KiBiOCR

Mathieu `RustyCrowbar' Corre
Thibaud `zehir' Michaud
Valentin `toogy' Iovene
Pierre `Grimpow' Gorjux

11 décembre 2013

Table des matières

Introduction

This document will present the current state of our project, KiBiOCR, as part of EPITA's syllabus. KiBiOCR's name speaks for itself. It is an OCR (Optical Character Recognition) program. Its goal is then to recognize characters in a given image.

You will see that our project is breaked up in 5 very distinct parts :

— Lenna : preprocessing part of the program. Lenna modifies user's input image to help other parts of the program to do their job.

— Freddy : `he' is in charge of the segmentation part, which is identifying images, paragraphs, words and characters positions.

— Anna : this is our artificial neural network (ANN). `She' identifies characters segmented by Freddy.

— *(Le petit)* Robert : this is our dictionary. `He' detects the document language and errors made by Anna/Freddy in the identification process and then try to find the nearest words.

— Guy : our user interface. This is the way our program and the user will be able to interact.

We gave surnames to the parts of our program because we like private jokes but mainly because shortnames are really convenient when we talk about our project.

1 | Lenna -- *Preprocessing*

Before trying to detect paragraphs, lines and characters in the image, and then recognizing the characters, we have to "clean" the image. In this chapter we will describe the different algorithm we implemented in this purpose.

1.1 Binarization

This is the first algorithm we wrote, because it was easy, essential and a good start to learn the library's basic functions. We first implemented the most naive algorithm you could think of : computing each pixel's luminosity by averaging its RGB components, and applying a thresholding. Every pixel below 127 was considered white and every pixel above 128 was considered black. Of course this was not the definitive algorithm.

We could have enhanced it by setting the threshold to the average luminosity, but it wouldn't have been much better. We instead looked for a better algorithm and found the Otsu's method to be fitting exactly our purposes. Moreover it is pretty straightforward to implement. Basically Otsu's method is just a way of finding the ideal threshold. The ideal threshold is defined as being the one which minimize the variance between the two classes formed by that threshold. Therefore it has to try every possible threshold and compute every time the variance of each class. Thanksfully a little mathematical trick allows us to significantly reduce the time complexity by maximizing the between-class variance instead of minimizing the within-class variance.

1.2 Noise reduction

We tried two method to reduce the noise in the image : gaussian blur and median filter. We were unsatisfied with both because letters were less readable after the filter, and they hardly reduced the noise. We will have to search better method for noise reduction, but for now we will simply use an algorithm that removes every black pixel surounded by four white pixels. Even if this is a much simpler algorithm, letter shapes are saved and some noise is removed.

1.3 Rotation

It is very difficult to detect text zones in the image if they are not straight. Therefore we will try to detect the angle of the image, and then rotate it.

The first step before rotating the image is to compute the size of the new image. As for the rest of the algorithm we use rotation matrix, but only on the corners. From the minimum and maximum horizontally and vertically, we can deduce the width and height of the rotated image.

A naive implementation starts by going through the original image, and transposing every pixel (x, y) into the corresponding one in the rotated image. We do that by computing the dot product between the pixel's coordinates and the rotation matrix of the given angle. We want to turn the image with the center of the image as the origin, so we have to add half of the width (respectively height) of the image to x (respectively y)

before actually computing the rotated point. It turns out to be a pretty bad algorithm, because not every pixel of the rotated image will have an antecedent in the original image. This is due to the fact that some pixels in the original image have the same corresponding rotated pixel, because of the rounding made to get integer coordinates. This creates an aliasing effect.

Instead, we need to make the reverse operation : take every pixel from the new empty image, and find its corresponding pixel in the original image. Therefore, the only thing to do is to apply a rotation matrix with the opposite angle.

Because of the imprecision of the integer coordinates, the rotated image is still not perfect. A potential improvement will be to use an algorithm that takes this imprecision into account, like the bilinear interpolation.

1.3.1 Angle detection

Hough transform

The Hough transform is a general method for finding lines and ellipses in an image. It is often used because once the mathematical principle is understood, it is not that hard to implement, and it has a much lower time complexity and better results than some other methods like Fourier transform. For our purposes, we only need to handle the most simple case : line detection in a binary image.

The Hough transform relies on a very particular representation of lines of the plane : instead of being as the two parameters a and b in the classical equation $y = ax + b$, it is described as the couple (r, θ) where r is the distance of the line from the origin and θ is the angle of this distance from the absciss.

How is this representation useful ? It comes from the fact that, given a point of coordinates (x, y) and the angle θ of a line going through this point, we can express very simply the distance r of the line from the origin. The equation is as follows :

$$r = x\cos\theta + y\sin\theta$$

So for a point of coordinates (x, y) we can plot the graph of r in function of θ. It is easy to guess from the equation that it will be a sinusoid. This sinusoid represents the set of lines going through this particular point. Now if we store this graph on an accumulator, and add to this accumulator the sinusoid of every other points, here is what happens :

Clearly there is a point where all sinusoids cross. The coordinates of this point represents the values r and θ of the line going through every point at once.

More preprocessing

There is still one thing to fix before applying the Hough transform to our scanned image : A text doesn't appears as a straight line. We tried to apply the Hough transform directly to a binarized document, but most of the time it would find either the diagonal of the document (since the diagonal goes through more black points than an horizontal or a vertical line), or the angle of a prominent bookbinding or document border. After a few researches, we came with our own solution : we first find "blocks" of pixels, defined as being a set of conjoint pixels. We remove every pixel of this block and replace them by a single pixel at the center of the block. There are two purposes to this algorithm : replacing every character by its center to make text lines apparent, and reducing the importance of large blocks of pixels as bookbindings and images in the Hough transform. Here is a sample output :

Figure 1.1 – input image

Figure 1.2 – center of each block

Figure 1.3 – line found by the Hough transform

This method has a major flaw that we wish to correct before the end of the project : when the noise isn't reduced enough, every separate pixel in the "noisy zone" is considered a block and gets a huge role in the Hough transform, potentially leading the angle detection to fail completely. Every further step of the OCR relies on a perfectly straight image, and it is therefore very important that we correct this.

1.4 TODO

If Lenna is almost finished, some improvements still remain :

— Rotation : it is divided in two parts : angle detection, which is done and the real rotation. The current algorithm gives very poor results. We are going to implement bicubic interpolation which is a standard in many image editing programs (including Adobe Photoshop).

— Noise reduction improvement : our noise reduction algorithm is not strong enough on some images. Sometimes it fails and Freddy cannot work with Lenna's output.
— Hough transform preprocess The method we use has a major flaw that we wish to correct before the end of the project : when the noise isn't reduced enough, every separate pixel in the "noisy zone" is considered a block and gets a huge role in the Hough transform, potentially leading the angle detection to fail completely. Every further step of the OCR relies on a perfectly straight image, and it is therefore very important that we correct this.
— Image scaling : Guy may need an image scaling algorithm to display thumbnails of user's image.

2 | Freddy -- *Segmentation*

Freddy is our segmentation module. It uses OCSFML. To detect lines as today, we first assume that all the previous preprocessing steps were done perfectly, which means no artefacts, good rotation, binarization... For now, we can only get text, in one column, and no images. We will work on those situations, and Freddy will be working as expected on time.

Freddy est notre module de segmentation, il utilise OCSFML.

2.1 L'ancienne méthode, ses points faibles

Au moment de la première soutenance, nous utilisions un algorithme très simple, et moyennement efficace : nous faisions un histogramme horizontal, un tableau de taille la hauteur en pixels de l´image. Nous le remplissions avec le nombre de pixels noirs à la ligne correspondante. Cette opération nous permettait de savoir à quel moment une nouvelle ligne ou un nouveau paragraphe apparaissent.
Une fois ce découpage effectué, il nous suffisait de répéter cette opération horizontalement sur chacune des lignes repérées, afin de détecter les mots et caractères composant chacune des lignes.
Cependant, nous avons été confrontés à plusieurs problèmes, qui nous ont amenés à chercher une autre méthode plus performante, que nous verrons en détail dans la prochaine section. En effet, avec cette méthode, les caractères avaient tendance à être coupés, ou bien collés à d´autres. Il en était de même pour les lignes, qui avaient tendance à fusionner. Cette

technique nécessitait également d´avoir une image sortie du prétraitement parfaite, la moindre impureté ou la moindre ligne, image ruinant totalement le résultat de l´algorithme.

2.2 Le RLSA

Pour détecter les paragraphes, lignes, mots ou les caractères, nous utilisons donc maintenant l'algorithme Run Lenght Smooth Algorithm (RLSA) modifié par nos soins, pour couvrir au mieux nos besoins. Le principe est simple : on regarde horizontalement et verticalement quels sont les pixels noirs adjacents, c´est à dire quels sont les pixels noirs séparés de moins de n pixels noirs. Il en résultera donc deux matrices de bouléens aux dimensions de l´image. Dans ces matrices représentant la couleur noire ou blanche des pixels de l´image, nous aurons changé tous les pixels blancs entre deux pixels noirs adjacents. Les images représentées par ces matrices ressemblent à l´image d´entrée, mais comme si l´encre avait bavée horizontalement ou verticalement.
Nous appliquons ensuite un opérateur binaire logique entre les bits des deux matrices aux mêmes coordonées, afin de former une nouvelle matrice. Cette nouvelle matrice, toujours aux dimensions de l´image d´origine, représente l´image résultat, qui sera utilisée pour la suite des traitements. Dans cette image, les lettres, mots, lignes ou paragraphes se retrouveront rassemblés en blocs noirs, suivant le nombre de pixels adjacents que l´on aura considéré.
Cette transformation nous permet ensuite, via un autre algorithme que nous détaillerons plus tard, de récupérer les coordonées et dimensions des zones noircies.

2.2.1 Optimisations

Pour sélectionner les caractères, mots, lignes ou paragraphes, nous devons modifier la constante représentant le nombre de pixels adjacents à fusionner. Cependant, cette modification seule ne nous permet pas de sélectionner efficacement une partie précise de l´image. En effet, les caractères sont globalement carrés, tandis que les mots et lignes sont assez horizontaux. Une seule constante ne suffira donc

2.3 Bounding boxes

2.4 Line and character detection

2.4.1 Making horizontal/vertical histogram

To detect lines, we need to make histograms, containing the number of black pixels in every line/column of line of pixel of the provided image.We put everything in lists.

2.4.2 Delimit lines/characters

When the number of black pixel changes a lot, we decide to create or to end a line/to create or end a character. We put it in lists.

2.4.3 Delimit lines/characters

When the number of black pixel changes a lot, we decide to create or to end a line/to create or end a character. We put it in lists

3 | Anna -- *Identification*

Après avoir pré-traité l'image et trouvé les paragraphes, les lignes et les caractères dans le document, il faut reconnaître ces caractères. Pour ça, nous utilisons un des algorithmes de machine learning les plus répandu : le réseau de neurone feed-forward. Nous allons ici détailler ce que nous avons appris sur cet algorithme, et comment nous l'avons implémenté.

3.1 Machine learning

Le machine learning est un champ de recherche dont le but est de trouver des algorithmes capables de s'améliorer sur une tâche particulière avec l'expérience. Souvent, ce que l'on appelle l'expérience consiste en un ensemble d'exemples entrée-sortie correspondant à ce que l'on veut apprendre. Dans notre cas, la tâche du réseau de neurones artificiel sera d'associer à une image de caractère un numéro (par exemple son code ascii), et les exemples sont des paires (image de caractère, numero du caractère).

3.2 La régression linéaire à une seule variable

Ramenons nous à un problème plus simple : on a des exemples de la forme (x, y) où x et y sont des nombres, et notre but est de trouver la droite correspondant le mieux à ces exemples.

La ligne sera de la forme $h_w(x) = w_1x+w_0$. La première chose à faire est de déterminer une fonction d'erreur pour évaluer les différentes droites possibles. Il se trouve que l'erreur quadratique possède des propriétés intéressantes (comme le fait d'être toujours positive et dérivable). Cette fonction d'erreur s'écrite comme suit :
$Err(h_w) = \sum\limits_{j=1}^{N} (y_j - h_w(x_j))^2$
où (x_j, y_j) est le j-ème exemple et $h_w(x_j)$ est l'image du point x par la droite $h_w : x \to w_1x + w_0$. Notre but est maintenant de minimiser cette fonction. Imaginons que cette fonction a une forme de bol, et que nous commencions à un point (w_1, w_0) aléatoire de cette fonction (c'est à dire, avec une droite du plan aléatoire). Pour trouver le minimum de cette fonction, il suffit de suivre la pente vers le bas. Mathématiquement parlant, cela signifie qu'il faut prendre le point actuel, et lui soustraire le gradient de la fonction d'erreur. Ce gradient est multiplié par une constante appelée taux d'apprentissage, noté α, pour réguler la taille du pas. Un faible taux d'apprentissage signifie une convergence lente, mais avec un taux trop haut, on risque de passer par dessus le minimum et de ne pas converger. On modifie donc le point courant avec la formule :

$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Err(w)$

3.3 La régression linéaire à plusieurs variables

L'exstension de cet algorithme à plusieurs variables se fait assez naturellement. Notons $\mathbf{x} = (x_0, x_1, \ldots, x_n)$ r. La fonction d'estimation devient alors $h_w(\mathbf{x}) = w_nx_n + \cdots + w_1x_1 + w_0x_0$.

3.4 La regression logistique

Ce dont nous avons besoin pour reconnaître les caractères est un algorithme de classification. Nous n'avons besoin que de quelques modifications pour faire de l'algorithme précédent un classificateur binaire : nous gardons la fonction d'estimation $h_w(x)$, mais nous passons ensuite le résultat de cette fonction dans une autre fonction dont la sortie se trouve dans $[0;1]$. La fonction de pas en est une : sa sortie vaut 0 si son entrée est négative, et 1 sinon. Mais cette fonction n'est pas dérivable, et nous verrons plus tard que cela pose problème. On lui préfère donc souvent la fonction logistique (aussi appelée sigmoïde)

Son équation est $g(x) = \frac{1}{1+e^{-x}}$, et sa dérivée est $g(x)(1 - g(x))$. En appliquant le même algorithme que pour la régression linéaire, la formule pour mettre à jour le point actuel devient :

$$w_i \leftarrow w_i + \alpha(y - h_w(\mathbf{x}))h_w(\mathbf{x})(1 - h_w(\mathbf{x}))x_i$$

3.5 Perceptron

La régression logistique est la brique de base du réseau de neurones. C'est ce que nous appelons un neurone, du fait de la ressemblance avec le fonctionnement d'un neurone biologique. Maintenant complexifions un peu le problème : nous devons ranger les entrées parmi plusieurs classes possibles. Supposons par exemple que les entrées puissent appartenir à trois classes possibles.

La solution est de passer l'entrée à trois neurones différents. Cela produira un vecteur de sortie plutot qu'une simple valeur, et idéalement, ce vecteur convergera vers quelque chose comme (1, 0, 0), (0, 1, 0) ou (0, 0, 1), ces trois vecteurs représentant évidemment les trois classes possibles. Cet algorithme est appelé un réseau de neurones feed-formard à une couche, ou perceptron.

3.6 Réseau de neurones feed-forward multi-layer

3.6.1 Principe

Un réseau de neurone feed-forward multi-couches est un perceptron dont le vecteur de sortie sera l'entrée d'un autre percptron, et ansi de suite jusqu'à un perceptron de sortie. Chaque perceptron est appelée une ``couche'' du réseau de neurone. Les couches qui ne sont ni la couche de sortie ni la couche d'entrée sont appelées couches cachées. Le gros avantage d'un réseau de neurones multi-couches par rapport au perceptron est qu'il peu potentiellement calculer n'importe que fonction continue avec seulement une seule couche cachée et avec le bon nombre de neurones dans chaque couche. C'est le résultat énoncé par le théorème universel d'approximation.

3.6.2 Backpropagation

Pour l'algorithme d'apprentissage, la différence majeure est la fonction dérivée de la fonction d'erreur. Il n'est pas nécessaire ici d'expliciter cette dérivée, mais il est à noter que le calcul de la dérivé des poids d'une couche l dépendra de calculs fait à la couche l+1. Ainsi la mise à jour des poids doit se faire de la dernière couche à la première, d'où le nom de ``backpropagation''.

3.7 Optimisations

The only optimization we implemented at this point is the momentum. In the gradient descent process, we can consider that the current point has some inertia by adding a certain percentage of the gradient of the last iteration in the update rule. There are two purposes to this : converging faster, and avoiding local minimum.

Deux optimisations ont étés implémentées pour augmenter la vitesse de convergence du réseau de neurones et éviter les minimums locaux :

— Le moment : appelons Δ_n le nombre que l'on ajoute à un poids w_i, j à une étape n de l'apprentissage. A l'étape n+1, on ajoutera en plus du gradient de la fonction d'erreur un certain pourcentage de Δ_n. Cela permet à la fois d'éviter les minimums locaux, car le point possède une sorte d'` `inertie'', et à la fois de converger plus vite.

— Taux d'apprentissage non constant : une technique courante pour augmenter la précision de la convergence est de modifier le taux d'apprentissage au cours du temps. Il est grand au départ pour chercher un minimum global avec peu de précision, puis il diminue pour cherger plus précisément ce minimum.

4 | Robert -- *Spell checking*

Robert is a module, which can detect the language used in a given text, and eventually indicate to the user which word is wrong, and give a selection of possible corrections.

4.1 Generation of Dictionnaries

A dictionnary is a file which contains a lot of words of a specific language we can find in a text.
We didn't create them ; it's too long and we can easily find some complete dictionnaries on Internet (150 000 words in English, 700 000 words in French, for example).
But actually, searching a word in a file is'nt a really good idea and may take a long time. That's why before doing anything else, we have to generate a data structure which will store the whole words of the dictionnary file. This one is a hashtable, because the time taken to search a common word is constant.
So, when loading Robert, OCaml will generate the whole dictionnaries needed by creating hashtables and put them in a list.

But there is still a problem : the time taked to create a hashtable from a dictionnary is around 1 second. If we want to accept many differents languages, the loading of the module would be too long. The solution is the Marshall module : it let us serialize an object into a bytes array, and save it into files. The opposite procedure is also possible. So OCaml has juste to read the byte array and deserialize it to load a hashtable. This technic need more memory but is really faster (around 1000% faster)

4.2 Detecting language in a given Text

The algorithm used is pretty simple : it will analyse the first X words (X is an empirical number, set as 200). For each language, it will count the number of words recognised ; The final language is the one which has the mosts of right matches.

4.3 Detection of wrong words

It's quite the same thing. The algorithm make a list, and starts to read the whole text. Each time a word is wrong, it is added to the list.

4.4 Selection of possible corrections for a wrong word

Generally, when a word is wrong, it's because the user writted it thinking about a right word, with the same pronunciation.
That's why the Soundex algorithm has been used in our project. It transforms a given string into its phonetic equivalent. So, we created a new type of dictionnary : a "phonetic hashtable" : we can find the word by searching the phonetic string. So, if we're searching a phonetic string we can find the whole words which have the same phonetic string. We have a first list with a lot of possible corrections.
But we won't give for examples 42 possibilies of correction to a wrong word : the user would be lost ; that's why we'll give to the user only 5 possibilities.

The possibilities will be sorted using another algorithm, called Levenshtein. Levenshtein calculates the number of modifications we have to make in order to change a word A into a word B. So, we'll only have the 5 words which are the closest to the wrong word.

5 | Guy -- *Graphical User Interface*

5.1 A quick overview of Guy

Guy is our GUI. It uses LablGTK and OCSFML, and will probably be using html to render the text output. It has several roles that we will describe right now. It is today composed of one window, to choose the file. It will be, for the second soutenance, composed of two or three more windows, to show the flow of the processing and the time taken by every main algorithm that we are using.

5.2 A user-friendly interface

5.2.1 Helping the user to choose the image to process

Guy has a built-in file browser, which is very intuitive, and easy to use, to select the file you want to use. Once the file has been selected, it's path will be displayed in an editable text box. However, the user can directly type in the provided text box the path of his desired file. Of course, it is still possible to use our program via the command-lines.

5.2.2 Displaying the usefull informations

We will display a miniature image of the image (assuming that it really is an image) that the user choosed.

5.2.3 Showing the final result to the user

Guy will be charged to show to the user the final work of our OCR. Indeed, we will present the image the user gave us, and the text we could detect, with the eventual orthograph corrections using Robert. It needs to be well organised, so that in case of a long document, the user could find where everything in the output text is located in his image, and vice-versa.

5.3 Guy is the foreman

One of the role of our GUI is to make every module work one after the other, calling them one after the other, with the right parameters. It will be possible to include one or several progress bars to inform the user of the flow of the processing.

6 | Website

6.1 Python

This year, we decided to switch from PHP to Python. We think that Python is a stronger and more relevant language and we wanted to learn its syntax and its specifics. So, here we are.

Plus, if we want to put our OCR online so that people can use it via a web interface, it would me so much easier to do it with Python than with PHP. Python can easily call other programs and execute commands on the server in a very secure way. It is more complicated with PHP.

6.2 Django

There is a very nice Python web framework out there on the web that is called Django. Working with Django is quick : it is made for rapid development and for perfectionists (we are).

6.3 Nginx

We wanted to use something else than a basic Apache server so we decided to go with Nginx, a quick, clean and easy to configure web server.

6.4 Github

Our code is hosted on the now very famous website (and git server) GitHub. For the moment, it is located in a private repository (for obvious reasons) but we plan to go opensource once the last soutenance will be passed !

Conclusion

Our project is moving forward very well. Our team is organized, we are all working with passion and seriousness. We often schedule meetings and establish task lists so that everyone is aware of what has to be done and can pick up something he is interested in.

Some very tough tasks are waiting for us and we hope we will be able to show up at the final with a fresh and working piece of art.