Tarea 2 - Automatic Segmentation

CC5509: Reconocimiento de Patrones

Submitted To:
Mauricio Cerda Villablanca

Submitted By :
Alexandre Poupeau

June 10, 2019

Contents

1 Abstract

This report is about automatic segmentation of *Drosophilia* epidermis images. As we have ground truth labels, this is a classification task. The report focuses on the importance of balancing your data and the choice of the model.

2 Introduction

We have got 55 raw images of shape $(512, 512)$ and we wish to automatically segmentalize those images. In order to train models, we got the ground truth classification for each pixel in every single image. Therefore we also have at our disposal 55 label images of shape $(512, 512)$ with value of 0 for "separation class" and 255 for "in-cell class". In reality, we divided every ground truth image by 255 so that the classes are cleaner : 0 for "separation class" and 1 for "in-cell class".

First, we will see how we extracted the features for each pixel. Secondly, we will see how we constructed the dataset and then how we chose the best model for this automatic segmentation problem.

3 Experiments and implementation

In this part, we will explore the experiments and how we implemented it in Python.

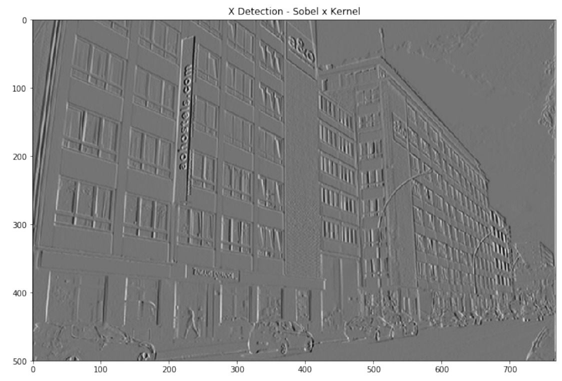
3.1 Convolution

As we want to do some classification, we need to extract features from the images. In this work, we will use three features for each pixel of each image : mean, gradient magnitude and the laplacian. Here is how these features are calculated. The sign $*$ symbolize convolution.

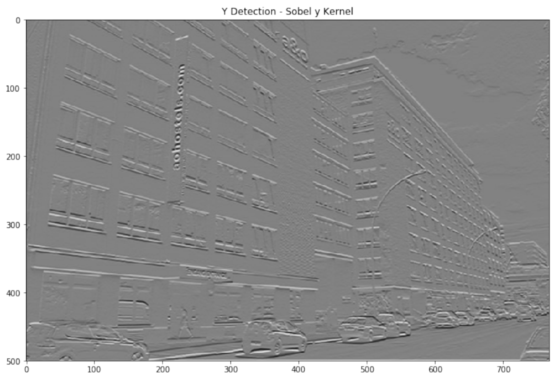
- Mean of a pixel p is $\mu_I = I * W_\mu$ with $W_\mu = \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}$ which basically just create the same image but just blurred.
- Gradient Magnitude is $\nabla I = \sqrt{D_x^2 + D_y^2}$ where $D_x = I * W_{sob,x}$ with $W_{sob,x} = \begin{bmatrix}-1 & 0 & 1\\-2 & 0 & 2\\-1 & 0 & 1\end{bmatrix}$ and $D_y = I * W_{sob,y}$ with $W_{sob,y} = \begin{bmatrix}-1 & -2 & -1\\0 & 0 & 0\\1 & 2 & 1\end{bmatrix}$. The sobel x kernel allows to detect vertical edges and the sobel y kernel horizontal ones.
- Laplacian is $\mathcal{L}_I = I * W_\mathcal{L}$ where $W_\mathcal{L} = \begin{bmatrix}-1 & -1 & -1\\-1 & 8 & -1\\-1 & -1 & -1\end{bmatrix}$. This kernel is particularly used to do edge detection.

(a) Original image

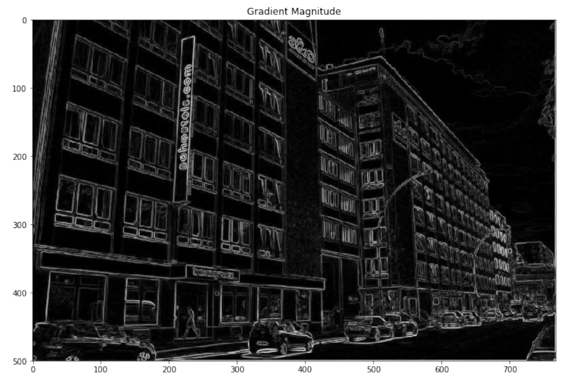
(b) Mean kernel



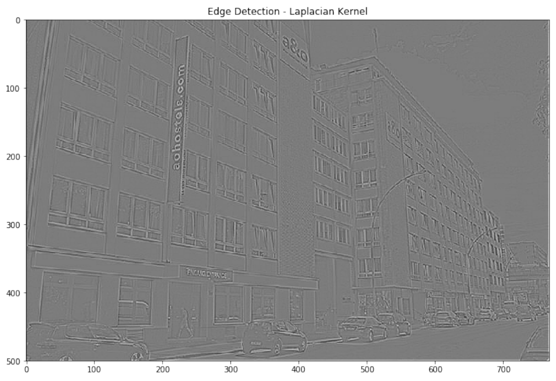
(c) Sobelx kernel



(d) Sobely kernel



(e) Gradient Magnitude



(f) Laplacian kernel

(g) Effect of different kernels

Before extracting features we wanted to make sure that our convolutions made sense so we just tried out with a random building image. We can notice that the laplacian does a good job to detect letters from the advertisement banner.

3.2 Dataset contruction

We firstly created two functions extractFeatures(img, flatten) and linkImgFeaturesAndClass(img, class_pixels_img).

- extractFeatures(img, flatten) : This first function extracts the three features for a given raw image. It returns three images of the same shape as the input, each image corresponds to a kernel convolution transformation : mean, gradient magnitude and laplacian. Thus the ouput has the shape (3, 512, 512). If flatten is True, the output has a shape (512×512, 3).
- linkImgFeaturesAndClass(img, class_pixels_img) : Returns array of shape (512×512, 4). This is simply the concatenation of the output of extractFeatures function with flatten==True and the label for each pixel.

Creating the dataset from this point was really easy. We just created another function in order to construct it named computeclfDataset(X, y, nb_max_img=55).

- computeclfDataset(X, y, nb_max_img=55) : It takes as input all raw and label images and returns the dataset of shape ($nb\_max\_img \times 512 \times 512$, 4). In reality, we used nb_max_img = 5 for all the experiments in this report. We had the occasion to compare with more data such as nb_max_img = 30 and the results do not change much.

3.3 Segmentation

3.3.1 Balancing

Now that we have our dataset, we can do automatic segmentation.

- displayPredAndTrue(img, class_pixels_img, trained_clf): This function takes a raw image as an input, its ground truth image and a pre-trained classification model. It displays two images, the prediction and the ground truth segmentation. This is a very useful function that we used each time we needed to get an idea of how different was the predicted and real segmentation for a given classification model.

We trained our first RamdomForest model and it seemed like it was too perfect, or at least, for a first attempt : we obtained a score of 82% accuracy. We tried to visualize how well the model was doing by using the display function. The prediction was all white. At first, we did not understand. Nevertheless, thinking about it a bit more we found the origin of the problem. The dataset is not balanced ! There are much more 1 class elements or "in-cell" elements than "separation" class elements. Thus the model trains much more on them and when it comes to predict it always predicts 1.

To be more precise, using only the 5 first images, we have at our disposal 1310720 pixels. The number of 0 class pixels is equal to 217063 and the number of 1 class pixels is equal to 1093657. So 1 class pixels represent 83% of all the samples and 0 class pixels only 17%. Therefore we needed to balance the dataset before training.

We created a function balance the dataset as we wish.

- equilibrate(dataset, margin): This function basically removes 1 elements until a given threshold. It takes the dataset that we have as an input and counts the number of 0 and 1 class elements. Then it balances the dataset such that the final number of 1 class elements n_1 and

0 class elements n_0 respect this condition : $n_1 = margin \times n_0$. If $margin = 1$, the dataset is completely balanced.

Nonetheless, we had a question : what margin should we use ? Our first intuition was $margin = 1.25$ because it balances the dataset and at the same time it lets 1 class elements still be more present than the 0 class elements. This seems like a good logical choice. With this margin, we managed to get this confusion matrix using the test dataset :

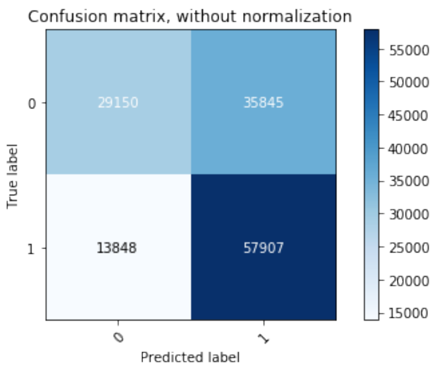


Figure 2: Confusion matrix - Model RandomForest 10 trees and max_depth = 2

The result is not that bad, we obtain an accuracy of 63.6%. However we decided to do a margin study. This way it would tell us more about how to evaluate a good model for this segmentation problem too.

3.3.2 Margin study

We quickly figured out that the most important elements in our predictions were the True Negative elements (or well-classified 0 class / "separation" class elements). We understood that we needed to focus on training a model capable of identifying them well. On the previous confusion matrix, we can clearly see that there are too much elements predicted as 1 but in reality are 0. The accuracy does not capture this phenomena well so we decided to use other metrics to evaluate better our models.

We firstly came out with the TNR or True Negative Rate which measures "for all the negative (0) labeled elements, the percentage of predicted as negative." This is an indicator of quantity. We also used the NPV Negative Positive Value which measures "for all negative predictions, the percentage of true negative". This is different than the first and it is a indicator of negative prediction quality / precision. A good model would in particular be a model with a high TNR and a high NPV. Thus, we also used a metric call the $Fscore$ applied to TNR and NPV. It can be computed as follows : $Fscore = \dfrac{2 \times TNR \times NPV}{TNR + NPV}$ and $Fscore \in [0, 1]$. In the future, we call the $Fscore$ the $Segscore$ or segmentation score because it is a much better indicator of a good automatic segmentation than the accuracy.

We computed a graph to visualize the evolution of those metrics as we increase the margin.

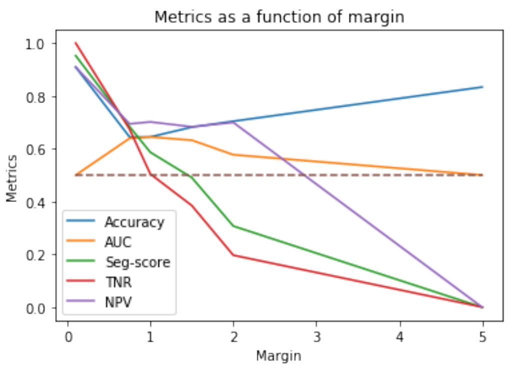
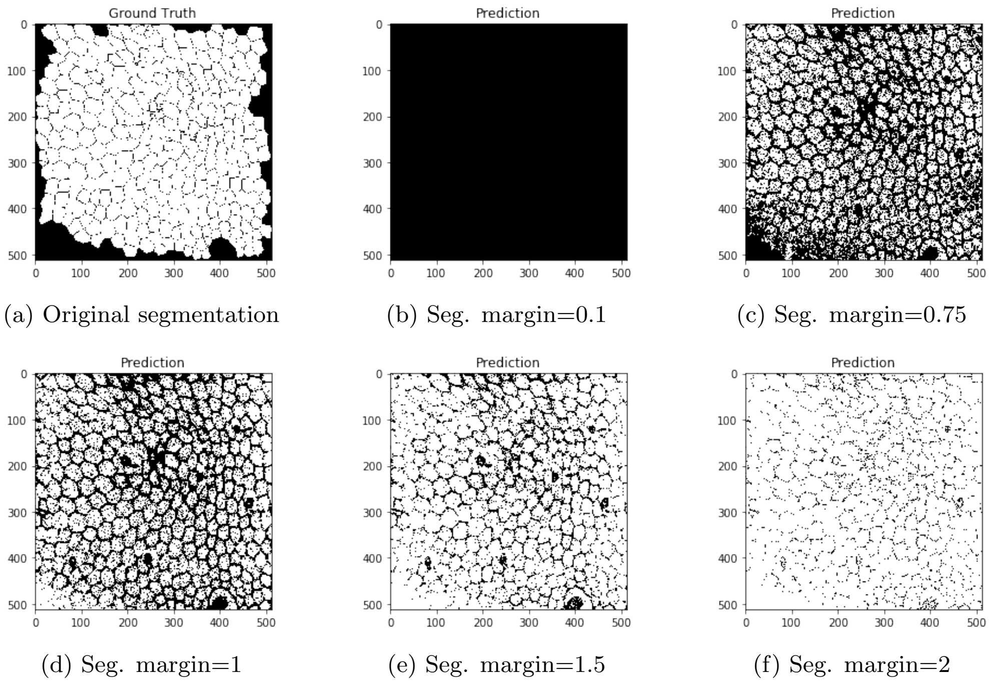


Figure 3: Metrics as a function of the margin

As we can see, the accuracy always increase when we increase the margin. It makes sense since in those cases the model only predicts 1. As there are more 1 than 0, the accuracy increase. However the TNR, NPV and $Fscore$ decrease ! Based on the graph a good margin would be between 1 and 1.5. In those ranges, the $Segscore$ is bigger than 0.5 and the AUC value is at its highest. We finally kept the margin value 1.25. We have to admit that we had a really good intuition on this ! Hence all the following experiments are done on a dataset with a margin = 1.25.

Here are the automatic segmentation obtained using different margins. We trained a basic RandomForest classification model with 10 trees and a max depth of 2 to get all predictions.



(a) Original segmentation (b) Seg. margin=0.1 (c) Seg. margin=0.75

(d) Seg. margin=1 (e) Seg. margin=1.5 (f) Seg. margin=2

Figure 4: Effect of different margins

3.4 Model selection

We now want to evaluate three different classifiers for the automatic segmentation problem. We chose KNeighbors, Multi-Layer Perceptron and RandomForest. We had to find the best hyperparameters for this specific problem for each model.

3.4.1 Hyperparameters tuning

We used the technique Grid Search to find the optimal hyperparameters of each model. This technique consists in testing all the different combinations of hyperparameters using a grid that you want and evaluating a metric (we used the *Segscore*) on the test set for each combination. You then keep the hyperparameter combination with the highest test score. In general, we evaluate each combination using cross-validation to avoid problem with the classes eventual order in the dataset. In our case, we used a 3-cross-validation. Applying this method takes a while but it is worthy.

With the KNeighbors classifier, we found out that using 12 neighbors was the best. Interestingly, the best distance to use was Minkowski's distance or distance L_1 (absolute value) and not the classic Euclidian distance L_2.

For the Multi-Layer Perceptron we tried different combinations of hidden layers and found out that the best is (4, 4, 4) which means three hidden layers with 4 neurons. We could not tried all the different combination for the parameters (because it is impossible !), but this one was providing good results. What surprised us is the best hyperparameter for the activation function is tanh and not relu like in most cases.

For the RandomForest, the best number of trees was 70 and a max depth of 10. More was not better.

3.4.2 Models evaluations

Here is a tabular that summarizes all the metrics obtained on the "hyperparameters-tuned models" :

Model	Seg-score	Accuracy	AUC	TNR	NPV
KNeighbors	0.657	0.705	0.763	0.63	0.681
MLP	0.604	0.692	0.731	0.52	0.708
RandomForest	0.617	0.701	0.744	0.54	0.718

We can clearly see that KNeighbors is doing a much better job than the two other models. Indeed, its Seg-score is the higher and in particular its TNR is much higher than on the other two models. This means that it predicts much less false positive which is great ! One can notice on this tabular why the accuracy alone is not a good indicator of the quality of the model for this specific segmentation problem : it is overall the same for each model.

Moreover, we can take a look at each model ROC curve :

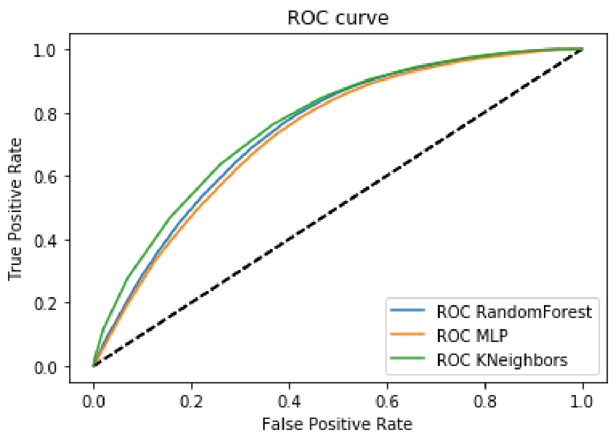
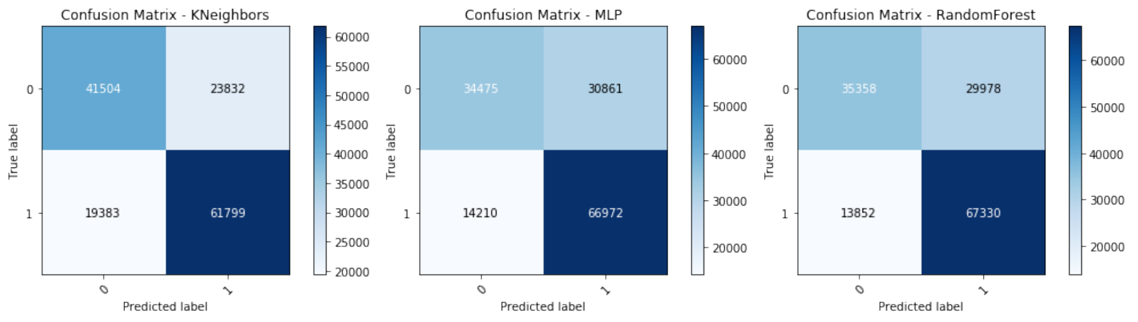


Figure 5: ROC curves

The area under the curve (AUC) is bigger for the KNeighbors classifiers so it means that it is overall a better model. We can also observe the confusion matrices :



(a) KNeighbors (b) MLP (c) RandomForest

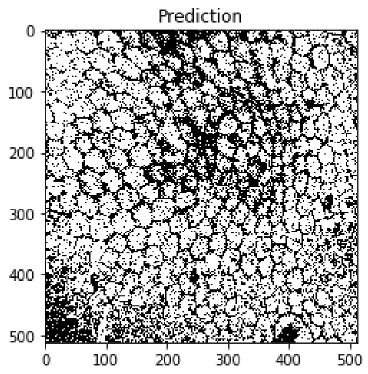
What is really astonishing is that both the RandomForest and MLP models present the same weakness : they struggle to detect the negative (0) elements. KNeighbors does a much better job on the point. These confusion matrices confirm what we analyzed earlier with the tabular. Another interesting element to notice is that RandomForest has the lowest number of false negative and thus its predictions present less noise.

Given all the arguments we have, we can conclude that the KNeighbors classifier is the best model for this problem.

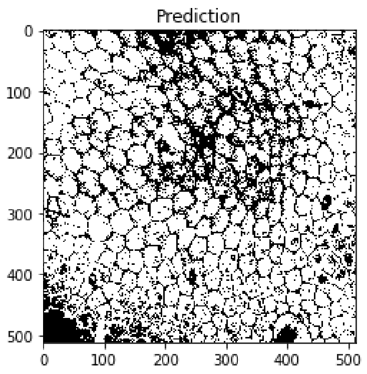
4 Conclusion

To conclude, never underestimate simple models. Always try them before using more complex ones. Here the KNeighbors classifier is simply doing a better job for this specific problem.

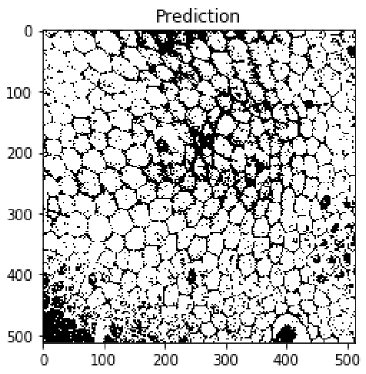
However, it is doing a better job if we just look at some specific metrics. One could argue that it is nonetheless the model with the most noise predictions or false negative (0) predictions (”nonin-cell" elements). From this point of view, Kneighbors is the worst model and RandomForest the best one. Here are the predictions for each "hyperparameters-tuned model" :



(a) KNeighbors



(b) MLP



(c) RandomForest

It is funny to see that, for a human, just seeing the predictions can be misleading. The KNeighbors provides, overall, the best performances, however with our human's eyes it really looks like it has too much noise and that it is the worst model.

An important fact we had the opportunity to see is that balancing your data is key. The margin used becomes kind of hyperparameter for the rest of your experiments. Choosing one is difficult and depends on the context of application. One can use a smaller margin (≤ 1) to obtain cleaner results (less noise).

My two last words are that on one hand, the solution of a problem is not always what we see nor we can imagine and on the other hand that the real "best" automatic segmentation model depends on the context. Maybe a company would rather prefer the RandomForest model for its clean separation / absence of noise over the KNeighbors model. Here we lack context to really perfectly conclude.