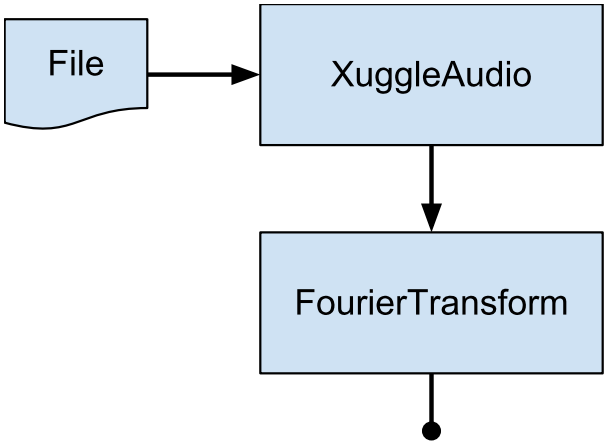


Run this demo on the audiocheck.net sine wave sweep and you'll see a peak in the graph moving up through the frequencies. The lowest frequencies are on the left of the visualisation and the highest frequencies on the right (the Nyquist frequency on the far right).

Tip

Try using the `AudioSpectrogram` visualisation which displays the FFT in a slightly different way. It plots the frequencies vertically, with the pixel intensities as the amplitude of the frequency. The spectrogram expects values between 0 and 1 so you will need to get normalised magnitudes from the Fourier processor: using `fft.getNormalisedMagnitudes( 1f/Integer.MAX_VALUE )`. You should see a nice line being drawn through the frequencies.

This example also introduces us to the processor chaining in the OpenIMAJ audio system. Chaining allows us to create a set of operations to apply to the audio data and to take the final data stream from the end. In this case we have chained the `FourierTransform` processor to the original audio stream and we're taking the data from the end of the stream, as shown in the diagram below.



When we call `nextSampleChunk()` on the `FourierTransform` object, it goes and gets the sample chunk from the previous processor in the chain, processes the sample chunk and returns a new sample chunk (in fact, the `FourierTransform` returns the sample chunk unchanged).

Let's put an EQ filter in the chain that will filter out frequencies from the original signal:

```
EQFilter eq = new EQFilter( xa, EQType.LPF, 5000 );
FourierTransform fft = new FourierTransform( eq );
```

We have set the low-pass filter (only lets low frequencies through) to 5KHz (5000Hz), so when you run the program again, you will see the peak fall off some way along its trip up to the high frequencies. This sort of filtering can be useful in some circumstances for directing processing to specific parts of the audio spectrum.

Tip

As the frequency peak falls off, the bar visualisation will rescale to fit and it might not be too easy to see what's going on. Try disabling the automatic scaling on the bar visualisation and set the maximum value to be fixed around 1E12:

```
vis.setAutoScale( false );
vis.setMaximumValue( 1E12 );
```

So, we've learned a bit about audio, seen the basics of the audio subsystem in OpenIMAJ and even started looking into the audio. In the next chapter, we'll start extracting features and trying to do something interesting with it, bringing in other parts of the OpenIMAJ system.

9.1. Exercises

9.1.1. Exercise 1: Spectrogram on Live Sound

Make the application display the audio spectrogram of the live sound input from your computer. You can use the `JavaSoundAudioGrabber` class in a separate thread to grab audio from your computer. When you talk or sing into the computer can you see the pitches in your voice? How does speech compare to other sounds?

Chapter 10. Feature Extraction from Audio

Just like images, we can extract features that can be used to get a higher-level understanding of the audio. There are some features that have become de-facto in audio processing, and one of these is the Mel-Frequency Cepstrum Coefficients (MFCCs). They give an overview of the shape (or envelope) of the frequency components of the audio based on some perceptual scaling of the frequency domain.

OpenIMAJ provides an MFCC class based around the **jAudio** implementation. Unsurprisingly, it's called `MFCC`! We can use it in exactly the same way as the FFT processor, so if you take the code from FFT example in the previous chapter you can change the FFT processor to be the MFCC processor.

```
MFCC mfcc = new MFCC( xa );
...
while( (sc = mfcc.nextSampleChunk()) != null )
{
 double[][] mfccs = mfcc.getLastGeneratedFeature();
 vis.setData( mfccs[0] );
}
```

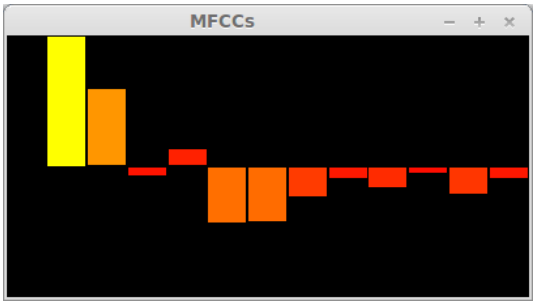
MFCCs were specifically developed for speech recognition tasks and so are much more suitable for describing speech signals than a sine wave sweep. So, let's switch to using the `JavaSoundAudioGrabber` so we can speak into the computer. Secondly, we'll fix the analysis window that we're using. The literature shows that 30ms windows with 10ms overlaps are often used in speech processing. At 44.1KHz, 10ms is 441 samples, so we'll use the `FixedSizeSampleAudioProcessor` to deal with giving us the appropriate size sample chunks.

```
JavaSoundAudioGrabber jsag = new JavaSoundAudioGrabber( new AudioFormat( 16, 44.1, 1 ) );
FixedSizeSampleAudioProcessor fssap = new FixedSizeSampleAudioProcessor( jsag, 441*3, 441 );
```

You should see that when you speak into the computer, the MFCCs show a noticeable change compared to the sine wave sweep.

Tip

You might want to try fixing the axis of the visualisation bar graph using the method `setAxisLocation( 100 )`.



10.1. Exercises

10.1.1. Exercise 1: Spectral Flux

Update the program to use the SpectralFlux feature. Set the bar visualisation to use a maximum value of 0.0001. What do you think this feature is showing? How would this feature be useful?

Part V. Streaming Data Fundamentals

The analysis of streaming multimedia data is becoming a hot research topic. In OpenIMAJ we've built a framework that aims to make it easy to work with streams of all kinds of data. This even includes some classes that connect to live data sources such as Twitter. The framework even provides a number of strategies to seamlessly deal with streams of data that arrive at a rate faster than they can be consumed. The tutorials in this section will get you up to speed using the OpenIMAJ stream classes and show you how to use them to build some practical applications.

Chapter 11. Twitter Streams and Images

This tutorial will show you how to extract and analyse images posted on Twitter in real-time. To get started, create a new project using the OpenIMAJ archetype and in the main method, create a connection to the live Twitter sample stream as follows:

```
TwitterAPIToken token = DefaultTokenFactory.get(TwitterAPIToken.class);
TwitterStreamDataset stream = new TwitterStreamDataset(token);
```

At this point, run the code. The first time you run it you will be instructed to register as a Twitter developer to get an API key, which you will then need to enter at the prompt. If you've followed the Image Datasets tutorial you should be familiar with this process from using the `FlickrImageDataset` and `BingImageDataset` classes; as with those classes you could also construct a `TwitterAPIToken` and set its fields manually. You'll notice that not much happens other than a few debug messages. Also, notice that the program doesn't end; this is because there is a thread running in the background reading tweets into your `stream` object. Manually stop the program running.

To demonstrate that Tweets are really being added into your `stream` object, add the following code to print the text content of each Tweet and run it again:

```
stream.forEach(new Operation<Status>() {
    public void perform(Status status) {
        System.out.println(status.getText());
    }
});
```

You should now see a large volume of Tweet messages being written to your console. Stop the program and remove the above `forEach` loop. We'll now look at how we can get images out of the Tweets.

Tweets themselves do not contain images; rather tweets might contain URLs, which might correspond to images, or web-sites where an image is hosted. URLs might be in the textual body of the Tweets and/or in special fields that form part of the status objects. OpenIMAJ makes it easy to extract the URLs by *mapping* a stream of Twitter `Status` objects to `URL` objects:

```
Stream<URL> urlStream = stream.map(new TwitterURLExtractor());
```

The `ImageSiteURLExtractor` class can be used to process the URLs and return just those that correspond to images. The `ImageSiteURLExtractor` is also aware of a number of standard image hosting sites, and is able to resolve the actual image URL from the web-page URL that would normally appear in a Tweet. The `ImageSiteURLExtractor` is just another `Function` so can be applied as another map:

```
Stream<URL> imageUrlStream = urlStream.map(new ImageSiteURLExtractor(false));
```

The boolean in the `ImageSiteURLExtractor` constructor disables support for the Tumblr hosting site; you can enable it, but you'll need to create a Tumblr API key (which you will be prompted for). Now to get the images, we can apply a static instance of function class called `ImageFromURL` which has been configured to read `MBFImages`:

```
Stream<MBFImage> imageStream = imageUrlStream.map(ImageFromURL.MBFIMAGE_EXTRACTOR);
```

Now let's display the images in the stream as they arrive:

```
imageStream.forEach(new Operation<MBFImage>() {
    public void perform(MBFImage image) {
        DisplayUtilities.displayName(image, "image");
    }
});
```

If at this point you run the program, you should see be able to see the images that are currently being shared on Twitter.

Now let's modify the code so we can generate a visualisation of the *faces* that appear on Twitter. Add an extra inline map function before the previous `forEach` loop, so that the code looks like this:

```
imageStream.map(new MultiFunction<MBFImage, MBFImage>() {
    public List<MBFImage> apply(MBFImage in) {

    }
}).forEach(new Operation<MBFImage>() {
    public void perform(MBFImage image) {
        DisplayUtilities.displayName(image, "image");
    }
});
```

and add the following to the apply method:

```
List<DetectedFace> detected = detector.detectFaces(in.flatten());

List<MBFImage> faces = new ArrayList<MBFImage>();
for (DetectedFace face : detected)
    faces.add(in.extractROI(face.getBounds()));

return faces;
```

and finally add the following just before the apply method:

```
 HaarCascadeDetector detector = HaarCascadeDetector.BuiltInCascade.frontalface_default.load();
```

Now, if you run the software, you should be able to to see the faces of people who are being tweeted at the current time.

11.1. Exercises

11.1.1. Exercise 1: The TwitterSearchStream

The `TwitterSearchStream` class repeatedly calls the Twitter search API with a specific query in order to produce a stream of Tweets related to a specific subject. Try using the `TwitterSearchStream` to find and display tweeted images of your favourite animal.

11.1.2. Exercise 2: The colour of Twitter

Can you make a program that continuously shows the average colour of the last 100 tweeted images?

11.1.3. Exercise 3: Trending images

Images are often (re)-tweeted or shared. Using the histogram features and similarity measures you learned about in the Global image features tutorial, can you make a program that computes which images have been re-tweeted the most over a fixed number of tweets?

Part VI. Machine Learning Fundamentals

Chapter 12. Classification with Caltech 101

In this tutorial, we'll go through the steps required to build and evaluate a near state-of-the-art image classifier. Although for the purposes of this tutorial we're using features extracted from images, everything you'll learn about using classifiers can be applied to features extracted from other forms of media.

To get started you'll need a new class in an existing OpenIMAJ project, or a new project created with the archetype. The first thing we need is a dataset of images with which we'll work. For this tutorial we'll use a well known set of labelled images called the Caltech 101 dataset[1]. The Caltech 101 dataset contains labelled images of 101 object classes together with a set of background images. OpenIMAJ has built in support for working with the Caltech 101 dataset, and will even automatically download the dataset for you. To use it, enter the following code:

```
GroupedDataset<String, VFSListDataset<Record<FImage>>, Record<FImage>> allData =
   Caltech101.getData(ImageUtilities.FIMAGE_READER);
```

You'll remember from the image datasets tutorial that `GroupedDataset`s are Java `Map`s with a few extra features. In this case, our `allData` object is a `GroupedDataset` with `String` keys and the values are lists (actually `VFSListDataset`s) of `Record` objects which are themselves typed on `FImage`s. The `Record` class holds metadata about each Caltech 101 image. `Record`s have a method called `getImage()` that will return the actual image in the format specified by the generic type of the `Record` (i.e. `FImage`).

For this tutorial we'll work with a subset of the classes in the dataset to minimise the time it takes our program to run. We can create a subset of groups in a `GroupedDataset` using the `GroupSampler` class:

```
GroupedDataset<String, ListDataset<Record<FImage>>, Record<FImage>> data =
   GroupSampler.sample(allData, 5, false);
```

This basically creates a new dataset called `data` from the first 5 classes in the `allData` dataset. To do an experimental evaluation with the dataset we need to create two sets of images: a **training** set which we'll use to learn the classifier, and a **testing** set which we'll evaluate the classifier with. The common approach with the Caltech 101 dataset is to choose a number of training and testing instances for each class of images. Programatically, this can be achieved using the `GroupedRandomSplitter` class:

```
GroupedRandomSplitter<String, Record<FImage>> splits =
   new GroupedRandomSplitter<String, Record<FImage>>(data, 15, 0, 15);
```

In this case, we've created a training dataset with 15 images per group, and 15 testing images per group. The zero in the constructor is the number of validation images which we won't use in this tutorial. If you take a look at the `GroupedRandomSplitter` class you'll see there are methods to get the training, validation and test datasets.

Our next step is to consider how we're going to extract suitable image features. For this tutorial we're going to use a technique commonly known as the Pyramid Histogram of Words (**PHOW**). PHOW is itself based on the idea of extracting **Dense SIFT** features, quantising the SIFT features into **visual words** and then building **spatial histograms** of the visual word occurrences.

The Dense SIFT features are just like the features you used in the "SIFT and feature matching" tutorial, but rather than extracting the features at interest points detected using a difference-of-Gaussian, the features are extracted on a regular grid across the image. The idea of a visual word is quite simple: rather than representing each SIFT feature by a 128 dimension feature vector, we represent it by an identifier. Similar features (i.e. those that have similar, but not necessarily the same, feature vectors) are assigned to have the same identifier. A common approach to assigning identifiers to features is to train a **vector quantiser** (just another fancy name for a type of *classifier*) using k-means, just like we did in the "Introduction to clustering" tutorial. To build a histogram of visual words (often called a **Bag of Visual Words**), all we have to do is count up how many times each identifier appears in an image and store the values in a histogram. If we're building spatial histograms, then the process is the same, but we effectively

[1] http://www.vision.caltech.edu/Image_Datasets/Caltech101/

cut the image into blocks and compute the histogram for each block independently before concatenating the histograms from all the blocks into a larger histogram.

To get started writing the code for the PHOW implementation, we first need to construct our Dense SIFT extractor - we're actually going to construct two objects: a `DenseSIFT` object and a `PyramidDenseSIFT` object:

```
DenseSIFT dsift = new DenseSIFT(5, 7);
PyramidDenseSIFT<FImage> pdsift = new PyramidDenseSIFT<FImage>(dsift, 6f, 7);
```

The `PyramidDenseSIFT` class takes a normal `DenseSIFT` instance and applies it to different sized windows on the regular sampling grid, although in this particular case we're only using a single window size of 7 pixels.

The next stage is to write some code to perform **K-Means** clustering on a sample of SIFT features in order to build a `HardAssigner` that can assign features to identifiers. Let's wrap up the code for this in a new method that takes as input a dataset and a `PyramidDenseSIFT` object:

```
static HardAssigner<byte[], float[], IntFloatPair> trainQuantiser(
            Dataset<Record<FImage>> sample, PyramidDenseSIFT<FImage> pdsift)
{
    List<LocalFeatureList<ByteDSIFTKeypoint>> allkeys = new ArrayList<LocalFeatureList<ByteDSIFTKeypoint>>();

    for (Record<FImage> rec : sample) {
        FImage img = rec.getImage();

        pdsift.analyseImage(img);
        allkeys.add(pdsift.getByteKeypoints(0.005f));
    }

    if (allkeys.size() > 10000)
        allkeys = allkeys.subList(0, 10000);

    ByteKMeans km = ByteKMeans.createKDTreeEnsemble(300);
    DataSource<byte[]> datasource = new LocalFeatureListDataSource<ByteDSIFTKeypoint, byte[]>(allkeys);
    ByteCentroidsResult result = km.cluster(datasource);

    return result.defaultHardAssigner();
}
```

The above method extracts the first 10000 dense SIFT features from the images in the dataset, and then clusters them into 300 separate classes. The method then returns a `HardAssigner` which can be used to assign SIFT features to identifiers. To use this method, add the following to your main method after the `PyramidDenseSIFT` construction:

```
HardAssigner<byte[], float[], IntFloatPair> assigner =
   trainQuantiser(GroupedUniformRandomisedSampler.sample(splits.getTrainingDataset(), 30), pdsift);
```

Notice that we've used a `GroupedUniformRandomisedSampler` to get a random sample of 30 images across all the groups of the training set with which to train the quantiser. The next step is to write a `FeatureExtractor` implementation with which we can train our classifier:

```
static class PHOWExtractor implements FeatureExtractor<DoubleFV, Record<FImage>> {
    PyramidDenseSIFT<FImage> pdsift;
    HardAssigner<byte[], float[], IntFloatPair> assigner;

    public PHOWExtractor(PyramidDenseSIFT<FImage> pdsift, HardAssigner<byte[], float[], IntFloatPair> assigner)
    {
        this.pdsift = pdsift;
        this.assigner = assigner;
    }

    public DoubleFV extractFeature(Record<FImage> object) {
        FImage image = object.getImage();
        pdsift.analyseImage(image);

        BagOfVisualWords<byte[]> bovw = new BagOfVisualWords<byte[]>(assigner);

        BlockSpatialAggregator<byte[], SparseIntFV> spatial = new BlockSpatialAggregator<byte[], SparseIntFV>(
                bovw, 2, 2);

        return spatial.aggregate(pdsift.getByteKeypoints(0.015f), image.getBounds()).normaliseFV();
    }
}
```

This class uses a `BlockSpatialAggregator` together with a `BagOfVisualWords` to compute 4 histograms across the image (by breaking the image into 2 both horizontally and vertically). The `BagOfVisualWords` uses the `HardAssigner` to assign each Dense SIFT feature to a visual word and the compute the histogram. The resultant spatial histograms are then appended together and normalised before being returned. Back in the main method of our code we can construct an instance of our PHOWExtractor:

```
FeatureExtractor<DoubleFV, Record<FImage>> extractor = new PHOWExtractor(pdsift, assigner);
```

Now we're ready to construct and train a classifier - we'll use the linear classifier provided by the `LiblinearAnnotator` class:

```
LiblinearAnnotator<Record<FImage>, String> ann = new LiblinearAnnotator<Record<FImage>, String>(
              extractor, Mode.MULTICLASS, SolverType.L2R_L2LOSS_SVC, 1.0, 0.00001);
ann.train(splits.getTrainingDataset());
```

Finally, we can use the OpenIMAJ evaluation framework to perform an automated evaluation of our classifier's accuracy for us:

```
ClassificationEvaluator<CMResult<String>, String, Record<FImage>> eval =
   new ClassificationEvaluator<CMResult<String>, String, Record<FImage>>(
    ann, splits.getTestDataset(), new CMAnalyser<Record<FImage>, String>(CMAnalyser.Strategy.SINGLE));

Map<Record<FImage>, ClassificationResult<String>> guesses = eval.evaluate();
CMResult<String> result = eval.analyse(guesses);
```

12.1. Exercises

12.1.1. Exercise 1: Apply a Homogeneous Kernel Map

A Homogeneous Kernel Map transforms data into a compact linear representation such that applying a linear classifier approximates, to a high degree of accuracy, the application of a non-linear classifier over the original data. Try using the `HomogeneousKernelMap` class with a `KernelType.Chi2` kernel and `WindowType.Rectangular` window on top of the `PHOWExtractor` feature extractor. What effect does this have on performance?

Tip

Construct a `HomogeneousKernelMap` and use the `createWrappedExtractor()` method to create a new feature extractor around the `PHOWExtractor` that applies the map.

12.1.2. Exercise 2: Feature caching

The `DiskCachingFeatureExtractor` class can be used to cache features extracted by a `FeatureExtractor` to disk. It will generate and save features if they don't exist, or read from disk if they do. Try to incorporate the `DiskCachingFeatureExtractor` into your code. You'll also need to save the `HardAssigner` using `IOUtils.writeToFile` and load it using `IOUtils.readFromFile` because the features must be kept with the same `HardAssigner` that created them.

12.1.3. Exercise 3: The whole dataset

Try running the code over all the classes in the Caltech 101 dataset. Also try increasing the number of visual words to 600, adding extra scales to the `PyramidDenseSIFT` (try [4, 6, 8, 10] and reduce the step-size of the DenseSIFT to 3), and instead of using the `BlockSpatialAggregator`, try the `PyramidSpatialAggregator` with [2, 4] blocks. What level of classifier performance does this achieve?

Part VII. Facial Analysis

OpenIMAJ contains a number of tools for face detection, recognition and similarity comparison. In particular, OpenIMAJ implements a fairly standard recognition pipeline. The pipeline consists of four stages:

1. Face Detection
2. Face Alignment
3. Facial Feature Extraction
4. Face Recognition/Classification

Each stage of the pipeline is configurable, and OpenIMAJ contains a number of different options for each stage as well as offering the possibility to easily implement more. The pipeline is designed to allow researchers to focus on a specific area of the pipeline without having to worry about the other components. At the same time, it is fairly easy to modify and evaluate a complete pipeline.

In addition to the parts of the recognition pipeline, OpenIMAJ also includes code for tracking faces in videos and comparing the similarity of faces.

Bear in mind that as with all computer vision techniques, because of the variability of real-world images, each stage of the pipeline has the potential to fail.

1. Face Detection

The first stage of the pipeline is face detection. Given an image, a face detector attempts to localise all the faces in the image. All OpenIMAJ face detectors are subclasses of `FaceDetector`, and they all produce subclasses of `DetectedFace` as their output. Currently, OpenIMAJ implements the following types of face detector:

- `org.openimaj.image.processing.face.detection.SandeepFaceDetector`: A face detector that searches the image for areas of skin-tone that have a height/width ratio similar to the golden ratio. The detector will only find faces that are upright in the image (or upside-down).
- `org.openimaj.image.processing.face.detection.HaarCascadeDetector`: A face detector based on the classic Viola-Jones classifier cascade framework. The classifier comes with a number of pre-trained models for frontal and side face views. The classifier is only mildly invariant to rotation, and it won't detect non-upright faces.
- `org.openimaj.image.processing.face.keypoints.FKEFaceDetector`: The Frontal Keypoint Enhanced (FKE) Face Detector is not actually a detector in it's own right, but rather a wrapper around the `HaarCascadeDetector`. The FKE provides additional information about a face detection by finding facial keypoints on top of the face. The facial keypoints are located at stable points on the face (sides of the eyes, bottom of the nose, corners of the mouth). The facial keypoints can be used for alignment and feature extraction as described in the next section.
- `org.openimaj.image.processing.face.detection.CLMFaceDetector`: The Constrained Local Model (CLM) face detector uses an underlying `HaarCascadeDetector` to perform an initial face detection and then fits a statistical 3D face model to the detected region. The 3D face model can be used to locate facial keypoints within the image and also to determine the pose of the face. The model is a form of parameterised statistical shape model called a "point distribution model"; this means that the 3D model has an associated set of parameters which control elements of its shape (i.e. there are parameters that determine whether the mouth is open or closed, or how big the nose is). During the process of fitting the model to the image, these parameters are learned automatically, and are exposed through the detections (`CLMDetectedFaces`) returned by the `CLMFaceDetector`.

- `org.openimaj.image.processing.face.detection.IdentityFaceDetector`: The identity face detector just returns a single detection for each input image it is given. The detection covers the entire area of the input image. This is useful for working with face datasets that contain pre-extracted and cropped faces.

2. Face Alignment

Many forms of face recogniser work better if the facial image patches used for training and querying are aligned to a common view. This alignment allows for the recognition system to concentrate on the appearance of the face without having to explicitly deal with variations in the pose of the face. The `FaceAligner`s take in faces detected by a `FaceDetector` as input, and output an image with the aligned face rendered within it.

OpenIMAJ contains a number of face alignment options. Currently, these include:

- `org.openimaj.image.processing.face.alignment.AffineAligner`: An aligner that can align faces detected by the `FKEFaceDetector` to a neutral pose by applying an rigid affine transformation estimated from the mapping of facial keypoints in the detected image to the points in a model with neutral pose.
- `org.openimaj.image.processing.face.alignment.CLMAligner`: An aligner that warps a face detected by the `CLMFaceDetector` to a neutral pose. The alignment is non-rigid and warps each corresponding triangle of the detected face to a model with neutral pose.
- `org.openimaj.image.processing.face.alignment.IdentityAligner`: The identity aligner does no alignment; it just returns the cropped face image from the detector. This is useful when working with face datasets that contain pre-aligned images.
- `org.openimaj.image.processing.face.alignment.MeshWarpAligner`: The mesh warp aligner performs a similar job to the `CLMAligner`, but for `FKEDetectedFace`s detected by the `FKEFaceDetector`. A mesh is constructed over the set of detected facial keypoints and a non-linear warp is applied to project each keypoint to a canonical position within a neutral pose.
- `org.openimaj.image.processing.face.alignment.RotateScaleAligner`: The rotate and scale aligner maps faces detected by the `FKEFaceDetector` to an aligned pose by performing a rotation followed by a scaling.
- `org.openimaj.image.processing.face.alignment.ScalingAligner`: The scaling aligner takes any type of `DetectedFace` and scales the cropped face image in the detection to a fixed size.

3. Facial Feature Extraction

Once you have detected a face (and possibly chosen an aligner for it), you need to extract a feature which you can then use for recognition or similarity comparison. As with the detection and alignment phases, OpenIMAJ contains a number of different implementations of `FacialFeatureExtractor`s which produce `FacialFeature`s together with methods for comparing pairs of `FacialFeature`s in order to get a similarity measurement. The currently implemented `FacialFeature`s are listed below:

- `CLMPoseFeature`: A feature that represents the pose of a face detected with the `CLMFaceDetector`. The pose consists of the pitch, roll and yaw of the face. The feature can expose itself as a `DoubleFV` and can be compared using a `FaceFVComparator`.
- `CLMPoseShapeFeature`: A feature that represents the shape parameters and pose of a face detected with the `CLMFaceDetector`. The shape vector describes the shape of the face as a small set of variables, and the pose consists of the pitch, roll and yaw of the face. The feature can expose itself as a `DoubleFV` and can be compared using a `FaceFVComparator`.
- `CLMShapeFeature`: A feature that represents the shape parameters of a face detected with the `CLMFaceDetector`. The shape vector describes the shape of the face as a small set of variables. The feature can expose itself as a `DoubleFV` and can be compared using a `FaceFVComparator`.

- `DoGSIFTFeature`: A feature built by detecting local interest points on the face using a Difference of Gaussian pyramid, and then describing these using SIFT features. The `DoGSIFTFeatureComparator` can be used to compare these features.
- `EigenFaceFeature`: A feature built by projecting the pixels of an aligned face into a lower-dimensional space learned through PCA. The feature extractor must be "trained" with a set of example aligned faces before it can be used. This forms the core of the classic Eigen Faces algorithm. The feature can expose itself as a `DoubleFV` and can be compared using a `FaceFVComparator`.
- `FaceImageFeature`: A feature built directly from the pixels of an aligned face. No normalisation is performed. The feature can expose itself as a `FloatFV` and can be compared using a `FaceFVComparator`.
- `FacePatchFeature`: A feature built by concatenating the pixels from each of the normalised circular patches around each facial keypoint from an `FKEDetectedFace`. The feature can expose itself as a `FloatFV` and can be compared using a `FaceFVComparator`.
- `FisherFaceFeature`: A feature built by projecting the pixels of an aligned face into a lower-dimensional space learned through Fisher's Linear Discriminant Analysis. The feature extractor must be "trained" with a set of example aligned faces before it can be used. This forms the core of the classic Fisher Faces algorithm. The feature can expose itself as a `DoubleFV` and can be compared using a `FaceFVComparator`.
- `LocalLBPHistogram`: Feature constructed by breaking the image into blocks and computing histograms of Local Binary Patterns (LBPs) for each block. All histograms are concatenated to form the final feature. The feature can expose itself as a `FloatFV` and can be compared using a `FaceFVComparator`.
- `LtpDtFeature`: A feature built from Local Trinary Patterns (LTPs) within an aligned image. The features are constructed to be compared using a Euclidean Distance Transform with the `LtpDtFeatureComparator` or `ReversedLtpDtFeatureComparator` comparators.

4. Face Recognition/Classification

The final stage of the pipeline uses extracted `FacialFeatures` to perform face recognition (determining who's face it is) or classification (determining some characteristic of the face; for example male/female, glasses/no-glasses, etc). All recognisers/classifiers are instances of `FaceRecogniser`. There are a couple of default implementations, but the most common is the `AnnotatorFaceRecogniser` which can use any form of `IncrementalAnnotator` to perform the actual classification. There are also specific recognisers for the Eigen Face and Fisher Faces algorithms that can be constructed with internal recognisers (usually a `AnnotatorFaceRecogniser`) that perform specific machine learning operations. All `FaceRecognisers` are capable of serialising and deserialising their internal state to disk. All recognisers are also capable of incremental learning (i.e. new examples can be added at any point).

Currently, there are implementations of `IncrementalAnnotator` that implement common machine-learning algorithms including k-nearest-neighbours and naive-bayes. Batch annotators (`BatchAnnotators`), such as a Support Vector Machine annotator can also be used by using an adaptor to convert the `BatchAnnotator` into an `IncrementalAnnotator` (for example a `InstanceCachingIncrementalBatchAnnotator`).

The face detection and recognition components can be managed separately, however, the `FaceRecognitionEngine` class can be used to simplify usage.

5. Face Similarity

The `FaceSimilarityEngine` class provides methods for assessing the similarity of faces by comparing `FacialFeatures` using appropriate comparators.

Chapter 13. Face recognition 101: Eigenfaces

Before we get started looking at the rich array of tools OpenIMAJ offers for working with faces, lets first look at how we can implement one of the earliest successful face recognition algorithms called "Eigenfaces". The basic idea behind the Eigenfaces algorithm is that face images are "projected" into a low dimensional space in which they can be compared efficiently. The hope is that intra-face distances (i.e. distances between images of the same person) are smaller than inter-face distances (the distance between pictures of different people) within the projected space (although there is no algorithmic guarantee of this). Fundamentally, this projection of the image is a form of feature extraction, similar to what we've seen in previous chapters of this tutorial. Unlike the extractors we've looked at previously however, for Eigenfaces we actually have to "learn" the feature extractor from the image data. Once we've extracted the features, classification can be performed using any standard technique, although 1-nearest-neighbour classification is the standard choice for the Eigenfaces algorithm.

The lower dimensional space used by the Eigenfaces algorithm is actually learned through a process called Principle Component Analysis (PCA), although sometimes you'll also see this referred to as the *discrete Karhunen–Loève transform*. The PCA algorithm finds a set of orthogonal axes (i.e. axes at right angles) that best describe the variance of the data such that the first axis is oriented along the direction of highest variance. It turns out that computing the PCA boils down to performing a well-know mathematical technique called the eigendecomposition (hence the name Eigenfaces) on the covariance matrix of the data. Formally, the eigendecomposition factorises a matrix, A, into a canonical form such that $Av = \lambda v$, where v is a set of vectors called the eigenvectors, and each vector is paired with a scalar from λ called an eigenvalue. The eigenvectors form a mathematical *basis*; a set of right-angled vectors that can be used as axes in a space. By picking a subset of eigenvectors with the largest eigenvalues it is possible to create a *basis* that can approximate the original space in far fewer dimensions.

The Eigenfaces algorithm is simple to implement using OpenIMAJ using the `EigenImages` class. The `EigenImages` class automatically deals with converting the input images into vectors and zero-centering them (subtracting the mean) before applying PCA.

Eigenfaces will really only work well on (near) full-frontal face images. In addition, because of the way Eigenfaces works, the face images we use must all be the same size, and must be aligned (typically such that the eyes of each subject must be in the same pixel locations). For the purposes of this tutorial we'll use a dataset of approximately aligned face images from the AT&T "The Database of Faces" (formerly "The ORL Database of Faces")[1]. Start by creating a new OpenIMAJ project, and then load the dataset:

```
VFSGroupDataset<FImage> dataset =
    new VFSGroupDataset<FImage>("zip:http://datasets.openimaj.org/att_faces.zip", ImageUtilities.FIMAGE_READER);
```

For the purposes of experimentation, we'll need to split the dataset into two halves; one for training our recogniser, and one for testing it. Just as in the Caltech 101 classification tutorial, this can be achieved with a `GroupedRandomSplitter`:

```
int nTraining = 5;
int nTesting = 5;
GroupedRandomSplitter<String, FImage> splits =
    new GroupedRandomSplitter<String, FImage>(dataset, nTraining, 0, nTesting);
GroupedDataset<String, ListDataset<FImage>, FImage> training = splits.getTrainingDataset();
GroupedDataset<String, ListDataset<FImage>, FImage> testing = splits.getTestDataset();
```

The first step in implementing an Eigenfaces recogniser is to use the training images to learn the PCA basis which we'll use to project the images into features we can use for recognition. The `EigenImages` class needs a list of images from which to learn the basis (i.e. all the training images from each person), and also needs to know how many dimensions we want our features to be (i.e. how many of the eigenvectors corresponding to the biggest eigenvalues to keep):

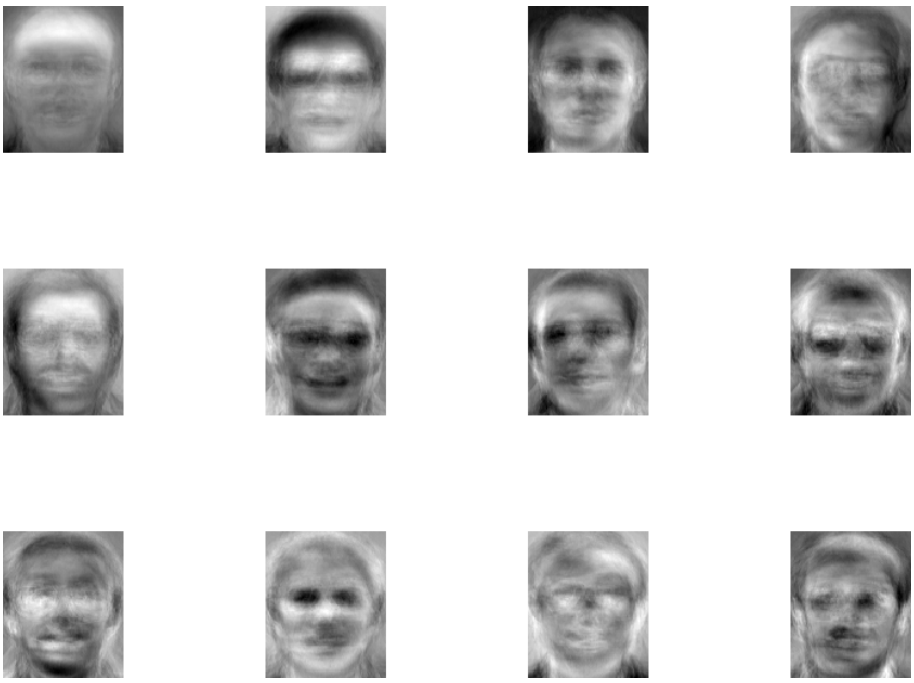
```
List<FImage> basisImages = DatasetAdaptors.asList(training);
int nEigenvectors = 100;
EigenImages eigen = new EigenImages(nEigenvectors);
eigen.train(basisImages);
```

[1] http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html

One way of thinking about how we use the basis is that any face image can literally be decomposed as weighted summation of the basis vectors, and thus each element of the feature we'll extract represents the weight of the corresponding basis vector. This of course implies that it should be possible to visualise the basis vectors as meaningful images. This is indeed the case, and the `EigenImages` class makes it easy to do. Let's draw the first 12 basis vectors (each of these basis images is often referred to as an EigenFace):

```
List<FImage> eigenFaces = new ArrayList<FImage>();
for (int i = 0; i < 12; i++) {
    eigenFaces.add(eigen.visualisePC(i));
}
DisplayUtilities.display("EigenFaces", eigenFaces);
```

At this point you can run your code. You should see an image very similar to the one below displayed:



Now we need to build a *database* of features from the training images. We'll use a `Map` of Strings (the person identifier) to an array of features (corresponding to all the features of all the training instances of the respective person):

```
Map<String, DoubleFV[]> features = new HashMap<String, DoubleFV[]>();
for (final String person : training.getGroups()) {
    final DoubleFV[] fvs = new DoubleFV[nTraining];

    for (int i = 0; i < nTraining; i++) {
        final FImage face = training.get(person).get(i);
        fvs[i] = eigen.extractFeature(face);
    }
    features.put(person, fvs);
}
```

Now we've got all the features stored, in order to estimate the identity of an unknown face image, all we need to do is extract the feature from this image, find the database feature with the smallest distance (i.e. Euclidean distance), and return the identifier of the corresponding person. Let's loop over all the testing images, and estimate which person they belong to. As we know the true identity of these people, we can compute the accuracy of the recognition:

```
double correct = 0, incorrect = 0;
for (String truePerson : testing.getGroups()) {
    for (FImage face : testing.get(truePerson)) {
        DoubleFV testFeature = eigen.extractFeature(face);

        String bestPerson = null;
        double minDistance = Double.MAX_VALUE;
        for (final String person : features.keySet()) {
            for (final DoubleFV fv : features.get(person)) {
                double distance = fv.compare(testFeature, DoubleFVComparison.EUCLIDEAN);

                if (distance < minDistance) {
                    minDistance = distance;
                    bestPerson = person;
                }
            }
        }

        System.out.println("Actual: " + truePerson + "\tguess: " + bestPerson);

        if (truePerson.equals(bestPerson))
            correct++;
        else
            incorrect++;
    }
}

System.out.println("Accuracy: " + (correct / (correct + incorrect)));
```

Now run the code again. You should see the actual person identifier and predicted identifier printed as each face is recognised. At the end, the overall performance will be printed and should be close to 93% (there will be some variability as the test and training data is split randomly each time the program is run).

13.1. Exercises

13.1.1. Exercise 1: Reconstructing faces

An interesting property of the features extracted by the Eigenfaces algorithm (specifically from the PCA process) is that it's possible to reconstruct an estimate of the original image from the feature. Try doing this by building a PCA basis as described above, and then extract the feature of a randomly selected face from the test-set. Use the `EigenImages#reconstruct()` to convert the feature back into an image and display it. You will need to normalise the image (`FImage#normalise()`) to ensure it displays correctly as the reconstruction might give pixel values bigger than 1 or smaller than 0.

13.1.2. Exercise 2: Explore the effect of training set size

The number of images used for training can have a big effect in the performance of your recogniser. Try reducing the number of training images (keep the number of testing images fixed at 5). What do you observe?

13.1.3. Exercise 3: Apply a threshold

In the original Eigenfaces paper, a variant of nearest-neighbour classification was used that incorporated a distance threshold. If the distance between the query face and closest database face was greater than a threshold, then an *unknown* result would be returned, rather than just returning the label of the closest person. Can you alter your code to include such a threshold? What is a good value for the threshold?

Part VIII. Advanced Techniques

OpenIMAJ contains a number of advanced features that are not directly related to multimedia processing, analysis and generation. This part of the tutorial looks at some of those features. In particular there are tutorials looking at how to make use of multiple processors using OpenIMAJ's `Parallel` and how to reference your code using OpenIMAJ's `Reference` annotations.

Chapter 14. Parallel Processing

Modern computers tend to have multiple processors. By making use of all the processing ability of your machine your programs can run much faster. Writing code that takes advantage of multiple processors in Java usually involves either manually creating and managing threads, or using a higher level concurrent programming abstraction library, such as the classes found in the excellent `java.util.concurrent` package.

A common use-case for multithreading in multimedia analysis is the application of an operation to a collection of objects - for example, the extraction of features from a list of images. This kind of task can be effectively parallelised using Java's concurrent classes, but requires a large amount of boiler-plate code to be written each time. To help reduce the programming overhead associated with this kind of parallel processing, OpenIMAJ includes a `Parallel` class that contains a number of methods that allow you to efficiently and effectively write multi-threaded loops.

To get started playing with the `Parallel` class, create a new OpenIMAJ project using the archetype, or add a new class and main method to an existing project. Firstly, lets see how we can write the parallel equivalent of a `for (int i=0; i<10; i++)` loop:

```
Parallel.forIndex(0, 10, 1, new Operation<Integer>() {
 public void perform(Integer i) {
     System.out.println(i);
 }
});
```

Try running this code; you'll see that all the numbers from 0 to 9 are printed, although not necessarily in the correct order. If you run the code again, you'll probably see the order change. It's important to note that the when parallelising a loop that the order of operations is not deterministic.

Now let's explore a more realistic scenario in which we might want to apply parallelisation. We'll build a program to compute the normalised average of the images in a dataset. Firstly, let's write the non-parallel version of the code. We'll start by loading a dataset of images; in this case we'll use the CalTech 101 dataset we used in the Classification 101 tutorial, but rather than loading record object, we'll load the images directly:

```
VFSGroupDataset<MBFImage> allImages = Caltech101.getImages(ImageUtilities.MBFIMAGE_READER);
```

We'll also restrict ourselves to using a subset of the first 8 groups (image categories) in the dataset:

```
GroupedDataset<String, ListDataset<MBFImage>, MBFImage> images = GroupSampler.sample(allImages, 8, false);
```

We now want to do the processing. For each group we want to build the average image. We do this by looping through the images in the group, resampling and normalising each image before drawing it in the centre of a white image, and then adding the result to an accumulator. At the end of the loop we divide the accumulated image by the number of samples used to create it. The code to perform these operations would look like this:

```
List<MBFImage> output = new ArrayList<MBFImage>();
ResizeProcessor resize = new ResizeProcessor(200);
for (ListDataset<MBFImage> clzImages : images.values()) {
    MBFImage current = new MBFImage(200, 200, ColourSpace.RGB);

    for (MBFImage i : clzImages) {
        MBFImage tmp = new MBFImage(200, 200, ColourSpace.RGB);
        tmp.fill(RGBColour.WHITE);

        MBFImage small = i.process(resize).normalise();
        int x = (200 - small.getWidth()) / 2;
        int y = (200 - small.getHeight()) / 2;
        tmp.drawImage(small, x, y);

        current.addInplace(tmp);
    }
    current.divideInplace((float) clzImages.size());
    output.add(current);
}
```

We can use the `DisplayUtilities` class to display the results:

```
DisplayUtilities.display("Images", output);
```

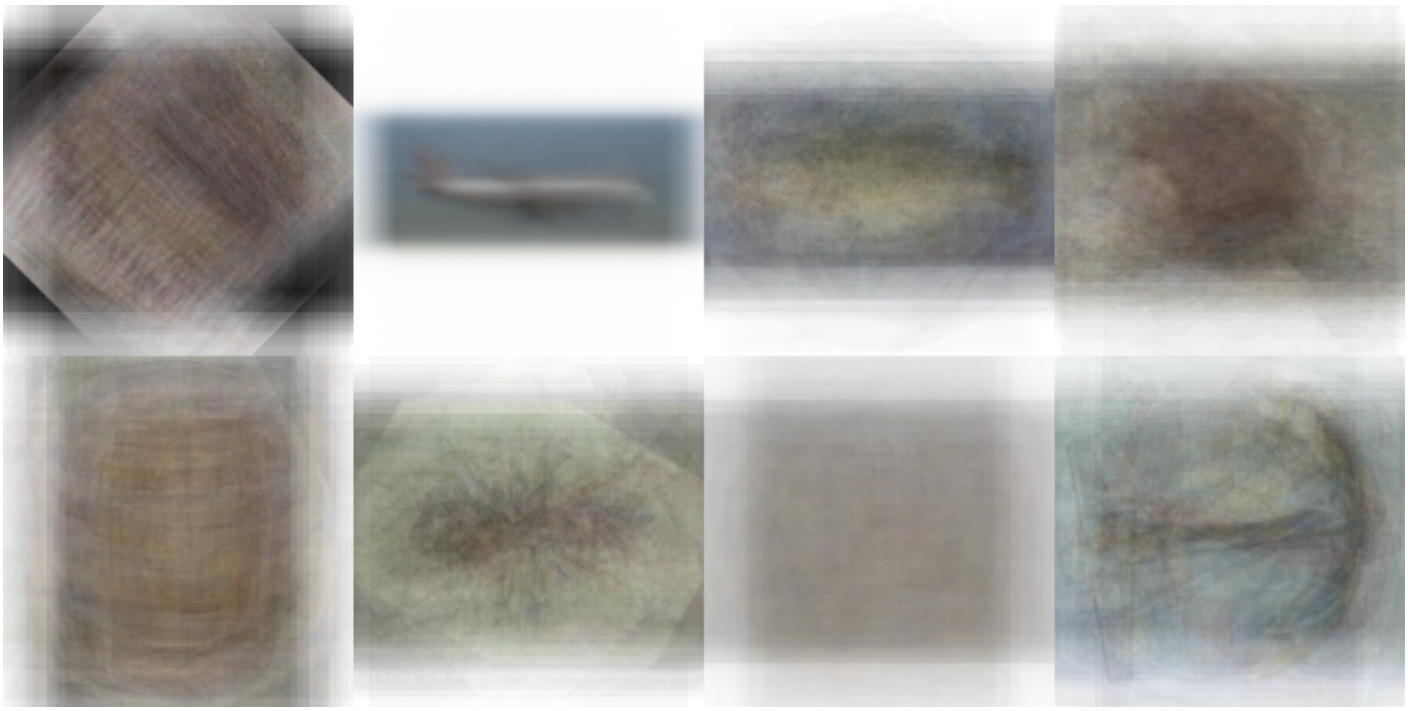
Before we try running the program, we'll also add some timing code to see how long it takes. Before the outer loop (the one over the groups provided by `images.values()`), add the following:

```
Timer t1 = Timer.timer();
```

and, after the outer loop (just before the display code) add the following:

```
System.out.println("Time: " + t1.duration() + "ms");
```

You can now run the code, and after a short while (on my laptop it takes about 7248 milliseconds (7.2 seconds)) the resultant averaged images will be displayed. An example is shown below. Can you tell what object is depicted by each average image? For many object types in the CalTech 101 dataset it is quite easy, and this is one of the reasons that the dataset has been criticised as being *too easy* for classification experiments in the literature.



Now we'll look at parallelising this code. We essentially have three options for parallelisation; we could parallelise the outer loop, parallelise the inner one, or parallelise both. There are many tradeoffs that need to be considered including the amount of memory usage and the task granularity in deciding how to best parallelise code. For the purposes of this tutorial, we'll work with the inner loop. Using the `Parallel.for` method, we can re-write the inner-loop as follows:

```
Parallel.forEach(clzImages, new Operation<MBFImage>() {
    public void perform(MBFImage i) {
        final MBFImage tmp = new MBFImage(200, 200, ColourSpace.RGB);
        tmp.fill(RGBColour.WHITE);

        final MBFImage small = i.process(resize).normalise();
        final int x = (200 - small.getWidth()) / 2;
        final int y = (200 - small.getHeight()) / 2;
        tmp.drawImage(small, x, y);

        synchronized (current) {
            current.addInplace(tmp);
        }
    }
});
```

For this to compile, you'll also need to make the `current` image `final` by adding the `final` keyword to the line in which it is created:

```
final MBFImage current = new MBFImage(200, 200, ColourSpace.RGB);
```

Notice that in the parallel version of the loop we have to put a `synchronized` block around the part where we accumulate into the `current` image. This is to stop multiple threads trying to alter the image concurrently. If we now run the code, we should hopefully see an improvement in the time it takes to compute the averages. You might also need to increase the amount of memory available to Java for the program to run successfully.

On my laptop, with 8 CPU cores the running time drops to ~3100 milliseconds. You might be thinking that because we have gone from 1 to 8 CPU cores that the speed-up would be greater; there are many reasons why that is not the case in practice, but the biggest is that the process that we're running is rather I/O bound because the underlying dataset classes we're using retrieve the images from disk each time they're needed. A second issue is that there are a couple of slight bottlenecks in our code; in particular notice that we're creating a temporary image for each image that we process, and that we also have to synchronise on the `current` accumulator image for each image. We can factor out these problems by modifying the code to use the *partitioned* variant of the for-each loop in the `Parallel` class. Instead of giving each thread a single image at a time, the partitioned variant will feed each thread a collection of images (provided as an `Iterator`) to process:

```
Parallel.forEachPartitioned(new RangePartitioner<MBFImage>(clzImages), new Operation<Iterator<MBFImage>>() {
 public void perform(Iterator<MBFImage> it) {
     MBFImage tmpAccum = new MBFImage(200, 200, 3);
     MBFImage tmp = new MBFImage(200, 200, ColourSpace.RGB);

     while (it.hasNext()) {
         final MBFImage i = it.next();
         tmp.fill(RGBColour.WHITE);

         final MBFImage small = i.process(resize).normalise();
         final int x = (200 - small.getWidth()) / 2;
         final int y = (200 - small.getHeight()) / 2;
         tmp.drawImage(small, x, y);
         tmpAccum.addInplace(tmp);
     }
     synchronized (current) {
         current.addInplace(tmpAccum);
     }
 }
});
```

The `RangePartitioner` in the above code will break the images in `clzImages` into as many (approximately equally sized) chunks as there are available CPU cores. This means that the `perform` method will be called many fewer times, but will do more work - what we've done is called increasing the task granularity. Notice that we created an extra working image called `tmpAccum` to hold the intermediary results. This means memory usage will be increased, however, also notice that whilst we still have to synchronise on the current image, we do far fewer times (once per CPU core in fact). Try running the improved version of the code; on my laptop it reduces the total running time even further to ~2900 milliseconds.

14.1. Exercises

14.1.1. Exercise 1: Parallelise the outer loop

As we discussed earlier in the tutorial, there were three primary ways in which we could have approached the parallelisation of the image-averaging program. Instead of parallelising the inner loop, can you modify the code to parallelise the outer loop instead? Does this make the code faster? What are the pros and cons of doing this?

Colophon

This book was produced using a workflow managed through Apache Maven using DocBook. XLST and XSL were used to produce the rendered versions. The cover was created using OmniGraffle.

The cover image is from the Dover Pictorial Archive. The cover font is Times New Roman. The text font is Linux Libertine; the heading font is Open Sans Condensed; and the code font is Inconsolata LGC.

The hand icon (for notes) is from `http://clipartist.info/`. The bulb icon (for tips) is by YassineMrabet from Wikimedia Commons.