Wojciech Czarnecki

Uniwersytet Jagielloński

AIClassifier - The Generic ImageJ Segmentation Plugin

Praca semestralna nr 1

(semestr letni 2011/12)

Opiekun pracy: Zygmunt Vetulani

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

AIClassifier - The Generic ImageJ Segmentation Plugin

Wojciech Czarnecki

Department of Computer Linguistics and Artificial Intelligence,
Faculty of Mathematics and Computer Science,
Adam Mickiewicz University, Poznan, Poland
`w.czarnecki@amu.edu.pl`

Abstract

Image segmentation is one of the fundamental concerns of the computer vision. Unfortunately, there is no generic solution for all of its possible instances, which leads to need of developing various methods for wide range of problem domains.

In this paper we describe AIClassifier - plugin to support performing segmentation tasks using ImageJ. Its most distinguishable feature is its multi-level approach to the segmentation task with particular interest in methods of projecting image to the feature space. Introduction of another abstraction layer (projection to the feature space) and providing the set of tools for developing it together with a custom scripting language for expressing matrix based image features makes it fill the gap in existing solutions for the image segmentation problem.

1 Introduction

The problem of image segmentation, simplifying representation of an image into something that is easier to further analysis (Shapiro and Stockman, 2001) is one of the basic issues of image analysis/computer vision. Numerous approaches has been presented during past years, showing, that this is still a challenging problem in many fields (from surveillance systems, through robotics to medicine). Machine learning (supervised and unsupervised) methods are currently the most common, as the task of generating an exact set of rules for segmenting the image is often too complex for a human. For this reason, it is reasonable to create tools for people in this field, developing dozens of different models in order to find the best one for their task. In this paper we describe AIClassifier - the ImageJ plugin for supporting creation of segmentation pipelines using custom feature space projections and various classifiers.

The selection of ImageJ (Abrmoff et al., 2004) plugin as an architecture was based on many aspects, first of all - it is a big, open source (public domain), multi platform (as a consequence of being written in Java) application, which has been continuously developed for over 15 years. It is one of the most commonly used tool for digital image processing and analysis, especially in bioinformatics and medicine fields. ImageJ is also commonly used as the educational tool on universities offering Image processing/analysis/recognition courses. There are currently thousands of existing plugins and filters, that can be integrated with AIClassifier (as both pre- and postprocessing steps), while in the same time (to the authors best knowledge) there is no plugin offering at least part of our plugin features. This makes

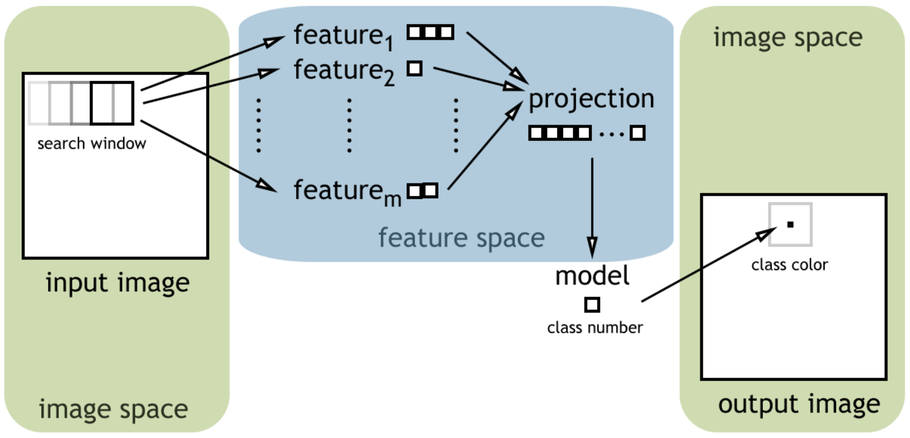


Figure 1: General plugin execution scheme

AIClassifier fill the gap in the existing set of tools for ImageJ on one hand, and benefit from a huge amount of existing tools, and a great number of potential users on the other.

2 Plugin concept

As a part of the ImageJ, plugin is also written in Java (compatible with 1.4 and above) and released on the open license (public domain) so anyone can study, run, copy, redistribute and improve it freely. While AIClasifier was designed, the main emphasis was put on its flexibility and modular architecture so it can be used as a segmentation pipeline development toolkit rather than simple segmentation plugin. Classical image segmentation pipeline consists of five steps:

1. Preprocessing
2. Sampling
3. Feature extraction (defining projection to the feature space)
4. Classification
5. Segmentation visualization

AIClassifier offers tools for rapid development of such methods from the sampling step to the results visualization, leaving the preprocessing to other plugins/filters available in ImageJ.

We decided to focus on the segmentation methods based on local pixels features which are one of the most commonly used approaches in many computer vision tasks (e.g. cell nuclei detection (Han et al., 2010), face detection (Mohamed et al., 2008) or cars detection (Papageorgiou and Poggio, 2000)). According to this assumption, plugin searches through whole image using a constant size window, and classifies each pixel using some model (classifier) on projection of its neighborhood window into feature space (fig.1).

3 Structure

The most important part of the AIClassifier is its modular architecture, which enables the user to easily and quickly exchange parts of the pipeline to satisfy the current needs.

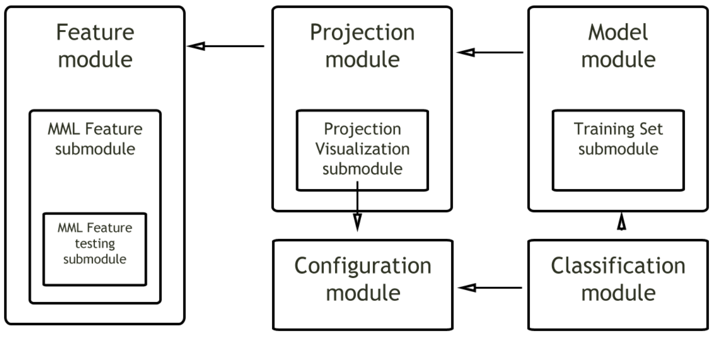


Figure 2: Modules dependencies structure. Arrows shows which module uses which one

Plugin consists of four main modules (fig.2):

- Features module, that allows development of various image features. Each of them can be saved in a separate file (.java, .mml, .class) and used in many various projections.
- Projection module, that allows connecting many features in one projection that can be then saved (in the .proj file), used for later segmentation or visualized using a Projection Visualization submodule.
- Model module, that allows selection among different types of classifiers that can be used with selected projection. Each of them can be saved in a separate .model file and used in many various segmentation tasks. There is also a Training Set submodule that allows generation of training set schemes for supervised classifiers.
- Classification module, that performs actual segmentation using provided parameters (like window size, step size, used model or selected set of classes colors).

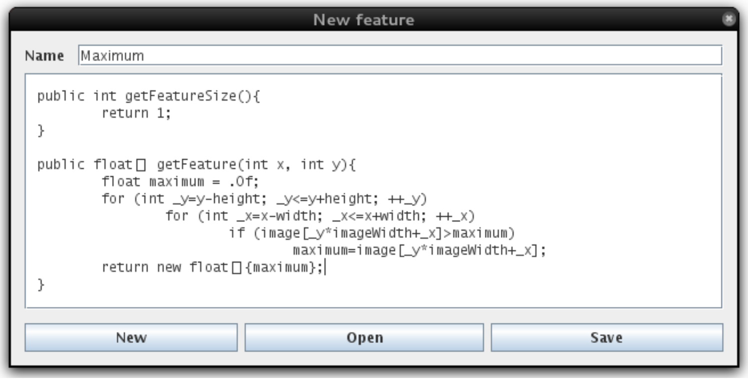
Each of these modules is responsible for one abstraction level of the segmentation process and is responsible for providing user with configuration/testing and/or visualization options for the current step.

3.1 Features

For a given image pixels intensity function $I(x, y)$ we can define image window centered in (x_0, y_0) of width w and height h as:

$$W_{I,x_0,y_0,w,h} = (I(x_0 - w, y_0 - h), I(x_0 - w + 1, y_0 - h), ..., I(x_0 + w, y_0 + h))$$

Formally speaking, feature is just a function f from the image (window) $\mathbb{N}^{width \times height}$ (in particular $width = 2w + 1$ and $height = 2h + 1$) space to the $\mathbb{R}^m$ space used for expressing the underlying structure of the pixels. The accurate choice of features for particular task is crucial for any computer vision problem (Szeliski, 2009). In AIClassifier user can define custom features that will be used during segmentation task. In particular, to achieve the simplest possible behavior where segmentation is based on the actual window pixels (and not any sophisticated features) one can define the identity feature, where simply $m = width \cdot height$ and $f(x) = x$. Features module allows putting features Java code directly in plugins window (fig.3), which then can be saved and used to define projections. This way the user can quickly define a whole set of features that will be used during work.



```
public int getFeatureSize(){
        return 1;
}

public float[] getFeature(int x, int y){
        float maximum = .0f;
        for (int _y=y-height; _y<=y+height; ++_y)
                for (int _x=x-width; _x<=x+width; ++_x)
                        if (image[_y*imageWidth+_x]>maximum)
                                maximum=image[_y*imageWidth+_x];
        return new float[]{maximum};
}
```

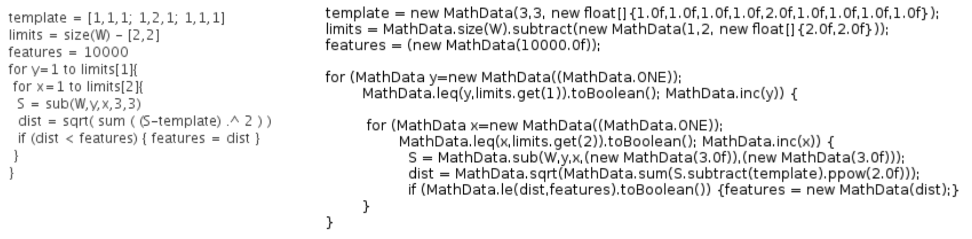
Figure 3: Features module window with the sample maximum feature.

One can also develop features in some external IDE using class templates provided with the plugin. Feature files are compiled and loaded (using reflection mechanism) when needed, so the user can always work on source files, leaving the compilation process to the plugin itself.

Unfortunately Java language does not support operators overloading and as a result - writing matrix operations can be very inconvenient. To speed up the process of generating features (which are in many cases based on matrices) and to make them more legible some kind of scripting language is required. Such language should be:

- simple and very easy to learn
- legible (in the sense of e.g. infix notation)
- compilable to the Java code (for efficiency reasons)
- supporting most of the common matrix operations used in computer vision (from multiplication, through substitutions to concatenations)
- light-weight

For purpose of this plugin, we developed such a simple language (which is also available on an open license).



```
template = [1,1,1; 1,2,1; 1,1,1]
limits = size(W) - [2,2]
features = 10000
for y=1 to limits[1]{
 for x=1 to limits[2]{
  S = sub(W,y,x,3,3)
  dist = sqrt( sum ( (S-template) .^ 2 ) )
  if (dist < features) { features = dist }
 }
}
```

```
template = new MathData(3,3, new float[]{1.0f,1.0f,1.0f,1.0f,2.0f,1.0f,1.0f,1.0f,1.0f});
limits = MathData.size(W).subtract(new MathData(1,2, new float[]{2.0f,2.0f}));
features = (new MathData(10000.0f));

for (MathData y=new MathData((MathData.ONE));
      MathData.leq(y,limits.get(1)).toBoolean(); MathData.inc(y)) {

       for (MathData x=new MathData((MathData.ONE));
             MathData.leq(x,limits.get(2)).toBoolean(); MathData.inc(x)) {
               S = MathData.sub(W,y,x,(new MathData(3.0f)),(new MathData(3.0f)));
               dist = MathData.sqrt(MathData.sum(S.subtract(template).ppow(2.0f)));
               if (MathData.le(dist,features).toBoolean()) {features = new MathData(dist);}
       }
}
```

Figure 4: MML code for calculating minimum euclidean distance to the *template* among all 3x3 submatrices of W (on the left side) and equivalent Java code from the MML parser (on the right side)

Micro Matrix Language (MML) is a simple scripting language developed for performing matrix operations in Java applications. Its syntax is based partially on C++ and partially

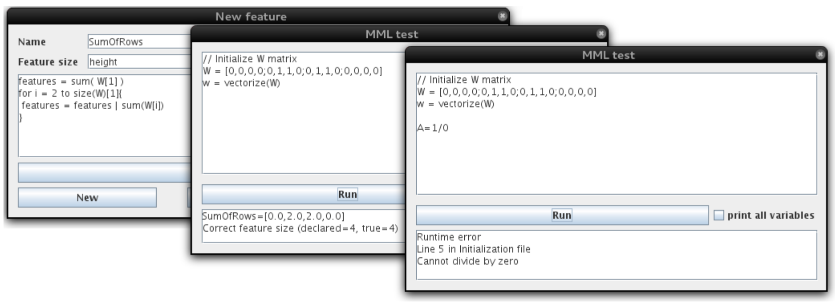
action	MML code
$A := \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	$A = [1, 2, 3; 4, 5, 6]$
$B := A^T$	$B = A'$
$B := A + 2A - 1$	$B = A + 2 * A - 1$
substitute first column of A by $\begin{pmatrix} 7 \\ 8 \end{pmatrix}$	$A[1] = [7; 8]$
substitute second row of A by $\begin{pmatrix} 9 & 10 & 11 \end{pmatrix}$	$A[2] = [9, 10, 11]$
add (concatenate) a column of zeros to A	$A = A \mid [0; 0]$
store element-wise multiplication of A and A in B	$B = A. * A$
substitute bolded elements of $\begin{pmatrix} a_{11} & \mathbf{a_{12}} & \mathbf{a_{13}} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = A$ with $\begin{pmatrix} 9 & 10 \end{pmatrix}$	$A[1][2] = [9, 10]$
store the maximum element of A in B	$B = max(A)$

Table 1: Sample MML code for simple matrix operations

on Octave/R (fig.4) to ensure that an end-user can start using it with a small amount of time spent on learning it (tab.1). It is parsed directly to the Java code, so features developed using this language have similar efficiency to those written natively in Java.

Current implementation of MML includes:

- simple matrix operators: $+ \quad - \quad * \quad / \quad \% \quad \hat{}$
- element-wise matrix operators: $.* \quad ./ \quad .\% \quad .\hat{}$
- matrix concatenation operators: $| \quad \_$
- logical operators: $> \quad >= \quad < \quad <= \quad == \quad != \quad and \quad or \quad not$
- mathematical constants: $e \quad pi$
- set of functions: $abs, max, min, mean, sum, count, zeros, ones, ident, sub, vectorize$
- $if/else$ statements
- for and $while$ loop



SumOfRows

height

```
features = sum( W[1] )
for i = 2 to size(W)[1]{
 features = features | sum(W[i])
}
```

```
// Initialize W matrix
W = [0,0,0,0;0,1,1,0;0,1,1,0;0,0,0,0]
w = vectorize(W)
```

```
SumOfRows=[0.0,2.0,2.0,0.0]
Correct feature size (declared=4, true=4)
```

```
// Initialize W matrix
W = [0,0,0,0;0,1,1,0;0,1,1,0;0,0,0,0]
w = vectorize(W)

A=1/0
```

```
Runtime error
Line 5 in Initialization file
Cannot divide by zero
```

Figure 5: Feature module with sample MML code for calculating sum of image window rows (on the left), its output in the testing submodule (in the middle) and error message that appears when there is some incorrect operation in the code (on the right)

Feature expressed as the MML script is just a sequence of operations that given the input matrix W (holding normalized pixels' intensities data from the current window $W_{I,x,y,w,h}$) generates the output vector of features and stores it in the variable $features$.

User can develop MML features in the Features module by simply putting its name, code and dimensionality in the window . AIClassifier also provides very simple testing environment, that evaluates a feature on the given test input data, views values of all variables used during execution (for easier debug), shows parse and execution errors and checks correctness of provided dimensionality (fig.5).

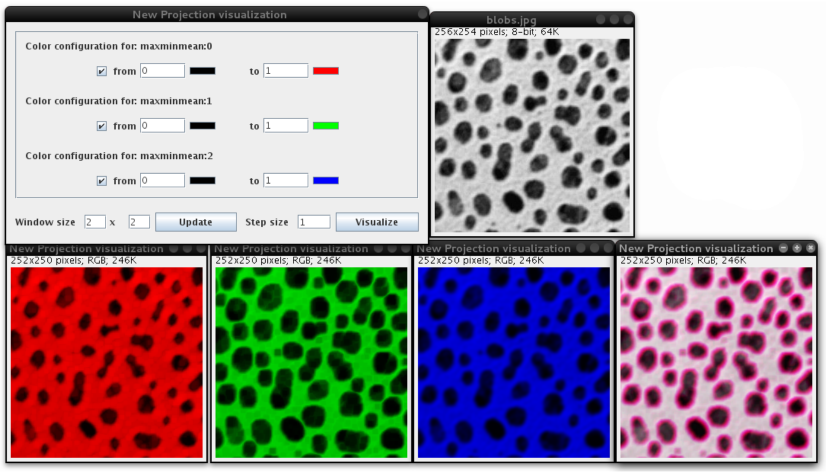
3.2 Projection

Having some finite sequence of features (functions) $f_j : \mathbb{N}^{width \times height} \rightarrow \mathbb{R}^{i_j}, j \in \{1, ..., k\}$ we can define projection of image window $W_{I,x_0,y_0,w,h}$ to the features space $\mathbb{R}^n$ as:

$$p(W_{I,x_0,y_0,w,h}) = (c_1, c_2, ..., c_n)$$

where $c_1 = f_1(W_{I,x_0,y_0,w,h})_1, ..., c_{i_1} = f_1(W_{I,x_0,y_0,w,h})_{i_1}, c_{i_1+1} = f_2(W_{I,x_0,y_0,w,h})_1, ...,$ and $n = \sum_{j=1}^{k} i_j$.

The user can create projections by adding previously defined features (.java, .mml or .class) using the Projection module. Once this step is completed one can save it for later use or analyze its behavior using the Projection Visualization submodule (fig.6). For each feature in the projection the user can assign (for each of its output dimensions) its minimum and maximum value and corresponding colors. Once this configuration is finished it is possible to visualize the projection (or a projection being any subsequence of its features) using any currently opened image in ImageJ. This process can be seen as an independent mechanism, which can be used for easy creation of complex image filters. For example a simple edge detection filter can be expressed as a black to white visualization of single amplitude feature (which can be expressed in MML as a one line code: $features = max(W) - min(W)$).


New Projection visualization
blobs.jpg
256x254 pixels; 8-bit; 64K
New Projection visualization
252x250 pixels; RGB; 246K
New Projection visualization
252x250 pixels; RGB; 246K
New Projection visualization
252x250 pixels; RGB; 246K
New Projection visualization
252x250 pixels; RGB; 246K
Update
Visualize

Color configuration for: maxminmean:0

[✓] from 0 ⬛ to 1 🟥

Color configuration for: maxminmean:1

[✓] from 0 ⬛ to 1 🟩

Color configuration for: maxminmean:2

[✓] from 0 ⬛ to 1 🟦

Window size 2 x 2 Step size 1

Figure 6: The Projection Visualization submodule with sample visualization of max, min and mean features (expressed in MML as $features = [max(W), min(W), mean(W)]$) in red, green and blue channels respectively. Whole projection is visualized on the rightmost image.

3.3 Model

Model module provides the user with a set of options to generate the final model for the classification task. The model consists of two parts: projection, created using the Projection module and the classifier, which can be selected from the dropdown list of available algorithms. The plugin supports both unsupervised and supervised classifiers, which allows the user to develop methods suitable for various tasks. The main task of each classifier is to assign each feature vector a corresponding class number so that the output classes are homogenous in some sense. Strictly speaking, a model m can be seen as a function from feature space to $\mathbb{N}$:

$$m((c_1, c_2, ..., c_n)) = o$$

where o is a corresponding class number.

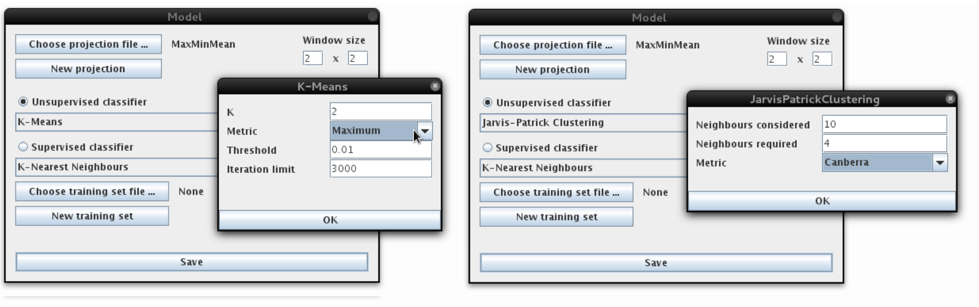
In supervised learning one needs to provide the set of input-output pairs that can be used to train a classifier. In our approach input vectors are feature space vectors, and output vectors are class numbers. Training set T of s pairs can be defined as:

$$T = \{((c_{i1}, c_{i2}, ..., c_{in}), o_i) : i \in \{1, 2, ..., s\}\}$$

where n is the dimensionality of the feature space and $o_i \in \mathbb{N}$ is the class number. We do expect that $\forall i \in \{1, 2, ..., s\}$ $m((c_{i1}, c_{i2}, ..., c_{in})) = o_i$ (of course this constraint is often loosen up, and we only try to maximize the amount of correctly classified examples or maximize the probability of correct classification, the exact definition depends on the classifier and training method). Generation of these data is realised by a Model module using a training set scheme provided by Training Set submodule. Training set scheme can consist of one of two types of data:

- list of images (windows) from hard disk and selected expected class numbers,
- list of pairs of images (input image - segmented image).

These two methods cannot be used together because of lack of direct correspondence between class numbers chosen in the first method and the image colors from the second one, so depending on the task the user has to select one of those approaches. Once created, a training set scheme can be saved and used in any model (and any projection) developed in the future. Actual generation of the training set is executed just before model's training, so there is already some projection p, that can be used to obtain feature space vectors from list of input images from training set scheme.


Model
Choose projection file ...
New projection
Choose training set file ...
New training set
Save
OK
Model
Choose projection file ...
New projection
Choose training set file ...
New training set
Save
OK

K-Means

MaxMinMean

Window size 2 x 2

[✓ Unsupervised classifier] K-Means

Supervised classifier K-Nearest Neighbours

None

K: 2
Metric: Maximum
Threshold: 0.01
Iteration limit: 3000

JarvisPatrickClustering

MaxMinMean

Window size 2 x 2

[✓ Unsupervised classifier] Jarvis-Patrick Clustering

Supervised classifier K-Nearest Neighbours

None

Neighbours considered: 10
Neighbours required: 4
Metric: Canberra

Figure 7: Sample configuration of the K-means classifier (on the left) and the Jarvis-Patrick Classifier (on the right).

There are currently seven classifiers implemented:

- Unsupervised:
  - K-means (Lloyd, 1982)
  - K-medians
  - K-medoids
  - DBScan (Ester et al., 1996)
  - Jarvis-Patrick Clustering (Jarvis and Patrick, 1973)
- Supervised:
  - K-nearest neighbours
  - Multi Layer Perceptron

Most of them (except MLP) accept as one of the configuration parameters (fig.7) a metric that should be used to calculate distances between vectors which adds another level of customization of the entire process. Distance measures are currently hard coded in the plugins class hierarchy and consist of Euclidean, Manhattan, Maximum, Canberra, Hamming, Hellinger and Jaccard metrics and Pearson and Spearmann distances.

3.4 Classification

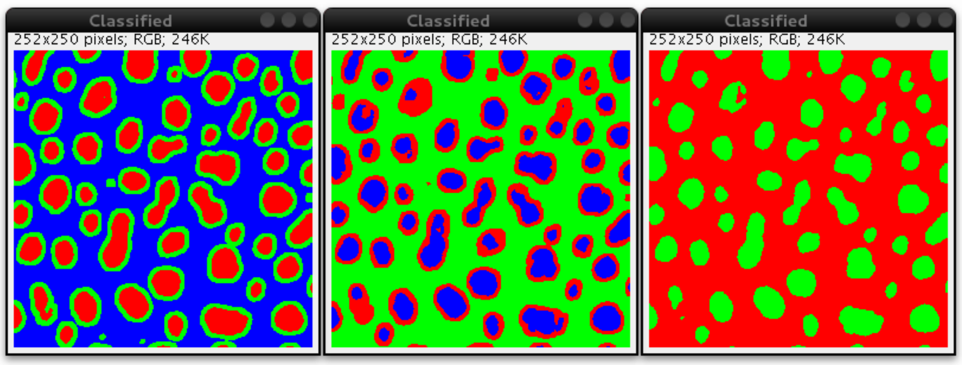

Classified
252x250 pixels; RGB; 246K
Classified
252x250 pixels; RGB; 246K
Classified
252x250 pixels; RGB; 246K

Figure 8: Example of image segmentations using maxminmean projection with different classifiers: 3-Means using Euclidean metric (the leftmost image), 3-Means using Canberra metric (in the middle), 2-Medoids using Hellinger metric (the rightmost image).

Classification module is the main module of the whole plugin. Having prepared different models the user uses it to perform actual segmentation of the currently selected image I in ImageJ. After setting segmentation parameters (size of the window w and h, step size s and the selected model m, which uses a projection p) the algorithm generates all image windows, projects them to the feature space and runs model's classification to obtain classes numbers which are then converted (based on the user configuration) back to the color space and placed in the output image as single pixels. For some preselected sequence of colors *color*, an output image $O(x, y)$ can be defined as:

$$O(x, y) = color_{m(p(W_{I,x+as,y+bs,w,h}))}$$

for all a and b such that point $(x + as, y + bs)$ is a correct location of some pixel of the input image. Sample segmentations using different models and projections can be seen on figure 8.

3.5 Implementation

Whole project is divided into two packages - AIClassifier.jar and MML.jar consisting plugin and implementation of MML language (parser and classes used in output code) respectively. This approach makes it possible to use MML by other developers in their own Java applications.

As stated before - all exchangeable elements of segmentation pipelines can be saved and loaded from files. While projection and model files are simple configuration files consisting information about data provided by the user - features are stored as fully functional functions. This approach is possible due to Java reflection mechanism that allows dynamic loading, creation and use of external classes at runtime. This lead to developing Projection class based on strategy software design pattern where exchangeable functionality are Feature functors. Instances of Features are created by a builder which using names of .class, .java or .mml files parses them (for .mml), compiles (for .mml and .java) and loads (for all of them) them to ReflectionFeature objects.

4 Summary

AIClassifier is a segmentation pipeline development tool with strong emphasis on speed and ease of customization. The proposed scripting language MML makes it even easier to develop various features that can be tested and analyzed without leaving the ImageJ. Included Projection Visualization submodule makes it also a good tool for the fast generation of image filters for other applications.

	AIClassifier	A	B	C
Amount of models	7	3	1	13
Projections	yes	no	2	no
Scripting	yes	no	no	no
Available metrics	9	1	1	1
Amount of output classes	any	any	2	2

Table 2: Feature comparison among some ImageJ segmentation plugins

A ij-plugins-toolkit, B Thresholding Plugin, C Multi-thresholding

Presented application fills the gap in the currently existing ImageJ segmentation plugins. By introducing another abstraction layer (projection to the feature space) and providing the set of tools for developing them it gives the user much more possibilities than any other ImageJ based solution (tab.2).

4.1 Future work

There are still many possible expansions of AIClassifier - one of the most important parts can be a deeper integration with ImageJ plugins mechanism. This would allow users to create ImageJ macros that directly execute parts of our plugin and as a result - automate parts of the process. This way also image filters developed using the Projection Visualization submodule could be added to ImageJ interface as separate plugins. The next step would be implementing the interface for external features (so it would be possible to write scripts in octave, R, python or any other language that supports command line interface). The set of available classifiers can be easily expanded by integrating some big classification library (e.g. WEKA (Hall et al., 2009)) which is currently an ongoing process. It would also be valuable to add customizable metrics (possibility to express them in Java or MML instead of precompiled set of metrics).

References

L. Shapiro and G. Stockman, *Computer Vision*, D. H. Ballard and C. M. Brown, Eds. Prentice Hall, 2001, vol. 2004, no. October.

M. D. Abrmoff, I. Hospitals, P. J. Magalhes, and M. Abrmoff, "Image processing with imagej," *Biophotonics International*, vol. 11, no. 7, pp. 36–42, 2004.

J. W. Han, T. P. Breckon, D. A. Randell, and G. Landini, "The application of support vector machine classification to detect cell nuclei for automated microscopy," *Machine Vision and Applications*, vol. 23, no. 1, pp. 15–24, 2010.

A. Mohamed, Y. Weng, J. Jiang, and S. Ipson, "Face detection based neural networks using robust skin color segmentation," *2008 5th International MultiConference on Systems Signals and Devices*, no. i, pp. 1–5, 2008.

C. Papageorgiou and T. Poggio, "A trainable system for object detection," *International Journal of Computer Vision*, vol. 38, no. 1, pp. 15–33, 2000.

R. Szeliski, "Computer vision: Algorithms and applications," *Computer*, vol. 2, p. 655, 2009.

S. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

M. Ester, H. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," *Proceedings of the 2nd International Conference on Knowledge Discovery and Data mining*, vol. 96, p. 226–231, 1996.

R. A. Jarvis and E. A. Patrick, "Clustering using a similarity measure based on shared near neighbors," *IEEE Transactions on Computers*, vol. C-22, no. 11, pp. 1025–1034, 1973.

M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *SIGKDD Explorations*, vol. 11, no. 1, pp. 10–18, 2009.