TRAINABLE WEKA SEGMENTATION
USER MANUAL

Ignacio Arganda-Carreras, Verena Kaynig, Curtis Rueden, Kevin W. Eliceiri,
Johannes Schindelin, Albert Cardona, H. Sebastian Seung

January 30, 2017
Contents

1 Introduction 4

2 The Graphical User Interface 7
  2.1 Training panel .......................... 8
    2.1.1 Train classifier .......................... 8
    2.1.2 Toggle overlay .......................... 8
    2.1.3 Create result .......................... 8
    2.1.4 Get probability .......................... 8
    2.1.5 Plot result .......................... 9
  2.2 Options panel .......................... 9
    2.2.1 Apply classifier .......................... 9
    2.2.2 Load classifier .......................... 10
    2.2.3 Save classifier .......................... 10
    2.2.4 Load data .......................... 10
    2.2.5 Save data .......................... 11
    2.2.6 Create new class .......................... 11
    2.2.7 Settings dialog .......................... 11
      2.2.7.1 Training features (2D) .............. 11
      2.2.7.2 Training features (3D) .............. 15
      2.2.7.3 Feature options ...................... 16
      2.2.7.4 Classifier options ...................... 16
      2.2.7.5 Class names ...................... 17
      2.2.7.6 Save feature stack ...................... 17
      2.2.7.7 Result overlay opacity ...................... 17
    2.2.8 WEKA .......................... 18
  2.3 Macro language compatibility .......................... 18
    2.3.1 Complete macro example .......................... 21

3 Image features 23
  3.1 Edge detection features .......................... 23
  3.2 Texture description features .......................... 24
  3.3 Noise removal features .......................... 25
  3.4 Membrane detection features .......................... 26
  3.5 Feature Selection .......................... 26
1 Introduction

State-of-the-art light and electron microscopes are capable of acquiring large image datasets, but quantitatively evaluating the data often involves manually annotating structures of interest. For example, to measure the average size of mitochondria in an electron microscopy image stack, each mitochondrion has to be outlined by a human annotator. This process is time-consuming and is becoming the main bottleneck in the evaluation pipeline. To overcome this problem, we have introduced the Trainable Weka Segmentation (TWS), a machine learning tool that leverages a limited number of manual annotations in order to train a classifier and segment the remaining data automatically. The tool works interactively, allowing the user to guide the training by providing corrections to the classifier output. In addition, TWS can provide unsupervised segmentation learning schemes (clustering) for image data and can be customized to employ user-designed feature maps or classifiers.

The usefulness of the TWS tool has already been demonstrated by its utilization in a wide range of imaging pipelines that involve disparate segmentation tasks: analyzing wing photomicrographs [Dobens and Dobens, 2013], visualizing myocardial blood flow [Krueger et al., 2013], monitoring nests of bees [Hart and Huang, 2012], cell tracking in the Mother Machine [Jug et al., 2014], and other applications. TWS has proven useful for performing segmentation using many different image modalities. These include magnetic resonance imaging [Kulinowski et al., 2011], two-photon microscopy [Villa et al., 2013], X-ray microtomography [Madra et al., 2014], serial-section transmission electron microscopy [Laptev et al., 2012], confocal fluorescence microscopy [Felcht et al., 2012, Frank et al., 2012, Crepaldi et al., 2013], micro- and computerized tomography [Maiora and Graña, 2012, Macdonald and Shefelbine, 2013], transmission scanning [Mathew et al., 2012], and angiography [Favazza et al., 2013].

Traditional image processing methods

Image segmentation is generally defined as the process of partitioning a digital image into non-intersecting regions. These regions or segments comprise sets of pixels that share certain visual characteristics and are assigned a specific label. For instance, in the microscopic image of a cell, one could segment the different organelles and label pixels belonging to the nucleus, the mitochondria, and other structures. Similarly, in an image from a security camera, one might want to identify suspicious objects and separate them from the rest of the pixels. In the same example, a face recognition system may attempt to label the person or persons appearing in the image. Therefore, image segmentation can be regarded as an ill-defined problem since, depending on the final application, different ways of partitioning the same image can be considered correct. Hundreds of automatic and semiautomatic segmentation

---

1 A search of “Trainable Weka Segmentation” in Google Scholar produces more than 100 hits.
algorithms have been presented since the appearance of the digital image. However, no single method can be considered appropriate for all types of images. Moreover, methods that have been designed for a particular type of image might not be applicable to other types.

Most traditional methods are based only on the intensity information of pixels. Nonetheless, humans use much more knowledge when performing manual segmentation. For that reason, in recent years, trainable methods have emerged as powerful tools to include part of that knowledge in the segmentation process and improve the accuracy of the labeled regions. Algorithms to perform this task have been developed principally for natural and medical images but can be adapted for other types of image data and transferred to platforms that are accessible to both experienced and inexperienced users. Such software should provide a user-friendly and intuitive framework for prototyping and applying machine learning algorithms to image data and visualizing their results.

Platforms that build in machine learning tools

By the time we started the TWS project, just a few software platforms partially provided both machine learning and image processing tools. These included commercial platforms (e.g., MATLAB, MathWorks, Natick, MA) and open-source platforms, such as the data mining platforms Konstanz Information Miner (KNIME) by Berthold et al. [2007] and RapidMiner\(^2\), or the biologists-oriented software CellProfiler by Kamentsky et al. [2011]. Commercial platforms usually target inexperienced users and a wide range of image types. However, the details of the algorithms are hidden, which is undesirable for use in scientific research. Conversely, those details are available in open-source platforms such as KNIME and RapidMiner, which is becoming the world-leading open-source system for data and text mining. Nevertheless, RapidMiner is developed primarily by the machine learning community for the machine learning community, and its image processing extension by Burget et al. [2010] provides only a minimal set of image tools. This makes the platform less attractive for computer scientists to use it to develop image segmentation solutions. Other projects like the Vision with Generic Algorithms (VIGRA) by Köthe [1999], offer powerful computer vision libraries with a focus on algorithms and data structures but no visualization tools or user-friendly interfaces. And only a development version of CellProfiler integrates VIGRA learning methods into custom segmentation pipelines. Nowadays, we can find more libraries (besides VIGRA) with applications in the field of bioimage informatics. That is the case of the computer vision library Mahotas [Coelho, 2012] and scikit-image / sklearn python libraries [Van der Walt et al., 2014, Pedregosa et al., 2011]. Regarding tools, we need to distinguish between those dedicated to object detection, such as the Cell Profiler Analyst [Jones et al., 2008], and those dedicated to machine learning based segmentation of biological images. In the later group, we now find ilastik [Sommer et al., 2011], which contains a powerful interface to supply user feedback, although is limited to a small set of classifiers; the Vaa3D plugin for interactive cell segmentation [Li et al., 2015], which uses region features; and, more recently, the data mining module of the web-based Cytomine software [Marée et al., 2016].

\(^2\)http://rapid-i.com
To address these deficiencies in the field, we started the new open-source software project TWS. The project combines the image processing toolkit Fiji (Fiji Is Just ImageJ) by Schindelin et al. [2012], a popular distribution of ImageJ by Rasband [1997-2009], with the state-of-the-art machine learning algorithms provided in the latest version of the data mining and machine learning toolkit Waikato Environment for Knowledge Analysis (WEKA) by Hall et al. [2009]. TWS provides a set of library methods for extracting statistical properties of an image from user-provided pixel samples and uses that information to segment the rest of the pixels in that image or a similar image. These methods are then implemented in a modular and transparent way and can be called up from any Fiji plugin, script, or macro. TWS also provides a friendly graphical user interface (GUI) for loading a two-dimensional (2D) or three-dimensional (3D) image and performing automatic segmentation by interactive learning.

Enhancing Fiji and WEKA

In the past few years, Fiji has become the software of reference for many scientists to meet their image analysis needs, especially in the field of biomedical imaging. Fiji provides its users with powerful tools to generate sophisticated image processing pipelines and algorithms, via scripting languages and library methods that can handle many types and sizes of images. At the same time, WEKA is nowadays recognized as a landmark system in data mining and machine learning. It has achieved widespread acceptance within academia and business circles, and has become a widely used tool for data mining research.

However, little (if any) of the success of both toolboxes would have been possible if they had not been released as open-source software. Giving users free access to the source code has enabled a thriving community to develop and facilitated the creation of many projects that incorporate or extend WEKA’s existing functionalities. One of the best examples of these projects is TWS, which combines both toolboxes to enlarge their capabilities and increase their impact and range of application. For WEKA users and developers, TWS offers transparent access to a whole new set of supervised and unsupervised learning problems based on an arbitrarily large number of image features. For Fiji users and developers, respectively, TWS provides a new and user-friendly way of performing image segmentation and facilitates access to learning tools that can be used to either enhance existing image processing algorithms and pipelines or create new ones.
2 The Graphical User Interface

TWS runs on any 2D or 3D image (grayscale or color). To use 2D features, you need to select the menu command “Plugins ⊳ Segmentation ⊳ Trainable Weka Segmentation”. For 3D features, call the plugin under “Plugins ⊳ Segmentation ⊳ Trainable Weka Segmentation 3D”. Both commands will use the same GUI but offer different feature options in their settings.

![Trainable Weka Segmentation](image)

**Figure 2.1:** Example of the first look of the plugin window when using it on a Transmission Electron Microscopy (TEM) 2D image.

By default, the plugin starts with two classes, i.e. it will produce binary pixel classification. The user can add traces to both classes using the whole set of tools for region of interest (ROI) drawing available in Fiji. That includes rectangular, round rectangular, oval, elliptical, brush polygon and freehand selections. By default, the freehand selection tool (of
1 pixel width) is automatically selected.

The user can pan, zoom in and out, or scroll between slices (if the input image is a stack) in the main canvas as if it were any other Fiji window. On the left side of the canvas there are two panels of buttons, one for the training and one for the general options. On the right side of the image canvas we have a panel with the list of traces for each class and a button to add the current ROI to that specific class. All buttons contain a short explanation of their functionality that is displayed when the cursor lingers over the buttons.

2.1 Training panel

2.1.1 Train classifier

This button activates the training process. One trace of two classes is the minimum required to start training. The first time this button is pressed, the features of the input image will be extracted and converted to a set of vectors of float values, which is the format the Weka classifiers are expecting. This step can take some time depending on the size of the images, the number of features and the number of cores of the machine where Fiji is running. The feature calculation is done in a completely multi-thread fashion. The features will be only calculated the first time we train after starting the plugin or after changing any of the feature options.

If the training ends correctly, then the displayed image will be completely segmented and the result will be overlaid with the corresponding class colors (see Fig. 2.2). Notice that all buttons are now enabled, since all their functionalities are possible after training.

While training, this button will show the label "STOP". By clicking on it, the whole training process will be interrupted and the plugin reset to the state previous to the training.

2.1.2 Toggle overlay

This button activates and deactivates the overlay of the result image. The transparency of the overlay image can be adjusted in the Settings dialog.

2.1.3 Create result

It creates and displays the resulting image. This image is equivalent to the current overlay (8-bit Color with same class colors). Each pixel is set to the index value of the most likely class (0, 1, 2...).

2.1.4 Get probability

Based on the current trained classifier, the probability that each pixel belongs to each class is displayed on a 32-bit hyperstack (see Fig. 2.3).
Figure 2.2: Example of the aspect of the plugin window after training on the TEM image of Fig. 2.1.

2.1.5 Plot result

This button calls the Weka core to generate the model performance chart based on the current classifier and training data. It displays the receiver operating characteristic (ROC), precision/recall, etc. curves (see Fig. 2.4). These curves allow to visualize the performance of the classifier based on different thresholds that can be applied to the probability maps.

2.2 Options panel

2.2.1 Apply classifier

By clicking on this button we can apply the current classifier to any image or stack of images we have in our file system. Two dialogs will pop up to, first, ask the user for the input image or stack and, second, ask if the result should be displayed as a probability map or a segmentation (label image with final classes). Then the plugin will perform the image
segmentation based on the current classifier and ---consequently--- selected features. This may take a while depending on the number and size of the input images and the number of cores of the machine. After finishing, the input image (or stack) and its corresponding segmentation will be displayed.

### 2.2.2 Load classifier

Here we can load any previously saved classifier. The plugin will check and adjust the selected features with the attributes of this new classifier. The classifier file format is the one used in Weka (.model).

### 2.2.3 Save classifier

It saves the current classifier into a file, under the standard Weka format (.model). This allows us to store classifiers and apply them later on different sessions.

### 2.2.4 Load data

Here we can load the data (in Weka format) from previous traces on the same or other image or stack. Again, the plugin will check and force the consistency between the loaded data and the current image, features and classes. The input file format is the standard Weka format: ARFF.
2.2 Options panel

Figure 2.4: Weka model performance chart. Displayed after clicking on "Plot result".

2.2.5 Save data

With this button we can save the current trace information into a data file that we can handle later with the plugin or the Weka Explorer itself. The plugin will save the feature vectors derived from the pixels belonging to each trace into an ARFF file at a location chosen by the user. Notice the traces (regions of interests selected by the user) are not saved but only their corresponding feature vectors. To save the ROIs, you can simply use the ROI Manager.

2.2.6 Create new class

The default number of classes of the plugin is two, but through this button we can increase up to an arbitrary number. The name of the new classes can be changed on the Settings dialog.

2.2.7 Settings dialog

The rest of tunable parameters of the plugin can be changed on the Settings dialog, which is displayed when clicking on this button. The dialog is slightly different for the 2D and 3D plugins since the training features are not exactly the same (see Fig. 2.5 and 2.6). Next we describe the different parts of the dialog.

2.2.7.1 Training features (2D)

Here we can select and deselect the training features, which are the key of the learning procedure. In the background, the plugin creates a stack of images representing the features—one image for each feature. For instance, if only Gaussian blur is selected as a feature, the classifier will be trained on the original image and some blurred versions to it with different $\sigma$ parameters for the Gaussian. $\sigma$ is commonly equal to $\sigma_{\text{min}}$, $2\sigma_{\text{min}}$, $4\sigma_{\text{min}}$, ..., $2^{n-1}\sigma_{\text{min}}$, where $2^{n-1}\sigma_{\text{min}} \leq \sigma_{\text{max}}$. By default $\sigma_{\text{min}} = 1$, $\sigma_{\text{max}} = 16$ and therefore $n = 5$. 
If the input image is grayscale, the features will be calculated using double precision (32-bit images). In the case of RGB input images, the features will be RGB as well.

The different available 2D image features are:

- **Gaussian blur**: performs \( n \) individual convolutions with Gaussian kernels with the normal \( n \) variations of \( \sigma \). The larger the radius the more blurred the image becomes until the pixels are homogeneous.

- **Sobel filter**: calculates an approximation of the gradient of the image intensity at each pixel [Sobel, 1990]. Gaussian blurs with \( \sigma \) varying as usual are performed prior to the filter.

- **Hessian**: Calculates a Hessian matrix \( H \) at each pixel. Prior to the application of any filters, a Gaussian blur with varying \( \sigma \) is performed. The final features used for pixel classification, given the Hessian matrix \[
\begin{pmatrix}
a & b \\
c & d
\end{pmatrix}
\] are calculated thus:

  - Module: \( \sqrt{a^2 + bc + d^2} \).
  - Trace: \( a + d \).
  - Determinant: \( ad - cb \).
2.2 Options panel

- First eigenvalue: \( \frac{a+d}{2} + \sqrt{\frac{4b^2 + (a-d)^2}{2}} \).
- Second eigenvalue: \( \frac{a+d}{2} - \sqrt{\frac{4b^2 + (a-d)^2}{2}} \).
- Orientation: \( \frac{1}{2} \arccos \left( \frac{4b^2 + (a-d)^2}{2} \right) \). This operation returns the orientation for which the second derivative is maximal. It is an angle returned in radians in the range \( \left[ -\frac{\pi}{2}, \frac{\pi}{2} \right] \) and corresponds to an orientation without direction. The orientation for the minimal second derivative can be obtained by adding (or subtracting) \( \frac{\pi}{2} \).
- Gamma-normalized square eigenvalue difference: \( t^4 (a-d)^2 \left( (a-d)^2 + 4b^2 \right) \), where \( t = 1^{3/4} \).
- Square of Gamma-normalized eigenvalue difference: \( t^2 ((a-d)^2 + 4b^2) \), where \( t = 1^{3/4} \).

- **Difference of Gaussians**: calculates two Gaussian blur images from the original image and subtracts one from the other. \( \sigma \) values are varied as usual, so \( \frac{n(n-1)}{2} \) feature images are added to the feature stack.

- **Membrane projections**: enhances membrane-like structures of the image through directional filtering. The initial kernel for this operation is hardcoded as a \( 19 \times 19 \) zero matrix with the middle column entries set to 1. Multiple kernels are created by rotating the original kernel by 6 degrees up to a total rotation of 180 degrees, giving 30 kernels. Each kernel is convolved with the image and then the set of 30 images are Z-projected into a single image via 6 methods:
  - sum of the pixels in each image
  - mean of the pixels in each image
  - standard deviation of the pixels in each image
  - median of the pixels in each image
  - maximum of the pixels in each image
  - minimum of the pixels in each image

  Each of the 6 resulting images is a feature. Hence pixels in lines of similarly valued pixels in the image that are different from the average image intensity will stand out in the Z-projections.

- **Mean, Variance, Median, Minimum, Maximum**: the pixels within a radius of \( \sigma \) pixels from the target pixel are subjected to the pertinent operation (mean/min etc.) and the target pixel is set to that value.

- **Anisotropic diffusion**: the anisotropic diffusion filtering [Tschumperle and Deriche, 2005] from Fiji with 20 iterations, \( \sigma \) smoothing per iterations, \( a_1 = 0.10, 0.35, a_2 = 0.9 \), and an edge threshold set to the membrane size.
• **Bilateral filter**: is very similar to the Mean filter but better preserves edges while averaging/blurring other parts of the image [Tomasi and Manduchi, 1998]. The filter accomplishes this task by only averaging the values around the current pixel that are close in color value to the current pixel. The “closeness” of other neighborhood pixels to the current pixels is determined by the specified threshold. I.e. for a value of 10 each pixel that contributes to the current mean have to be within 10 values of the current pixel. In our case, we combine spatial radius of 5 and 10, with a range radius of 50 and 100.

• **Lipschitz filter**: from Mikulas Stencel plugin [Štencel and Janáček, 2006]. This plugin implements Lipschitz cover of an image that is equivalent to a grayscale opening by a cone. The Lipschitz cover can be applied for the elimination of a slowly varying image background by subtraction of the lower Lipschitz cover (a top-hat procedure). A sequential double scan algorithm is used. We use down and top hats combinations, with slope $s = 5, 10, 15, 20, 25$.

• **Kuwahara filter**: another noise-reduction filter that preserves edges. This is a version of the Kuwahara filter [Kuwahara et al., 1976] that uses linear kernels rather than square ones. We use the membrane patch size as kernel size, 30 angles and the three different criteria (Variance, Variance / Mean and Variance / Mean $^2$).

• **Gabor filter**: at the moment this option may take some time and memory because it generates a very diverse range of Gabor filters (22 in total). This may undergo changes in the future. The implementation details are included in this script. The Gabor filter is an edge detection and texture filter [Fogel and Sagi, 1989], which convolves several kernels at different angles with an image. Each point in a kernel is calculated as

$$\cos(2\pi f x + \psi)e^{-0.5 \left( \frac{x^2}{\sigma_x^2} + \frac{y^2}{\sigma_y^2} \right)}.$$  

Gabor filters are band-pass filters and therefore implement a frequency transformation.

• **Derivatives filter**: calculates high order derivatives of the original 2D input image ($d^4/dx^2dy^2$, $d^6/dx^3dy^3$, $d^8/dx^4dy^4$, $d^{10}/dx^5dy^5$) using FeatureJ (it requires enabling the ImageScience update site in the updater).

• **Laplacian filter**: computes the Laplacian of the input image using FeatureJ (it requires enabling the ImageScience update site in the updater). It uses smoothing scale $\sigma$.

• **Structure filter**: calculates for all elements in the input image, the eigenvalues (smallest and largest) of the so-called structure tensor [Rao and Schunck, 1991, Weickert, 1999] using FeatureJ (it requires enabling the ImageScience update site in the updater). It uses smoothing scale $\sigma$ and integration scales 1 and 3.

• **Entropy**: draws a circle of radius $r$ around each pixel; gets the histogram of that circle split in $numBins$ chunks; then calculates the entropy as $\sum_{p \text{ in histogram}} -p \cdot \log_2(p)$, where $p$ is the probability of each chunk in the histogram. $numBins$ is equal to 32, 64, 128, 256. $r$ is equal to $\sigma$. 

---

Trainable Weka Segmentation user manual  
page 14 / 44
2.2 Options panel

- **Neighbors**: shifts the image in 8 directions by a certain number of pixels, $\sigma$. Therefore it creates $8n$ feature images.

When using grayscale images, the input image will be also included as a feature. In the case of color (RGB) images, the **Hue, Saturation and Brightness** values will be as well part of the features.

### 2.2.7.2 Training features (3D)

![Figure 2.6: Settings dialog for the Trainable Weka Segmentation 3D plugin.](image)

When calling the plugin from the menu command Plugins $\Rightarrow$ Segmentation $\Rightarrow$ Trainable Weka Segmentation 3D the set of available image features will be as follows:

- **Gaussian blur**: performs $n$ individual 3D convolutions with Gaussian kernels with the normal $n$ variations of $\sigma$. As in the 2D case, the larger the radius the more blurred the image becomes until the pixels are homogeneous.

- **Hessian**: using FeatureJ it computes for each image element (voxel) the eigenvalues of the Hessian, which can be used for example to discriminate locally between plate-like, line-like, and blob-like image structures. More specifically, it calculates the magnitude of the largest, middle and smallest eigenvalue of the Hessian tensor. It requires enabling the ImageScience update site in the updater. It uses smoothing scale $\sigma$.

- **Derivatives**: calculates high order derivatives of the input image (now in one more dimension: $\frac{d^4}{dx^4dy^2dz^2}$, $\frac{d^6}{dx^3dy^3dz^3}$, $\frac{d^8}{dx^5dy^3dz^2}$, $\frac{d^{10}}{dx^5dy^5dz^5}$) using FeatureJ (it requires enabling the ImageScience update site in the updater).
2.2 Options panel

- **Laplacian**: as in the 2D case, it computes the Laplacian of the input image using FeatureJ (it requires enabling the ImageScience update site in the updater). It uses smoothing scale $\sigma$.

- **Structure**: same as the 2D case but with an extra dimension. It calculates the eigenvalues (smallest and largest) of the so-called structure tensor for all elements in the input image using FeatureJ (so it requires enabling the ImageScience update site in the updater). It uses smoothing scale $\sigma$ and integration scales 1 and 3.

- **Edges**: detects edges using the Canny edge detection algorithm [Canny, 1986], which involves computation of the gradient magnitude, suppression of locally non-maximum gradient magnitudes, and (hysteresis) thresholding. Again, this feature uses FeatureJ so it requires enabling the ImageScience update site in the updater. It uses smoothing scale $\sigma$.

- **Difference of Gaussian**: similar to the 2D feature, it calculates two Gaussian blur images from the original image and subtracts one from the other. $\sigma$ values are varied as usual, so $\frac{n(n-1)}{2}$ feature images are added to the stack.

- **Minimum, Maximum, Mean, Variance, Median**: the voxels within a radius of $\sigma$ voxels from the target pixel are subjected to the pertinent operation (mean/min etc.) and the target voxel is set to that value.

2.2.7.3 Feature options

- **Membrane thickness**: expected value of the membrane thickness, 1 pixel by default. The more accurate, the more precise the filter will be. Only available for 2D features.

- **Membrane patch size**: this represents the size $n \times n$ of the field of view for the membrane projection filters. Only available for 2D features.

- **Minimum sigma**: minimum radius of the isotropic filters used to create the features. By default it is 1 pixel.

- **Maximum sigma**: maximum radius of the isotropic filters used to create the features. By default it is set to 16 pixels in 2D and 8 voxels in 3D.

2.2.7.4 Classifier options

The default classifier is *FastRandomForest*, a multi-threaded version of random forest [Breiman, 2001] by Fran Supek, initialized with 200 trees and 2 random features per node. However the user can select any available classifier in the Weka by clicking on "Choose" button. By left-clicking on the classifier text we can also edit the classifier options (see Fig. 2.7).

If you do not find the classifier you want, you might have to install the Weka package that includes it. For that, you need to launch the Weka GUI Chooser (by clicking on the Weka button of the left panel of the plugin GUI, see Sec. 2.2.8) and use the Weka Package Manager (under Tools $\triangleright$ Package manager).
2.2 Options panel

Figure 2.7: Classifier selection in the TWS Settings dialog.

2.2.7.5 Class names

The classes can be renamed using these text boxes. The GUI will be updated as soon as you click on “OK”.

2.2.7.6 Save feature stack

We can save the features into a file as a stack of images by clicking on this button. It will use the last feature configuration that is available.

2.2.7.7 Result overlay opacity

This slider sets the opacity of the resulting overlay image. Depending on the image contrast of our input images, we might be interested on adjusting this value to ease visualization.
2.2.8 WEKA

The Weka button launches the Weka GUI Chooser, where we can start all the applications available in Weka:

- **Explorer**: an environment for exploring data with Weka.
- **Experimenter**: an environment for performing experiments and conducting statistical tests between learning schemes.
- **KnowledgeFlow**: this environment supports essentially the same functions as the Explorer but with a drag-and-drop interface. One advantage is that it supports incremental learning.
- **SimpleCLI**: provides a simple command-line interface that allows direct execution of Weka commands for operating systems that do not provide their own command line interface.

2.3 Macro language compatibility

TWS is completely compatible with the popular ImageJ macro language. Each of the buttons in the GUI are macro-recordable and their commands can be reproduced later from a simple macro file.

The complete list of commands is as follows:

- **Start the plugin**:

  ```java
  run("Trainable Weka Segmentation");
  ```

- **Add traces (the current ROI) to a specific class**:
  Format: `addTrace( class index, slice number )`

  For example, to add the selected ROI of the first slice to the first class, we type:

  ```java
  call("trainableSegmentation.Weka_Segmentation.addTrace", "0", "1");
  ```

- **Train the current classifier**:

  ```java
  call("trainableSegmentation.Weka_Segmentation.trainClassifier");
  ```

- **Toggle the result overlay**:

  ```java
  call("trainableSegmentation.Weka_Segmentation.toggleOverlay");
  ```
2.3 Macro language compatibility

- Get the result label image:
  
  ```
  call("trainableSegmentation.Weka_Segmentation.getResult");
  ```

- Get the probability maps:
  
  ```
  call("trainableSegmentation.Weka_Segmentation.getProbability");
  ```

- Plot the model performance chart:
  
  ```
  call("trainableSegmentation.Weka_Segmentation.plotResultGraphs");
  ```

- Apply the current classifier to an image or stack:
  Format: `applyClassifier(input directory, input image or stack, show results flag, store results flag, probability maps flag, store folder)`

  Example:
  
  ```
  call("trainableSegmentation.Weka_Segmentation.applyClassifier","/home/iarganda/data/input−image.tif","showResults=true","storeResults=false","probabilityMaps=false","");
  ```

- Load a classifier from file:
  
  ```
  call("trainableSegmentation.Weka_Segmentation.loadClassifier","/home/iarganda/classifier.model");
  ```

- Save the current classifier into a file:
  
  ```
  call("trainableSegmentation.Weka_Segmentation.saveClassifier", "/home/iarganda/classifier.model");
  ```

- Load previously saved trace data from an ARFF file:
  
  ```
  call("trainableSegmentation.Weka_Segmentation.loadData","/home/iarganda/data.arff");
  ```

- Save current trace data (feature vectors of traces and classes) into a file:
  
  ```
  call("trainableSegmentation.Weka_Segmentation.saveData","/home/iarganda/data.arff");
  ```

- Create a new class:
2.3 Macro language compatibility

- Launch Weka:

```plaintext
call("trainableSegmentation.Weka_Segmentation.launchWeka");
```

- Change a class name:
  Format: `changeClassName( class index, class new name )`

  Example (change first class name to "background"):

```plaintext
call("trainableSegmentation.Weka_Segmentation.changeClassName", "0", "background");
```

- Set option to balance the class distributions:

```plaintext
call("trainableSegmentation.Weka_Segmentation.setClassBalance", "true");
```

- Set membrane thickness in pixels (only 2D):

```plaintext
call("trainableSegmentation.Weka_Segmentation.setMembraneThickness", "2");
```

- Set the membrane patch size in pixels, (only 2D: $N \times N$):

```plaintext
call("trainableSegmentation.Weka_Segmentation.setMembranePatchSize", "16");
```

- Set the minimum kernel radius (in pixels/voxels):

```plaintext
call("trainableSegmentation.Weka_Segmentation.setMinimumSigma", "2.0");
```

- Set the maximum kernel radius (in pixels/voxels):

```plaintext
call("trainableSegmentation.Weka_Segmentation.setMaximumSigma", "8.0");
```

- Set a new classifier:
  Format: `setClassifier( classifier class, classifier options )`

  Example (change classifier to NaiveBayes):

```plaintext
```
2.3 Macro language compatibility

- Set the result overlay opacity:

```java
1 call("trainableSegmentation.Weka_Segmentation.setOpacity", "50");
```

2.3.1 Complete macro example

The following code is a complete macro that loads a sample image (the default Leaf image available online), calls the TWS plugin and performs pixel binary classification based on a few image features and two sets of training pixels selected with the rectangular selection tool:

```java
// Open Leaf sample
run("Leaf (36K)");

// Start plugin
run("Trainable Weka Segmentation");

// Wait for the plugin to load
wait(3000);

// Select plugin window
selectWindow("Trainable Weka Segmentation v3.2.4");

// Add one region of interest to each class
makeRectangle(367, 0, 26, 94);
call("trainableSegmentation.Weka_Segmentation.addTrace", "0", "1");
makeRectangle(186, 132, 23, 166);
call("trainableSegmentation.Weka_Segmentation.addTrace", "1", "1");

// Enable some extra features
// Variance=true
// Mean=true
// Minimum=true
// Maximum=true
// Median=true

call("trainableSegmentation.Weka_Segmentation.setFeature", "Variance=true");
call("trainableSegmentation.Weka_Segmentation.setFeature", "Mean=true");
call("trainableSegmentation.Weka_Segmentation.setFeature", "Minimum=true");
call("trainableSegmentation.Weka_Segmentation.setFeature", "Maximum=true");
call("trainableSegmentation.Weka_Segmentation.setFeature", "Median=true");

// Change class names
// 0 = background
// 1 = leaf

call("trainableSegmentation.Weka_Segmentation.changeClassName", "0", "background");
call("trainableSegmentation.Weka_Segmentation.changeClassName", "1", "leaf");

// Balance class distributions
// true

call("trainableSegmentation.Weka_Segmentation.setClassBalance", "true");

// Train current classifier

call("trainableSegmentation.Weka_Segmentation.trainClassifier");

call("trainableSegmentation.Weka_Segmentation.getProbability");
```

Notice that in line 6 we wait during 3 seconds for the plugin to be loaded. This prevents the macro from calling the following instruction before the plugin window has been shown. Also, in line 7 we select the plugin window by its title, which has the version number ("v3.2.3") hard-coded on it (change it if you are using a different version).

After calling the macro, the plugin window should look like this:
2.3 Macro language compatibility

The macro also returns the probability maps of the “leaf” and “background” classes on a separate image:
3 Image features

Image features are the basis of the learning procedure in TWS. They allow us to represent images in a higher dimensional space that aims at facilitating the pixel classification. In other words, the image features help us to better describe the images by enhancing or collecting different aspects of the images. Based on these aspects, we can make a rough categorization of most of the standard TWS features as edge detectors, texture descriptors, noise reducers or membrane detectors. The importance of each feature in the learning process may vary depending on each specific learning task. Next, we show examples of each feature category to help users select the best features for their particular pixel classification problems. Notice that some features can be included in several categories.

3.1 Edge detection features

The edge detection features include all features that aim at identifying the borders of the objects or elements of the image. Therefore, in this category we include the following features: Sobel filter, Canny (included in Edges), Hessian, Difference of Gaussians, Gabor (some of them), Derivatives, and Laplacian.

Fig. 3.1 shows some of these features applied using a sigma value of 1.0.

Figure 3.1: Edge detection features with sigma 1.0.
3.2 Texture description features

The same features with a larger sigma will detect borders at a different scale, as can be seen in Fig. 3.2 for sigma 4.0.

![Figure 3.2: Edge detection features with sigma 4.0.](image)

Some of these features help as well to detect bright and dark “blobs”, that is, regions of the image that differ in brightness compared to the surrounding regions. This can be appreciated for instance in Fig. 3.1 and 3.2 Difference of Gaussians images.

In general, the edge detection features are useful when segmenting objects whose boundaries can be defined by a change of contrast in the intensity values of the image. These features will help the classifier decide what to do with pixels/voxels which are in the limit of distinct regions. Notice the whole TWS pipeline can be used for border detection, combining many different features for the purpose of predicting borders.

3.2 Texture description features

The texture description features include all features that help differentiating between the textures contained in the image. Out of the features available by default in TWS, the following can be considered texture descriptors: Variance, Mean, Maximum, Minimum, Median, Gabor (some of them), Structure, Entropy and Neighbors.

In this case, the sigma value will play an essential role by determining the radius of the region from which the feature will extract local texture statistics. Fig. 3.3 shows an example of these features using the same cell image as in the previous section.

These features help the classifier distinguish between areas of the image that do not have clear boundaries (large intensity variations) but contain distinct patterns that are homogeneous along the whole area. Many times, these patterns are ignored by the other image
3.3 Noise removal features

![Images of noise removal features: (a) Original image. (b) Median filter (sigma 4.0). (c) Entropy (sigma 1.0 and 256 bins). (d) Maximum filter (sigma 2.0). (e) Median filter (sigma 8.0). (f) Entropy (sigma 8.0 and 256 bins).]

Figure 3.3: Texture description features at different scales.

features, which are unable to detect them or interpret them as noise or background and remove them.

3.3 Noise removal features

The noise removal features are those that aim at "cleaning" the image from artifacts and background noise so only the relevant information of the image prevails. In this category we could include then the following features: Gaussian blur, Median, Anisotropic Diffusion, Bilateral, Lipschitz and Kuwahara.

Fig. 3.4 shows some examples of these features on an Transmission Electron Microscopy (TEM) image using different sigmas.

These features are obviously very useful when processing noisy images (unless we are using the classifier to identify noise) but also when one of the pre-defined classes contains pixels with a wide range of intensities. For instance, in the image shown in Fig. 3.4a, if our target is to classify pixels as “membrane” or “non-membrane” pixels, this type of features will come very handy. You can observe how the anisotropic diffused and Kuwahara filtered versions of the original image (Fig. 3.4b and 3.4d) present very homogenous intensity values for those specific regions, therefore facilitating their identification.
3.4 Membrane detection features

This type of features were created specifically in TWS for the task of identifying, enhancing and (sometimes) reconstructing membrane-like structures in our images. They are based on directional filters (of a size defined by the user) that enhance bright and dark elongated structures of a certain thickness (also defined by the user) and along different directions. In TWS plugin and library, these features are under the name *Membrane projections*.

Fig. 3.5 shows an example of these features applied to a Light Microscopy (LM) fluorescence image containing cell membranes with irregular intensities.

Membrane detection features can be used not only to improve the signal of noisy membrane-like objects in the image, but also to close gaps that might appear due to the biological sample processing [Kaynig et al., 2010]. Moreover, these features are helpful as well to classify tree-like structures such as vessels or neural arbors.

3.5 Feature Selection

By default, TWS runs in 2D with a large variety of features (namely *Gaussian blur, Sobel filter, Hessian, Difference of Gaussians* and *Membrane projections*) and sigmas (1, 2, 4, 8 and 16). If these settings are not changed, this means the creation of a total of 76 features, i.e., 76 versions of the input image in either 32-bit format (for grayscale images) or RGB format (for color images, which also include 3 extra features with the corresponding Hue,
3.5 Feature Selection

![Original LM image](image1.png)  ![Membrane projection](image2.png)  ![Zoom on original image](image3.png)  ![Zoom on membrane projection](image4.png)

**Figure 3.5:** Membrane detection features on a fluorescence image (Cell Image Library, CIL39686).

Saturation and Brightness values). Therefore, the user needs to take into account that, while this number of features can be calculated very fast for small images thanks to TWS multi-threaded implementation, the computational time will increase with the size of the training images. Fortunately, the image features only need to be calculated once for each setting configuration. Furthermore, the number of features used will affect the computer memory consumption, since they need to be loaded in memory during the learning process. On top of that, some classifiers can be seriously mislead by the presence of many non-informative features in the training samples, although that is not the case of TWS default classifier, a random forest. In 3D, TWS starts with a minimum set of features (Mean and Variance) but similar issues can be found, especially for large images.

For all the reasons mentioned above, the user should choose carefully the type and size of the image features to be used. Here you are a few general recommendations:

1. **Think on the type of image features that allow you to visually differentiate regions of interest** as a human and try to find an equivalent set of features in the feature list. For example, use texture descriptors if your regions of interest have distinct patterns, or use border detectors if you are interested on changes in the contrast of the image.

2. **Select the size of the features (minimum and maximum sigmas) based on the size of the structures of interest in the image.** The sigmas allow you to define a field of view, i.e., how many pixels/voxels around the center pixel the feature will take into account for its computation. You can approximate them by visually inspecting the
regions of interest and zooming in and out the image. For example, if you want to classify a region based on some patterns, select a maximum sigma so some of those patterns are inside the field of view.

3. **Start with a low number of features and progressively add more features based on performance.** If a feature does not improve the result, deselect it. You can either visually evaluate the performance by looking at the classification result image or, even better, use the numerical estimation provided by the out-of-bag error of the default random forest classifier.

Of course, none of these recommendations are very useful unless proper training samples are provided by the user. The **selection of enough samples that are representative** of the different existing classes is as crucial as the correct image feature selection. In that sense, using a balanced number of class samples is usually a good idea as well.
4 Library use

The TWS GUI is independent from the methods. The methods are implemented in separate files in a library-style fashion, so they can be called from any Fiji plugin without having to interact with the GUI. This facilitates its integration with other plugins and allows easy scripting.

4.1 Scripting the Trainable Segmentation

Scripting is one of the reasons Fiji is so powerful, and the Trainable Segmentation library (that includes the TWS plugin methods) is one of the best examples for scriptable Fiji components.

4.1.1 Getting started

The first thing you need to start scripting the Trainable Segmentation is to know which methods you can use. For that, please have a look at the API of the Trainable Segmentation library\(^1\).

Let’s go through the basic commands with examples written in Beanshell:

4.1.1.1 Initialization

In order to include all the library methods, the easiest (but not so elegant) way of doing it is importing the whole library:

```java
import trainableSegmentation.*;
```

Now we are ready to play. We can open our input image and assign it to a WekaSegmentation object or segmentator:

```java
// input train image
input = IJ.openImage("input–grayscale–or–color–image.tif");
// create Weka Segmentation object
segmentator = new WekaSegmentation(input);
```

As it is now, the segmentator has default parameters and default classifier. That means that it will use the same features that are set by default in the TWS plugin, 2 classes (named “class 1” and “class 2”) and a random forest classifier with 200 trees and 2 random features per node. If we are fine with that, we can now add some labels for our training data and train the classifier based on them.

\(^1\)http://javadoc.imagej.net/Fiji/trainableSegmentation/package-tree.html
4.1 Scripting the Trainable Segmentation

4.1.1.2 Adding training samples

There are different ways of adding labels to our data:

1. We can add any type of ROI to any of the existing classes using the method “addExample”:

```java
// add pixels to first class (0) from ROI in slice # 1
segmentator.addExample(0, new Roi(10, 10, 50, 50), 1);
// add pixels to second class (1) from ROI in slice # 1
segmentator.addExample(1, new Roi(400, 400, 30, 30), 1);
```

2. We can add samples defined by the labels of a binary image, where white pixels belong to one class and black pixels belong to the other class. There are a few methods to do this, for example:

```java
// open binary label image
labels = IJ.openImage("binary-labels.tif");
// for the first slice, add white pixels as labels for class 2 and black pixels as labels for class 1
segmentator.addBinaryData(labels, 0, "class 2", "class 1");
```

3. We can also add samples from a new input image and its corresponding labels:

```java
// open new input image
input2 = IJ.openImage("input-image-2.tif");
// open corresponding binary label image
labels2 = IJ.openImage("binary-labels-2.tif");
// for all slices in input2, add white pixels as labels for class 2 and black pixels as labels for class 1
segmentator.addBinaryData(input2, labels2, "class 2", "class 1");
```

4. If we want to balance the number of samples for each class, we can do it in a similar way using this other method:

```java
// number of training samples of each class
numSamples = 1000;
// for all slices in input2, add 1000 white pixels as labels for class 2 and 1000 black pixels as labels for class 1
segmentator.addRandomBalancedBinaryData(input2, labels2, "class 2", "class 1", numSamples);
```

In general, we can use all methods available in the API to add labels from a binary image in many different ways. Please, have a look at them and decide which one fits better your needs.

4.1.1.3 Training a classifier

Once we have training samples for all the defined classes, we are ready to train the classifier of our segmentator:
This may take a while depending on the number of training samples and features we use.

### 4.1.1.4 Applying classifier (getting results)

Once the classifier is trained (what will be displayed in the Log window), we can apply it to the entire training image and obtain a result in the form of a label image or a probability map for each class:

```java
// apply classifier to current training image and get label result
// (set parameter to true to get probabilities)
segmentator.applyClassifier(false);

// get result (float image)
result = segmentator.getClassifiedImage();
```

Of course, we might be interested on applying the trained classifier to a complete new 2D image or stack. In that case we use:

```java
// open test image
testImage = IJ.openImage("test-image.tif");

// get result (labels float image)
result = segmentator.applyClassifier(testImage);
```

### 4.1.1.5 Save and load operations

If the classifier you trained is good enough for your purposes, you may want to save it into a file for future use:

```java
// save classifier into a file (Weka .model format)
segmentator.saveClassifier("my-cool-trained-classifier.model");
```

And load it later in another script to apply it on new images:

```java
// load classifier from file
segmentator.loadClassifier("my-cool-trained-classifier.model");
```

You may also want to save the training data into a file you can open later in Weka:

```java
// save data into a ARFF
file segmentator.saveData("my-traces-data.arff");
```

Or load a file with traces information into the segmentator to use it as part of the training:

```java
// load training data from ARFF file
segmentator.loadTrainingData("my-traces-data.arff");
```

### 4.1.1.6 Setting the classifier

By default, the classifier is a multi-threaded implementation of a random forest. You can change it to any other classifier available in the Weka API\(^2\). For example, we can use a support vector machine classifier (named SMO in Weka):

\(^2\)http://weka.sourceforge.net/doc.dev/weka/classifiers/Classifier.html
4.1 Scripting the Trainable Segmentation

```java
import weka.classifiers.functions.SMO;
// create new SMO classifier (default parameters)
classifier = new SMO();
// assign classifier to segmentator
segmentator.setClassifier(classifier);
```

We might also want to use the default random forest classifier but tune its parameters. In that case, we can write something like this:

```java
import hr.ibr.fastRandomForest.FastRandomForest;
// create random forest classifier
rf = new FastRandomForest();
// set number of trees in the forest
rf.setNumTrees(100);
// set number of features per tree (0 for automatic selection)
rf.setNumFeatures(0);
// set random seed
rf.setSeed((new java.util.Random()).nextInt());
// set classifier
segmentator.setClassifier(rf);
```

4.1.2 Script examples

4.1.2.1 Applying classifier to all images in folder

Very frequently we might end up having to process a large number of images using a classifier that we interactively trained with the GUI of the TWS plugin. The following Beanshell script shows how to load a classifier from file, apply it to all images contained in a folder and save the results in another folder defined by the user:

```java
// @File(label="Input directory", description="Select the directory with input images", style="directory") inputDir
// @File(label="Output directory", description="Select the output directory", style="directory") outputDir
// @File(label="Weka model", description="Select the Weka model to apply") modelPath
// @String(label="Result mode", choices={"Labels", "Probabilities"}) resultMode
import trainableSegmentation.WekaSegmentation;
import ij.io.FileSaver;
import ij.IJ;
import ij.ImagePlus;

startTime = System.currentTimeMillis();

getProbs = resultMode.equals("Probabilities");

listOfFiles = inputDir.listFiles();
for (i = 0; i < listOfFiles.length; i++) { 
    if (listOfFiles[i].isFile()) {
        // try to read file as image
        image = new ImagePlus(listOfFiles[i].getCanonicalPath());
```
if ( image != null )
{
    // create segmentator
    segmentator = new WekaSegmentation( image );
    // load classifier
    segmentator.loadClassifier( modelPath.getCanonicalPath() );
    // apply classifier and get results
    result = segmentator.applyClassifier( getProbs );
    // save result as TIFF in output folder
    outputFileName = listOfFiles[ i ].getName().replaceFirst("[.][^\+]+$", "") + ".tif";
    new FileSaver( result ).saveAsTiff( outputDir.getPath() + File.separator +
        outputFileName );

    // force garbage collection (important for large images)
    segmentator = null;
    result = null;
    image = null;
    System.gc();
}

// print elapsed time
estimatedTime = System.currentTimeMillis() - startTime;
IJ.log("** Finished processing folder in " + estimatedTime + " ms **");

4.1.2.2 Defining your own features

Although TWS provides a large set of predefined image features, it might happen that you need to define your own features for a specific problem. You can do that with a simple set of instructions. Here is a little Beanshell script that makes two features from the Clown example and uses them to train a classifier (see the inline comments for more information):

```java
import ij.IJ;
import ij.ImagePlus;
import ij.ImageStack;
import ij.gui.Roi;
import ij.gui.PolygonRoi;
import ij.plugin.Duplicator;
import ij.process.FloatPolygon;
import ij.process.StackConverter;
import trainableSegmentation.FeatureStack;
import trainableSegmentation.FeatureStackArray;
import trainableSegmentation.WekaSegmentation;

image = IJ.openImage(System.getProperty("ij.dir") + "/samples/clown.jpg");
if (image.getNumStacks() > 1)
    new StackConverter(image).convertToGray32();
else
    image.setProcessor(image.getProcessor().convertToFloat());

duplicator = new Duplicator();

// process the image into different stacks, one per feature:
smoothed = duplicator.run(image);
IJ.run(smoothed, "Gaussian Blur...", "radius=20");
medianed = duplicator.run(image);
IJ.run(medianed, "Median...", "radius=10");
```
4.1 Scripting the Trainable Segmentation

// add new feature here (1/2)

// the FeatureStackArray contains a FeatureStack for every slice in our original image
featuresArray = new FeatureStackArray(image.getStackSize());

// turn the list of stacks into FeatureStack instances, one per original slice. Each FeatureStack contains exactly one slice per feature.
for (slice = 1; slice <= image.getStackSize(); slice++) {
    stack = new ImageStack(image.getWidth(), image.getHeight());
    stack.addSlice("smoothed", smoothed.getStack().getProcessor(slice));
    stack.addSlice("medianed", medianed.getStack().getProcessor(slice));
}

// add new feature here (2/2) and do not forget to add it with a unique slice label!
features = new FeatureStack(stack.getWidth(), stack.getHeight(), false);
features.setStack(stack);
featuresArray.set(features, slice - 1);
featuresArray.setEnabledFeatures(features.getEnabledFeatures());
}

wekaSegmentation = new WekaSegmentation(image);
wekaSegmentation.setFeatureStackArray(featuresArray);

// set examples for class 1 (= foreground) and 0 (= background)
void addExample(int classNum, int slice, float[] xArray, float[] yArray) {
    polygon = new FloatPolygon(xArray, yArray);
    roi = new PolygonRoi(polygon, Roi.FREELINE);
   IJ.log("roi: " + roi);
    wekaSegmentation.addExample(classNum, roi, slice);
}

// train classifier
if (!wekaSegmentation.trainClassifier())
    throw new RuntimeException("Uh oh! No training today.");

output = wekaSegmentation.applyClassifier(image);
output.show();
4.1 Scripting the Trainable Segmentation

If the script is executed without errors, the output image should look like the one shown in Fig. 4.1 (although sometimes might look different due to the random initialization of the classifier and the low number of features used).

![Figure 4.1: From left to right: Clown sample image and its binary classification result using two custom features (script from Sec. 4.1.2.2).](image)

4.1.2.3 Defining training samples with binary labels

Here is a simple script in Beanshell doing the following:

1. It takes one image (2D or stack) as training input image and a binary image as the corresponding labels.

2. Train a classifier (in this case a random forest, but it can be changed) based on randomly selected pixels of the training image. The number of samples (pixels to use for training) is also a parameter, and it will be the same for each class.

3. Apply the trained classifier to a test image (2D or stack).

```java
//@ImagePlus(label="Training image", description="Stack or a single 2D image") image
//@ImagePlus(label="Label image", description="Image of same size as training image containing binary class labels") labels
//@ImagePlus(label="Test image", description="Stack or a single 2D image") testImage
//@Integer(label="Num. of samples", description="Number of training samples per class and slice", value=2000) nSamplesToUse
//@OUTPUT ImagePlus prob

import ij.IJ;
import trainableSegmentation.WekaSegmentation;
import hr.irb.fastRandomForest.FastRandomForest;

startTime = System.currentTimeMillis();

seg = new WekaSegmentation(image);

rf = new FastRandomForest();
```
4.1 Scripting the Trainable Segmentation

```java
// Number of trees in the forest
rf.setNumTrees(100);

// Number of features per tree
rf.setNumFeatures(0);

// Seed
rf.setSeed((new java.util.Random()).nextInt());

// set classifier
seg.setClassifier(rf);

// Parameters
// membrane patch size
seg.setMembranePatchSize(11);
// maximum filter radius
seg.setMaximumSigma(16.0f);

// Selected attributes (image features)
enableFeatures = new boolean[]{
  true, // Gaussian_blur */
  true, // Sobel_filter */
  true, // Hessian */
  true, // Difference_of_gaussians */
  true, // Membrane_projections */
  false, // Variance */
  false, // Mean */
  false, // Minimum */
  false, // Maximum */
  false, // Median */
  false, // Anisotropic_diffusion */
  false, // Bilateral */
  false, // Lipschitz */
  false, // Kuwahara */
  false, // Gabor */
  false, // Derivatives */
  false, // Laplacian */
  false, // Structure */
  false, // Entropy */
  false, // Neighbors */
};

// Enable features in the segmentator
seg.setEnabledFeatures(enableFeatures);

// Add labeled samples in a balanced and random way
seg.addRandomBalancedBinaryData(image, labels, "class 2", "class 1", nSamplesToUse);

// Train classifier
seg.trainClassifier();

// Apply trained classifier to test image and get probabilities
prob = seg.applyClassifier(testImage, 0, true);

// Set output title
prob.setTitle("Probability maps of " + testImage.getTitle());

// Print elapsed time
estimatedTime = System.currentTimeMillis() - startTime;
IJ.log("** Finished script in " + estimatedTime + " ms **");
```

4.1.2.4 Color-based segmentation using clustering

The following Beanshell script shows how to segment a 2D color image or stack in an automatic fashion using the CIELab color space and two possible clustering schemes: k-means
4.1 Scripting the Trainable Segmentation

and expectation maximization (note: if you do not have Weka’s ClassificationViaClustering
classifier installed, check how to install new classifiers via Weka’s package manager3).

```java
// @ImagePlus image
// @int(label="Num. of clusters", description="Number of expected clusters", value=5) numClusters
// @int(label="Num. of samples", description="Number of training samples per cluster", value=1000) numSamples
// @String(label="Clustering method", choices="[SimpleKMeans,"EM"]") clusteringChoice
// @OUTPUT ImagePlus output
import ij.IJ;
import ij.ImageStack;
import ij.ImagePlus;
import ij.process.ColorSpaceConverter;
import ij.process.ByteProcessor;
import trainableSegmentation.FeatureStack;
import trainableSegmentation.FeatureStackArray;
import trainableSegmentation.WekaSegmentation;
import weka.clusterers.EM;
import weka.clusterers.SimpleKMeans;

// Load WEKA and Trainable Weka Segmentation learning schemes
new Weka_Segmentation();

// Color space converter to pass from RGB to Lab
converter = new ColorSpaceConverter();

// Initialize segmentator with the same number of classes as expected number of clusters
wekaSegmentation = new WekaSegmentation( image );
for ( i=2; i<numClusters; i++ )
wekaSegmentation.addClass();

// Initialize array of feature stacks (one per slice)
featuresArray = new FeatureStackArray( image.getStackSize() );
for ( slice = 1; slice <= image.getStackSize(); slice++ ) {
  // RGB to Lab conversion
  stack = new ImageStack( image.getWidth(), image.getHeight() );
  lab = converter.RGBToLab( new ImagePlus( "RGB", image.getStack().getProcessor( slice ) ) );
  stack.addSlice("a", lab.getStack().getProcessor( 2 ) );
  stack.addSlice("b", lab.getStack().getProcessor( 3 ) );
}

// Create empty feature stack
features = new FeatureStack( stack.getWidth(), stack.getHeight(), false );
// Set a and b features to the feature stack
features.setStack( stack );
// Put feature stack into the array
featuresArray.set(features, slice - 1);

// Create uniform labels of each cluster/class.
// (this information is not used by the clusterer but
pixels = new byte[ image.getWidth() + image.getHeight() ];
for ( i=0; i<pixels.length; i++ )
pixels [ i ] = (byte) ( i % numClusters + 1 );
labels = new ByteProcessor( image.getWidth(), image.getHeight(), pixels );

3http://imagej.net/Trainable_Weka_Segmentation_-_How_to_install_new_classifiers
```
4.1 Scripting the Trainable Segmentation

```java
// Add randomly chosen training data in a balanced way
wekaSegmentation.addRandomBalancedLabeledData( labels, features, numSamples);

// Set classifier to perform clustering
classifier = new ClassificationViaClustering();
// Set clusterer as selected by user
clusterer = null;
if (clusteringChoice.equals("SimpleKMeans"))
  clusterer = new SimpleKMeans();
else
  clusterer = new EM();
clusterer.setSeed((new Random()).nextInt());
clusterer.setNumClusters(numClusters);
classifier.setClusterer(clusterer);
wekaSegmentation.setClassifier(classifier);

// Train classifier and therefore clusterer
if (!wekaSegmentation.trainClassifier())
  throw new RuntimeException("Uh oh! No training today.");

// Apply classifier based on a,b features to whole image
wekaSegmentation.setFeatureStackArray(featuresArray);
output = wekaSegmentation.applyClassifier(image, featuresArray, 0, false);
output.setDisplayRange(0, numClusters - 1);
```

This can be a very useful approach to segment images where the elements contain very distinct colors. Let's see an example using a public image of hematoxylin and eosin (H&E) stained lung tissue:

![H&E stained lung tissue](https://commons.wikimedia.org/wiki/File:Emphysema_H_and_E.jpg)

Once the image is open, we can call the script and a dialog will pop up:
Here we can select the number of expected clusters, the number of samples per cluster used for training and the clustering method. The default values of 5 clusters, 1000 samples and “SimpleKMeans” involve that 5000 pixels will be used for training ($5 \times 1000 = 5000$) a $k$-means classifier and the resulting image will be an integer image containing labels in the range of $[0 – 4]$.

This would be a possible output of the script with 3 clusters, 2000 samples and “SimpleKMeans”:

The actual label values may vary between different executions of the same clustering due to its random seed initialization. In any case, the blood cells (originally in red), the cell nuclei (in blue-purple), other cell bodies (in pink) and the extracellular space get usually a very reasonable segmentation.
5 Practical issues

5.1 Usage with existing installation of Weka

By default, Weka will automatically load plugins installed under ~/wekafiles. If you already have an existing installation of Weka using Java 1.7 and are seeing an error about java.lang.UnsupportedClassVersionError: weka/filters/unsupervised/attribute/Independent-Components: Unsupported major.minor version 51.0, then you should remove or rename the ~/wekafiles folder before running Fiji.

5.2 Weka core version

Since the 3.2.0 release, TWS uses Weka 3.9.0 - development version. If you have problems loading models from previous versions of the plugin/library, most likely you need to recreate the models using the new version (see note 1 of the Weka official release1).

If you absolutely need to reuse an old model, you can transform it to the new version thanks to a model migrator tool2 provided by the Weka developers. For more information, check this post in the ImageJ forum:
http://forum.imagej.net/t/weka-segmentation-error-after-update-29-09-16/2898/23

5.3 Troubleshooting

For all questions, suggestions, bug reports and problems related to the TWS plugin or library, please use the ImageJ forum (http://forum.imagej.net/) and make sure to check previous posts that might have been done covering the same topic.

---

1http://forums.pentaho.com/showthread.php?204301-New-Weka-3-6-14-3-8-0-and-3-9-0-releases!
2http://www.cs.waikato.ac.nz/ml/weka/downloading.html
Bibliography


Bibliography

Magdalena Frank, Stéphane Duvezin-Caubet, Sebastian Koob, Angelo Occhipinti, Ravi Jagasia, Anton Petcherski, Mika O Ruonala, Muriel Priault, Bénédicte Salin, and Andreas S Reichert. Mitophagy is triggered by mild oxidative stress in a mitochondrial fission dependent manner. *Biochimica et Biophysica Acta (BBA)-Molecular Cell Research*, 2012.


