Automatic image segmentation techniques based on supervised machine learning algorithms

Dániel Grimm, BU44BJ

Eotvos Lorand University, Budapest

Abstract. Image segmentation, supervised machine learning algorithms, such as Random Forest Classification

1 General overview of the problem

1.1 Introduction

The goal of this project is to create a technique which can automatically segment images based on extracted image features. With the help of supervised machine learning algorithms, (in this case the Random Forest Classifier) there is no need for human interaction. While several image segmentation techniques exist; for example magic wand tool in Adobe Photoshop, they usually require manual interaction.

1.2 Description of the problem

Since the classifier has been trained on a specific set of training images, the input of the algorithm must be a very similar type of image (in this case leaf of grapes, with mostly homogeneous background).
Therefore the user can select one or more image(s) as input, and after running the algorithm he/she will receive segmented output image(s). The goal is to have only the leaves on the output pictures, the background has to be coloured with black colour, which means that part of the image is not part of the leaf.

2 Details of the method

2.1 The Pipeline

The pipeline consists of 6 main stages:

1. Get images, then split them into training/test set
2. Create image window locations in all train images
3. Create database from extracted image features from windows
4. Train model on data from the database
5. Predict input image with model
6. Calculate the accuracy of the output image

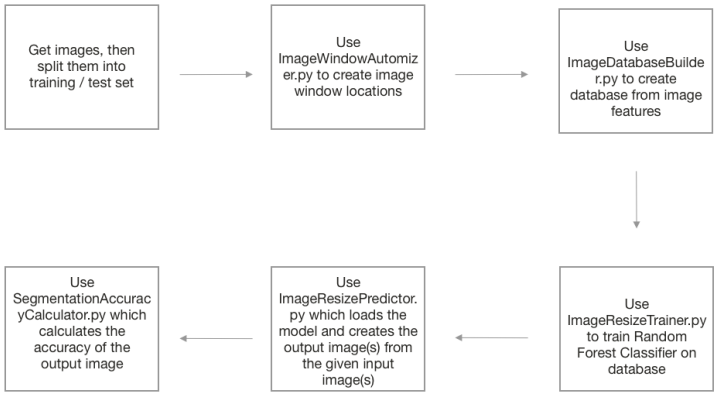

Fig. 1. Pipeline of the segmentation process

2.2 Stage 1: Getting images

The dataset contains images about grape leaves. The original dataset had 495 images but with very different backgrounds, and shapes. For example, there were images where multiple leaves were, or with different backgrounds like sky, wall, ground etc. I selected 50 most similar images for the training set and 10 images for the test set.

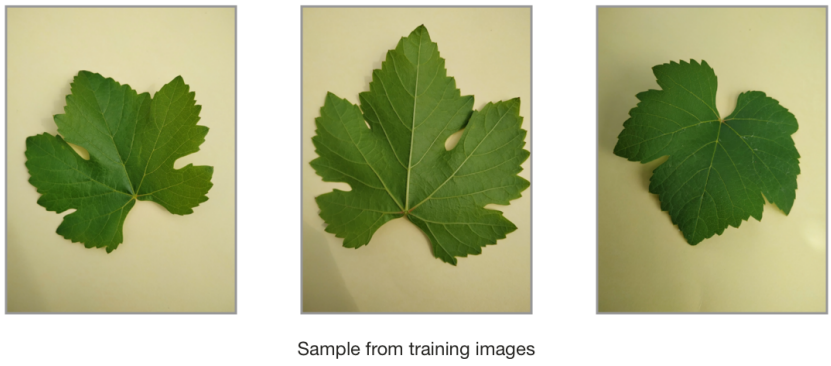

Fig. 2. Sample images from the training set

2.3 Stage 2: Creating image window locations

Inside all images (each image has a 2000x1500 size) I selected 2 little windows (each of them with a size of 106x106). The point behind this is that the algorithm which creates image features is based on multiple pixels, which means that it is important to have a border next to the pixels at the edges of the window, to be able to calculate the features there too. In this case, this border has a 3-pixel width, therefore we will get the window size by adding $3 + 100 + 3$ together.

Every image has 2 little windows, one is inside the grape leaf, and one is outside somewhere in the background.
I created a script, $ImageWindowAutomizer.py$ to automatize this process. What it does is that it will show all images from a specified folder one by one, then the user can click on the images, and the point where he/she clicked will be the $(0, 0)$ point of the little window. Left-click means the window is inside the leaf, right-click means it is outside. In the end, the script creates a .csv file with all coordinates and flags (inside/outside). Thus, it will be easy to automatically generate the database of the features from this .csv.

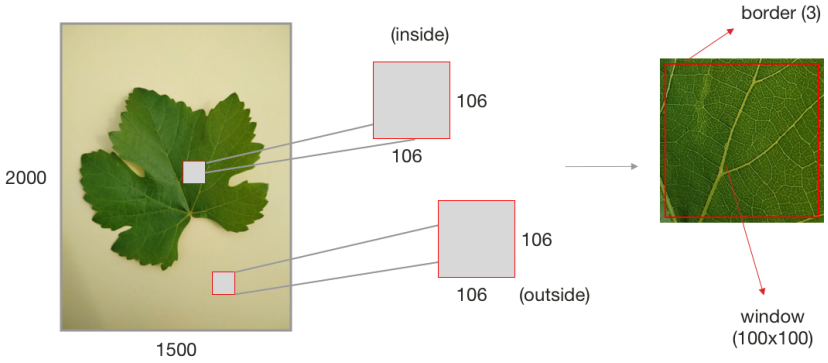

Fig. 3. Structure of an image window

2.4 Stage 3: Create database from image features

Data is generated in the following way: as the result of the previous step, each image has 2 windows, 1 is inside, and 1 is outside of the leaf. Each window consists 106x106 pixels, where 100x100 in the middle of the window is the "useful" part, and the border around the window (with a with/height of 3) is only for providing some pixels to the feature extracting.

The algorithm for extracting the image features is using the middle 100x100 pixels, by iterating over all of them. Each time for one pixel the data is gathered from a 7x7 sized field, where the middle one (index: (3,3)) is the actual pixel from the loop.

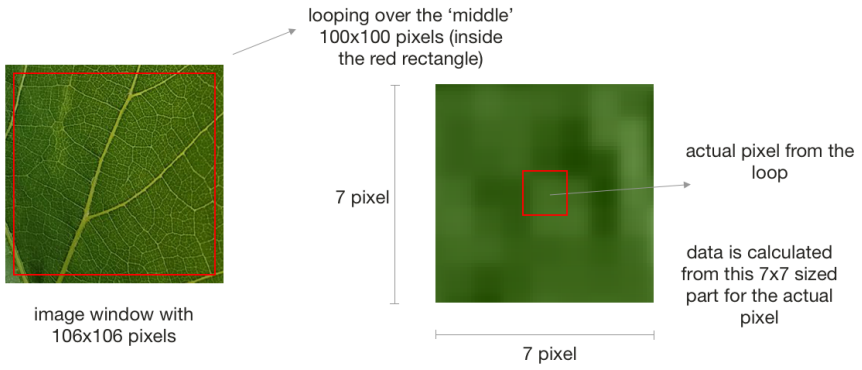

Fig. 4. Extracting data for every pixel

The following 12+1 features are extracted for every pixel and their 3pixel large neighbourhood:

1. **red_pixel**: the actual value of the red channel
2. **green_pixel**: the actual value of the green channel
3. **blue_pixel**: the actual value of the blue channel
4. **mean_red**: mean of red channel for all (7x7) pixels
5. **mean_green**: mean of green channel for all (7x7) pixels
6. **mean_blue**: mean of blue channel for all (7x7) pixels
7. **std_red**: standard deviation of red channel for all (7x7) pixels
8. **std_green**: standard deviation of green channel for all (7x7) pixels
9. **std_blue**: standard deviation of blue channel for all (7x7) pixels
10. **corr_rg**: correlation between red and green channel for all (7x7) pixels
11. **corr_rb**: correlation between red and blue channel for all (7x7) pixels
12. **corr_gb**: correlation between green and blue channel for all (7x7) pixels
13. **inside**: pixel is inside(1) or outside of the leaf(0)

These are colour based features, it is possible to choose additional features later, to make the prediction more accurate.
I created a script called *ImageDatabaseBuilder.py* which creates the .csv database from the images and the window locations.

First I was trying with a larger training set (450), but it was very slow since the script had to evaluate $450 * 2 * 100 * 100$ pixels. For this purpose, I created a multi-core version of this script ($ImageDatabaseBuilder\_MultiThread.py$), which generates the result so much faster. Since python does not support multi-core execution(it supports threading, but all threads are running on the same CPU core, which won't make the evaluation faster) by default, I had to use a library called *multiprocessing*.
This is the reason why I decided to have only 50 images in the training set, to make the database creation even faster. With 50 images, it is still enough to create a huge database, with $50 * 2 * 100 * 100$ lines.

	red_pixel	green_pixel	blue_pixel	mean_red	mean_green	mean_blue	std_red	std_green	std_blue	corr_rg	corr_rb	corr_gb	inside
0	80	118	41	77.755102	116.897959	38.897959	6.768868	6.415049	6.408683	0.967132	0.967152	0.996526	1
1	77	118	40	76.897959	116.612245	38.612245	6.112032	6.133935	6.127277	0.966801	0.966762	0.996200	1
2	78	119	41	76.285714	116.571429	38.571429	5.095016	5.345225	5.337583	0.976420	0.976317	0.994994	1
3	77	118	40	75.775510	116.346939	38.775510	5.304081	5.445654	5.234366	0.981270	0.979506	0.991476	1
4	77	118	40	75.265306	116.265306	38.836735	5.487557	5.487557	5.199478	1.000000	0.987149	0.987149	1

Fig. 5. Sample from the database

2.5 Stage 4: Training the model

For the model, I selected Random Forest Classifier due to its predicting power. The other model was SVM, but it took so much time to train because it is using an at least $O(n^2)$ algorithm.

For creating the model I used the script called $ImageResizeTrainer.py$. After scaling the dataset it trains the model with some hyperparameters.
Since in this problem evaluating the results requires to predict a test image, and check the results manually, and also the content of the database is very important (how do we choose the little windows inside the image), it is not so necessary to get the "best" model by tuning the hyperparameters. It can easily happen that we tune the parameters, and even if different metrics say that our model is better, at the end when we test it on unseen image data we will get poor results. The only hyperparameter that I changed from default 10 to 100 is the n_estimators which sets the number of trees in the forest.

After training the model I created a confusion matrix and used some metrics to check the performance of the model.

$$Confusion\ Matrix = \begin{vmatrix} 100087 & 2 \\ 6 & 99905 \end{vmatrix} \tag{1}$$

The model almost predicted every class from the test set, however as I mentioned before this does not mean that it will predict with the same accuracy for an unseen image.

	precision	recall	f1-score	support
0	1.00	1.00	1.00	100089
1	1.00	1.00	1.00	99911

It seems the model performed very well, maybe too well, it can happen that it has overfitted a little.
It is important to be able to save the model to the disk because then we don't need to retrain every time we want to use it in different files. For this purpose, I used a library called *Pickle*. This library can save any object into textual representation, and later it can load it too. This way I saved the classifier object, and the corresponding scaler objects too.

2.6 Stage 5: Predicting unseen images

For segmenting new, unseen images, the script called *ImageResizePredictor.py* can be used. In theory, if we want to segment an image which has a size of (2000x1500), then it means that - because we need to evaluate every pixel - it needs to loop over 3million pixels, which can take so much time. This is the reason why I had to shrink the image before applying model prediction, and at the end, enlarge it back to its original size. The ratio can be 0.25, or 0.5. In the case of 0.5, it will reduce the size by 0.5*0.5. The steps are the following:

1. Read input image
2. Resize image
3. Generate data from all pixels
4. Scale generated data
5. Put scaled data into a data frame, and insert it into the classifier
6. Create a mask from predictions
7. Resize mask to original image size
8. Apply erosion and closing operation on the image
9. Merge original image with the mask
10. Return output image

For trying to remove little black dots from the image (noise/wrong prediction), erosion morphological operation was used.
The execution time has been monitored by the script, thus we can use it for improving performance later. For the sake of simplicity I named the output images in the following way:

$$imageName + number\_of\_trees + currentTime + .jpg \tag{2}$$

2.7 Stage 6: Calculate accuracy of the segmented image

As I mentioned earlier it is not easy to measure accuracy just by viewing the results of the model. For this reason I created segmented images with the help of Photoshop's magic wand tool and compared my results to these images.

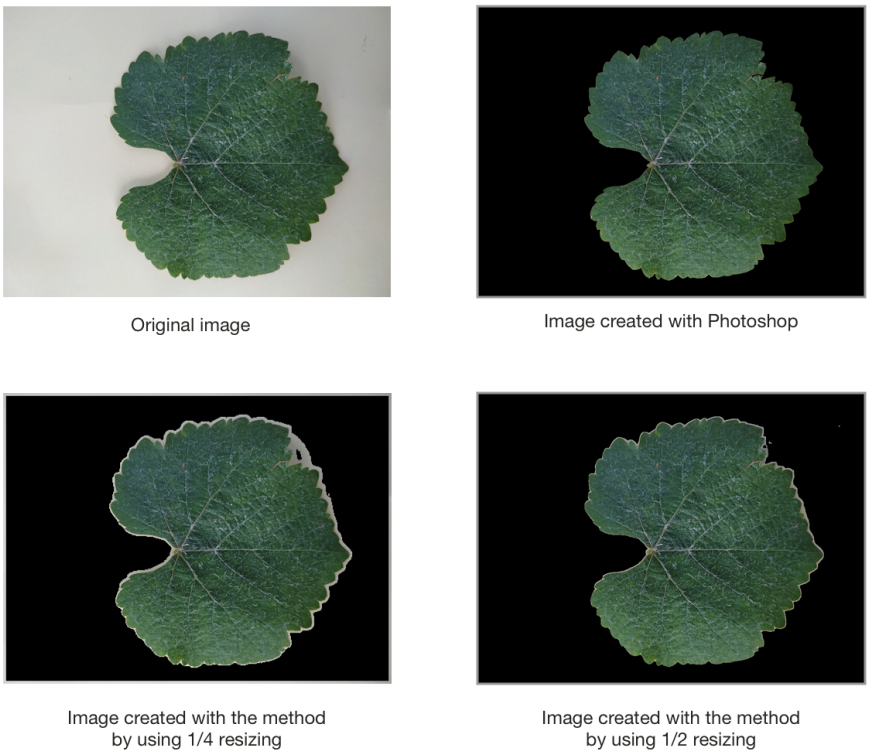

Fig. 6. Comparison of different methods

As it can be seen from the figure, the resizing with ratio 0.5 was the closest to the base (Photoshop). Changing the ratio was not an easy task, since decreasing it can speed up the process significantly however it will also provide a less accurate result. Selecting 0.5 was a sweet spot since it made the whole algorithm much faster, but still does not change too much to the full (ratio = 1) version.

For measuring the accuracy the script called *SegmentationAccuracyCalculator.py* can be used. It accepts the base image and the output image(s) as parameters, and creates a confusion matrix with the calculated accuracy.
For example running it for the image from the previous figure generates the following result:

```
ps_images/0134_ps_4.jpg
[ 1157001 ,   27736 ]
[    1813 , 1813450 ]
Accuracy: 0.9902%
```

Basically an accuracy of 99% can be considered as a very good result however the variance should be improved, because now huge differences can occur for different images.

2.8 Performance

Experiments were conducted with an Intel Core i7 8700 CPU which has 6/12 cores, and 32GB memory.
The script which is responsible for creating the image window locations, has no requirements, it does not matter what kind of CPU is running it.
For creating the database the situation is the following:
If we choose a training set of 450 images, the single core solution can take more than 1 hour to generate the whole database, which is just simply too much, it makes the testing process totally unusable. However running the same script for 50 images can reduce the execution time to 10 minutes which is acceptable. If we use the multi threaded version we can go under 20 and 5 minutes respectively.
Training the Random Forest Classifier takes around 1 minute, but training an SVM classifier was around 8-9 hours which makes it unusable for this problem. The reason behind this can be the number of observations, which is more than 1 million.
The execution speed of predicting a new image is based on the size of the image, which is in our case (2000x1500). This produces 3million iterations, which takes around 20 minutes to execute, however if we shrink the image for example by ratio = 0.5 = 1/2, then we can get an execution time of 20/1/4 = 5 minutes.

2.9 Environment

For development environment, Spyder and Jupyter Notebook were used, all scripts were written in Python.
For creating the base images for measuring the results, Adobe Photoshop has been used.

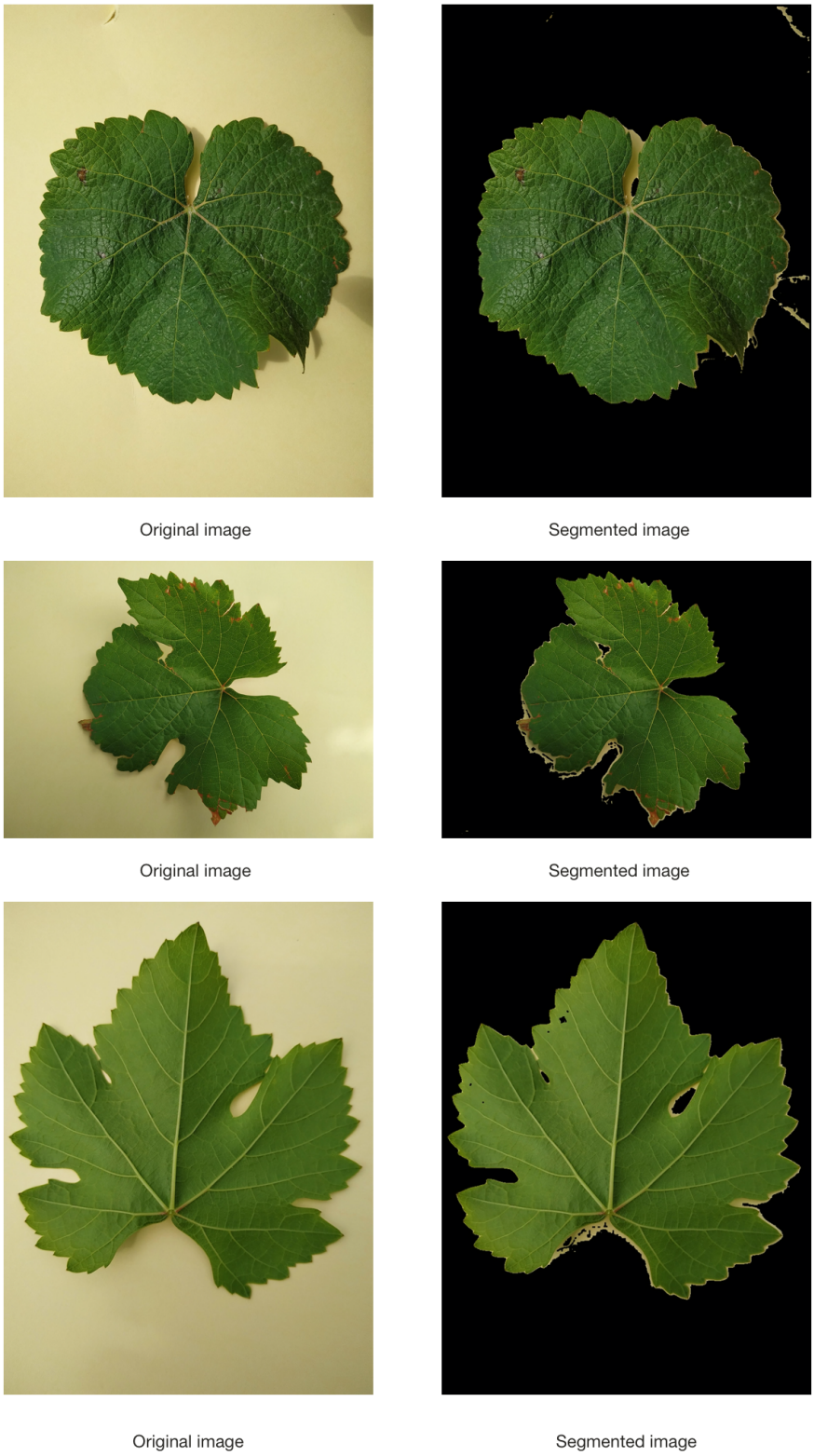

Fig. 7. Some input and output images

References

1. Jessica F. Lopes,Ana Paula A.C. Barbon,Giorgia Orlandi,Rosalba Calvini,Domenico P. Lo Fiego,Alessandro Ulrici,Sylvio Barbon: Dual Stage Image Analysis for a complex pattern classification task: Ham veining defect detection. In: Elsevier 2020. https://doi.org/10.1016/j.biosystemseng.2020.01.008
2. José Luis Seixas, Sylvio Barbon, Claudia Martins Siqueira, Ivan Frederico Lupiano Dias, André Giovanni Castaldin, Alan Salvany Felinto: Color energy as a seed descriptor for image segmentation with region growing algorithms on skin wound images In: IEEE 2014. https://doi.org/10.1109/HealthCom.2014.7001874
3. F. Schroff, A. Criminisi, A. Zisserman: Object Class Segmentation using Random Forests In: Microsoft Research 2008. https://doi.org/https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/Criminisi_bmvc2008.pdf
4. J. Kowski Rajan , D. Ferlin Deva Shahil , M.Tamil Elakkiya , Rajesh Prasad: Image Segmentation Using SVM Pixel Classification In: International Journal of Advanced Research in Basic Engineering Sciences and Technology (IJARBEST) Vol.3, Issue.3, March 2017 https://doi.org/https://www.google.com/url?sa=trct=jq=esrc=ssource=webcd=ved=2ahUKEwirsZC5u9fpAhUDmYsKHXQbDYQQFjABegQIBRACurl
5. José Luis Seixas Junior's slides, Esqueletização, em Português