Improvement and expansion of a decoder module for an OCR system

Arnau Vázquez Junyent

Abstract— OCR systems, short for Optical Character Recognition, are becoming increasingly popular due to the increase in the digitalization of everything. Books, textbooks, magazines and several other paper-based documents are being transformed into an electronic version to be manipulated by a computer. As well, instant translation by image is becoming a reality with the booming technology of smartphones. Nonetheless, OCR systems are still not perfect. The real world contains a lot of extra information and noise that is very difficult for a current OCR system to clean completely, as well as the immensity of variables that take place in handwritten characters and paper-based documents. This project is meant to further improve a decoding module that uses a graph-based algorithm to predict optimal words, and attempts to increase its overall accuracy by using synthetic dataset generation for testing and applying improvements to the base algorithm.

Index Terms— Probabilistic histogram, binary histogram, bigram, OCR, decoder, lexicon, PHOC, graph, synthetic generation.

1 Introduction

THERE are several approaches for extracting words from images, but most of them have in common the following steps.

Firstly, there is pre-processing. For the images to be treated by the recognition system, images have to be pre-processed, so the noise and impurities of the image are reduced.

Then there is text recognition. This module's task is to extract the feature of the characters and determine with a probability that the input data is a certain character. Feature extraction can only output what it reads relying on the "knowledge" it has acquired in the training phases, so it cannot detect incoherences in the choices it makes. That is where the part in which this paper focuses on comes in.

So, the last step is post-processing. To increase accuracy, there are different strategies that can be followed. Some of them include limiting the words that can be detected by a lexicon, for instance, only allowing English words to be decoded. This, however, makes it impossible for words that are not contemplated in the lexicon such as proper nouns. So, this project will be focusing on this part of an OCR system.

To be more specific, this project will follow the steps of a previous iteration [1] and will try to improve its overall decoding accuracy. In the following sections I will explain in more detail the base which this project has departed from, as well as the adaptations made for it to fit our current requirements. For now, I will explain the objectives of this project, as well as the necessary changes that have been made.

The main objective of this project is to improve the results obtained by the previous iteration of this project, doing so by improving the existing algorithm and developing new approaches that hadn't been considered. To be more specific, the main hypothesis proposed was that by using a 10-level histogram instead of a 5-level one, like in [1], we would obtain better results. This will be explained in more detail in the development section. Also, it was proposed that there were certain methods that could be applied to the algorithm to improve the results even further. So, the earlier objectives of this project included:

1. The adaptation of the previous iteration to test the proposed hypothesis (using 10-level histograms).
2. Analysis and understanding of the problem.
3. Development of improvements to the algorithm.

During the developlent of the project, the situation brought by the SARS-COV-2 pandemic made it impossible for me to access the dataset that was supposed to be used to test the algorithm. So, in order resolve this big issue, the objectives had to be reconsidered to the following:

1. Generation of a synthetic dataset to test the changes to the algorithm.
2. Develop the improvements.
3. Run tests and analyze the results obtained using the generated datasets, and applying the modifications developed.

In the different sections of this project I will focus on explaining the different aspects that comprise the whole project. Firstly, and to put in situation, I will briefly ex-

- *E-mail de contacte: arnau.vazquezj@e-campus.uab.cat*
- *Menció realitzada: Computació*
- *Treball tutoritzat per: Ernest Valveny (Ciencies Computació)*
- *Curs 2019/20*

plain the previous iteration of the project, what it acomplished and expose the hypothesis proposed to improve it. Then, I will explain the methodology of the development and the development itself. Finally, I will show the results obtained, analyze them and summarize the overall outcome of the project in the conclusion section.

2 STATE OF THE ART

This section is meant to give context to the more technical part of the project. I will be explaining from what base this project started from, and briefly mention some other techniques used in OCR post-processing.

The previous iteration of this project [1] focused on the development of a word decoder that was able to output a word using a graph structure and optimal track algorithms.

The input of the system was a representation of words called "pyramidal histogram of characters" or PHOC [2]. This representation consists of a probabilistic histogram of 604 dimensions separated by levels and with an addition of bigrams (combination of 2 characters). This histogram contains 14 sub-histograms of 36 dimensions which represent the letters of the English alphabet and the numbers 0 to 9 ((26 + 10)*14 = 504 dimensions).

These representations can be obtained differently depending on the format. Binary histograms are generated by using a digital word and arranging the values of the letters in the aforementioned structure. As this representation uses digital words it has no noise or representation errors. The only errors that can be caused come from the PHOC representation itself. Decimal histograms are generated by the output of the feature extraction part of the OCR and I have had no access to this generation method. Nontheless, I have been able to work with a previously generated dataset. Finally, I have developed 2 methods to generate synthetic probabilistic histograms, which I will explain in the development section.

In addition to the 504 dimensions, we add an extra 100 dimensions with the most common bigrams found in the word dataset. The 14 sub-histograms are distributed in 4 levels, where each one of them represents a way of partitioning the word. The first level contains 2 sub-histograms (word divided in 2 parts), where the first one represents the letters on the first half and the second one represents the other half. The same is done for the following levels but with 3, 4 and 5 divisions of the word.

This representation is treated so as to have a graph structure with all the costs and possible arrangements of the letters in the word. After that, the optimal word is calculated with an optimal path algorithm.

Some key concepts to be considered of the methodology are:

1. Letters only appear once for each 36-dim histogram.
2. The letters are not ordered within the 36-dim histogram. Instead we use the bigrams to determine the most probable combination.
3. We choose the level of the histogram according to the number of letters detected (this is an important concept that will be worked on in the improvement section).
4. One of the optimizations performed in [1] was to add a "letter acceptance threshold". This threshold determines whether a letter will be considered for the graph or not. After several tests, it was determined that a threshold of 0.4 produced the best results [1]. This particular concept is very important to carry on, since I will use it in further sections.

As for other methodologies used in the field of post-processing, I have found that there is a great deal of variety. To name a few, [4] uses supervised classification algorithms to detect errors in the read letters and then suggest optimal correction for each error, and [3] uses an algorithm based on Google's online spelling suggestion to correct errors.

3 METHODOLOGY

3.1 Software development methodology

For this project, I have followed what I found to be the most similar to the way I work, which is PSP (Personal Software Process).
PSP is intended to help software engineers grow and improve their performance by tracing the development progress and comparing the predicted results with the actual results. This helps the developer acquire experience and improve in the following aspects:

1. Being able to understand the necessities of the project and making a better initial approach.
2. Knowing the development's limitations and not committing to unreasonable results.
3. Learning to manage the quality and making the result as good as possible within our limitations.
4. Reducing the number of defects by making a more solid base and structuring the code in a more stable manner.

All in all, it is a methodology that makes an engineer grow and improve their performance through experience.
As far as software development paradigm goes, the one I have used is a mix between the spiral and cascade models.

For the project as a whole, I have followed the spiral model, I have planned, developed and tested each of the parts separately, but within each part I have followed what would be more similar to a cascade model, because for each part I needed to follow a sequence and each progress point relied on the previous ones.

To give some examples as to how I have applied this methodology, I have used the experience acquired in the first steps of the project to plan more realistically in terms

of deadlines, and to what degree of development I would be able to achive. As a more technical application, I have attempted to organize the code in a way that would allow me to modify it more clearly, to identify problems faster and to have several iterations of the same code to track my progress or revisit older parts of the code in case I needed them. This, as well as the weekly meetings with my tutor to plan and evaluate the progress, has allowed me to work organized and better prepared for changes and suggestions.

3.2 Development tools

The tools that I have used are:

1. Matlab R2017b: Used for dataset generation.
2. Python 3.7: Used to develop all the code.
3. Python libraries used:
    a. Numpy
    b. Matplotlib
    c. Random
4. Pycharm Community Edition 2019: Main editor.
5. JabRef: Used to manage references.

4 DEVELOPMENT

In this section I will explain thoroughly the steps in the three main parts of this project's development, as well as explaining the decisions I have made throughout it.

The development is separated in three main sections. Firstly, I will explain the adaptation of the original algorithm. After that, I will dive into sample generation, and finally, I will describe the improvement made to the original algorithm.

4.1 Code adaptation

The first step towards the improvement of the algorithm developed in [1], was to adapt its functionalities to the current requirements. The previous project consisted of three progressively more complex models where the latter was the one that achieved better results. In order to acquire a full understanding of the algorithm, I adapted the three models to my own way of programming, but at the same time keeping the base structure of the algorithm and adapted it to be able to operate with 10-level histograms. The main changes that needed to be made were:

1. Adapting the code to work with larger histograms. From 5-level to 10-level.
2. Adpting and correcting errors in the original representation method (in Matlab).
3. Adadpting the code to my way of programming by changing some functions to work in a way I was more comfortable with.

As briefly mentioned in the introduction section, the main hypothesis was that 10-level histograms would obtain better results than 5-level ones. For that to be proven, I needed to adapt some parts of the algorithm for it to be able to work with lager histograms.

During the early testing of the adaptation, I discovered that the original method of generating binary histograms was outputting some words that contained duplicated letters. By migrating the code from Matlab to Python I was able to determine the reason of that issue and correct it. Even though it didn't make that much of a difference in the results, it did correct some of the words and I thought it would be important further on, since I was going to be generating synthetic datasets based on those histograms, which explained in detail in the following section.

4.2 Synthetic histogram generation

One of the parts of the development that has taken a bigger part of the development time has been sample generation. This particular part of the development wasn't considered in the first proposal of objectives, but due to external causes, the SARS-COV-2 pandemic quarantine, I have not been able to access the test data that I was supposed to test the algorithm improvements with. This was a great issue that had to be solved in order to realistically test the decoder. At this point, we only had binary histograms to test any improvements that involved working with 10 levels, and those results cannot determine the reliability of those improvements for datasets obtained from real images. The solution proposed was to generate histograms that resembled the real ones. Using the real-image dataset from [1] (5-level histograms), I have been able to develop 2 methods of synthetic histogram generation. One that uses the previously generated binary histograms and adds noise similar to the obtained from real images, and a second version that generates levels 6 through 10 of the 5-level histograms obtained from real images.

4.2.1 Binary-based histogram generation

This generation method uses binary histograms, which are perfect representations of the word, and adds noise resembling the one that real-image histograms have. In order to obtain this, I have analysed the values of all 5-level histograms and determined a series of noise parameters. The advantage that this method offers over the other one, is that we can generate datasets as large as we want, since the only resource we need are the histograms we generate ourselves. Also, binary-based generation is quite simple to implement, but it requires a more thorough analysis in order to determine the values of the different parameters used in its generation.

The way this method works is the following. It takes a 10-level binary histogram and goes over every value. Depending on whether the value is a "1" or a "0" we add or subtract a semi-random value. These semi-random values are generated by Python's *random* library and they are composed by a range and a probability value that

determines within which range the value will be generated.

To calculate the parameters, I have portrayed all the values of all the real-image histograms into charts to be able to see what do the real values range between. The charts that I have used are: one containing all values below the acceptance threshold, one containing all values above the acceptance threshold and a last one containing values around the acceptance threshold.

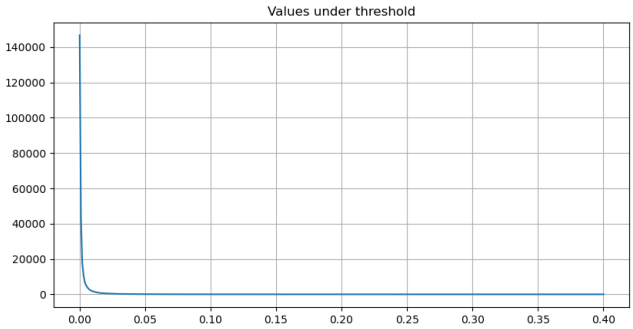


Figure 1. Chart with all values under the acceptance threshold. The x axis represents the number of values and the y axis represents the value they have in the histograms.

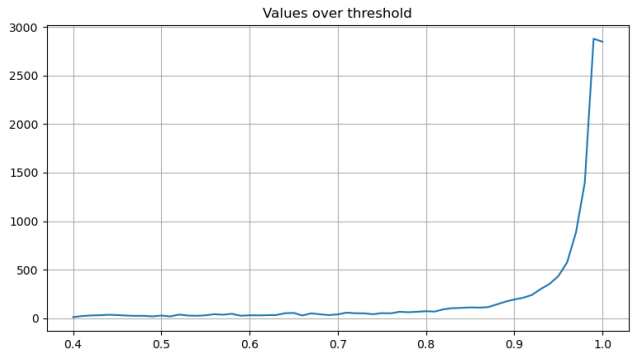


Figure 2. Chart with all values over the acceptance threshold. The x axis represents the number of values and the y axis represents the value they have in the histograms.

These charts have helped me see visually how the real values are distributed in fairly defined ranges. As we can see in Figure 1, nearly all of the values under the trehshold are close to 0. This means that most values that are not being considered as letters have very low noise. Since it is difficult to get actual ranges from reading a chart, I have classified the data using a simple script in Python. I concluded that 90% of all data under the threshold range between 0 and 0.01, and the rest of the values are scattered between 0.01 and 0.4. This 10% has around 9.5% of the values close to 0.01 (we can see this in the slight curve of the chart). As for the remaining 0.5%, we will take them as possible errors, which I will explain with Figure 3.

The values in Figure 2, show a similar result as Figure 1 but inverted. This means that most values that are being considered letters are very near to 1, which again means there is few noise in the data. Nontheless, the data is not as well distributed as in Figure 1. There is a less pronounced curve, which means that values are not concentrated in a small range. When classifiying the values, we see that 80% of the data sits around 0.9 and 1. The remaining 20% has a similar situation to Figure 1, since most values are grouped near the 80%. Still, around 0.5% of these fall in the range that I have set as the "error area" or values around the threshold.

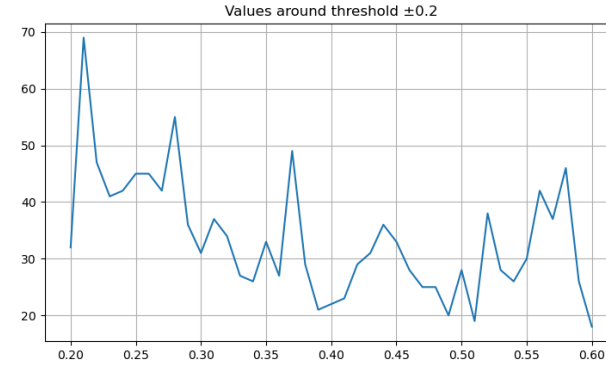


Figure 3. Chart with all values around the acceptance threshold by ±0.2

Lastly, we can see in Figure 3 all the values around the threshold ±0.2, also named "error area". The reason behind choosing this range lies within the results in the [1] concerning acceptance thresholds. In those results, we could see how the accuracy didn't vary much within the range 0.2 to 0.6, and 0.4 had the best results. So, by taking this into account, we can consider this range of values an area of error, where we can find misread values, residual values caused by the noise of the real world and partial readings. However, the actual presence of these values within the whole dataset represents solely a 0.5%.

So, for the first version of these parameters we have that for binary values that are 0 we add a random value between:

1. 0 and 0.01 with a 90% chance.
2. 0.01 and 0.2 with a 9.5% chance.
3. 0.2 and 0.6 with a 0.5% chance.

And for values that are 1 we subtract a random value between:

1. 0 and 0.1 with an 80% chance.
2. 0.1 and 0.6 with a 19.5% chance.
3. 0.6 and 0.8 with a 0.5% chance.

This analysis has been really useful to obtain a solid foundation for the parameters. Nonetheless, the results obtained by testing the generated histograms, using these first parameters, were slightly more optimistic than I needed them to be to replace real samples.

So, to polish the values of the parameters I used the 5-level real-image histograms to compare the similarity to the generated ones. By doing so, I realized that the amount of noise or error that was being used was too low. To fix the values, I ran several tests to determine the

amount of noise that would prove to be as similar to the real histograms as possible. As we will see in the results section in Table 4, that value has been proven to be 5% of error.

To sum up, this generation required a lot of testing and analysis, but it has allowed me to generate a very convincing synthetic datasets that contain almost 90.000 words.

4.2.1 Real image-based histogram generation

Unlike the first approach, this model generates histograms based on the histograms used in the previous iteration of this project [1], which were obtained from real images. The downside compared to the other method is that we are only able to generate a dataset as large as the size of the original 5-level dataset. On the other hand, this generation will surely be more faithful to the original one, so still it is really interesting to develop.

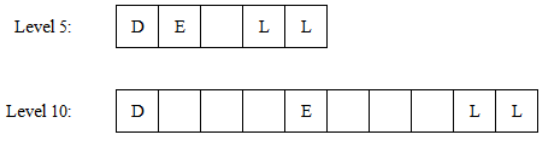
Although this model has more complexity in its development, it requires no analysis since we are building a copy of a 5-level version, thus becoming a more straightforward way of generating the samples.

In order to generate these histograms, we have to add new levels to the existing ones by taking into consideration its already existing values, so we are not going to be generating random histograms. One of the main issues has been that, depending on the length of each word, the level to be added must take into account possible mistakes made in the original histogram, as well as the order of the letters.

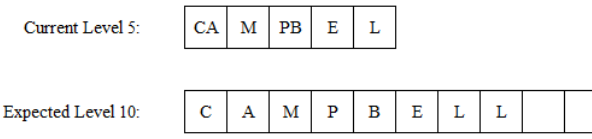
There are 3 word types that have been treated separately:

1. Words with 5 or less letters.
2. Words with more than 5 letters.
3. Words with more than 5 letters and consecutive double letters.

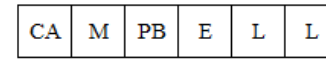
The easiest word type to treat has obviously been the one with 5 or less letters. The generation of their histograms has only consisted on adding empty histograms (with no letter) to fill the remaining levels. This empty histogram has been obtained by getting one random histogram of the word, replacing all its values above the acceptance threshold by a random number between 0 and 0.01, and placing it between other fragments of the level. The position of the empty histogram does not affect the result since only fragments containing a letter are taken into consideration by the algorithm. To have a more visual understanding of the process we can see it in the following image for the word "DELL".



Then, for words greater than 5 the process is more complicated. Now we have to take into account the order of the letters, which is not represented in the fragments of the histogram, and also the possibility that a word that has a double letter is not being represented correctly. To illustrate it the same way as in the earlier example, we will use the word "CAMPBELL".



In order to achieve the expected result, I followed 2 steps. Firstly, I had to determine whether the word contains consecutive double letters. To do so, I used the word itself and not its representation in histograms. If the word had a double letter I checked if it was being represented correctly, meaning that it would be appearing in two consecutive histogram fragments. As we can see in the example above, this did not apply for the word "CAMPBELL". So, in order not to lose letters in our representation I needed to generate a histogram containing the letter, in this case "L", and add it consecutive to the one containing that same letter.
After this step we would have:



Then, to simulate word partitions, I had to split the histograms containing more than one letter so [CA] would become [C][A]. To do so, I had again to use the word to check the correct order of the letters. Once the order was determined, I generated a copy of the current fragment, moved the value of the second letter to the newly generated fragment, and finally placing the fragment next to the current one.

This last process is done until all histograms contain only one letter, or we have generated all 5 remaining levels. For words between 6 and 9 letters long (included) I added empty histograms to reach the 10-level representation the same way I did for shorter words.

To sum it all up, this method ensures a realistic level of noise, but it can only generate the same number of samples as 5-level samples we have, which in this case is around 550 words, much smaller than the binary-based set.

4.3 Algorithm improvements

Throughout the project, there have been many ideas to improve the results obtained by the decoder, but the only one that has been fully tested, and has contributed with very positive results, is the addition of different versions of the same word, or "word versioning".

But before explaining it, I would like to mention some of the ideas that have been discarded. The latest idea has been to use trigrams instead of bigrams when assigning a cost to a possible word path in the graph. This would take into account groups of 3 letters the same way bigrams do.

However, after considering all the changes it required it was quickly discarded since there was no time left to develop it and there were other tasks that had to be done and added more value to the project. Some other ideas are derivated from word versioning, taking other approaches and trying different ways to generate different word versions, but none had proven useful enough, so I discarded their further development early on.

As for the actual improvement, it basically consists in generating 2 additional versions of the word and processing them all at once.

Roughly explained, the original algorithm takes the histogram level closest to the size of the word, which is calculated by checking values over the acceptance threshold. Then a graph is generated from that representation and an optimal word is calculated. By generating different versions of the same word, we attempt to amend some of the inherent errors that histogram representation has. As we saw in the real image-based generation, a double letter may be represented incorrectly in the level that we are going to chose for our representation. But maybe in some other level of the histogram the representation is done correctly.

To explain this in a more visual manner, we can take the word "SUPERSTITION", which is represented like this in its level 10 version:

Level 10:	S	U	P	E	R	S	T	TI	O	N

As we can see we are missing a letter "I". However, it could happen that its level 9 representation was:

Level 9:	S	U	P	E	R	S	TI	TI	ON

And by using this representation of the word we would have better chances of guessing the word correctly.

So, these different versions are added to the graph the same way the original version is added, by adding its first vertices and generating their corresponding sub-graphs. In the following image, we can see how the graphs would be for the word "AND".

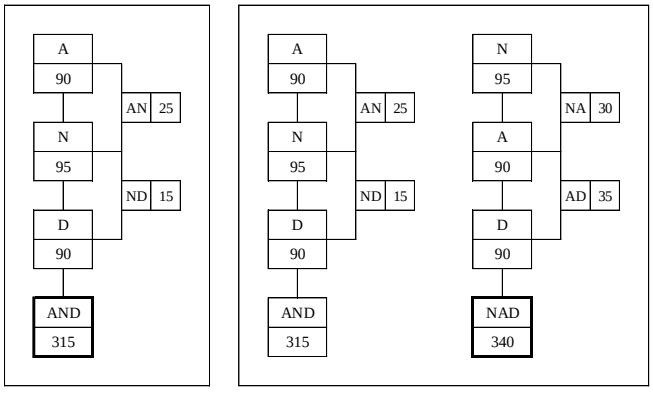


Figure 4. The diagram on the left is generated by levels 3 and 4. The diagram on the right is generated by level 2. Bigger boxes represent a letter and its probability value, and boxes in links represent the probability value of the bigram. The chosen representation for this case would be NAD (value = 340). I have used the word AND for it to be easier to represent.

Unfortunately, this arose some unpredicted issues. As we can see in the previous image, the chosen result (highlited box) is completely incorrect. And if it weren't for the application of the word versioning the calculated word would have been correct. This happened for many words, but mainly for words of a smaller length.

To fix this problem I proposed a slight change to the weighing of the word path costs. Even though we want to take into consideration other levels of the histogram, I believe we still should think that the most optimal representation should be the one that has the same, or closest, number of levels as number of letters. So, to give greater importance to the length-matching representation, I added an extra cost to that particular path in the graph. The value of the extra cost has been determined by testing with different values and the results have been the following.

LENGTH	NO BOOST	10% BOOST	20% BOOST	30% BOOST
1	100,0%	100,0%	100,0%	100,0%
3	78,9%	93,0%	97,2%	97,2%
4	86,4%	96,4%	98,2%	100,0%
5	83,6%	90,4%	94,5%	94,5%
6	82,5%	92,8%	91,8%	90,7%
7	79,7%	87,8%	86,5%	83,8%
8	73,8%	78,6%	76,2%	71,4%
9	68,0%	88,0%	84,0%	84,0%
10	88,0%	88,0%	72,0%	64,0%
11	100,0%	100,0%	100,0%	100,0%
12	25,0%	25,0%	25,0%	25,0%
13	25,0%	25,0%	0,0%	0,0%
14	0,0%	0,0%	0,0%	0,0%
TOTAL	**80,4%**	**89,4%**	**89,2%**	**88,3%**

Table 1. Table with the accuracies obtained by running tests with different boost parameters. Dataset of 546 words generated by binary-based generation.

We will cover the results more in depth in the following section, but as we can see in this example, the values work differently depending on the length of the word. With shorter words we get the best results by adding a 30% of the cost to the chosen histogram, and for longer words adding a 10% worked the best. At first, this was a surprising result, but after analyzing how this addition was changing the behavior of the algorithm, I could draw the following conclusion. Shorter words have lower costs, this means that any variation in the graph path represents a more significant increase or decrease of cost compared to the total. However, for longer words costs are much higher, and one error or variation in the whole path represents a slight change in its total value. Therefore, for shorter words I only took into account the additional versions if they were at least 30% better than the original, and for longer words I only took the words at least 10% better than the original. There is, although, one word length that stays between these 2 groups and that is 5-letter words. After several test runs, the best results were obtained when adding a 20% value to the cost of the orig-

inal histogram. This was the case in several test runs using different data sets, as we will see in the following section.

To sum it all up, there have been several ideas to improve this algorithm, but only the word versioning has given good and coherent results. As it can be seen in the following section, the results obtained have been completely positive.

5 RESULTS AND ANALYSIS

In this section I will go over the most representative results, as well as analyze them in order to draw coherent conclusions.

5-level to 10-level adaptation

This test is the one meant to prove our main hypothesis. We started off this project with the idea that by increasing the number of partitions of a word we would get better results. In Table 2, we can see the accuracy per word length. Taking into account that word from lengths 1 to 10 comprise around 85% of the whole dataset, by getting a 100% accuracy in these smaller words we are greately increasing our overall accuracy. The reason behind accomplishing 100% accuracy is that perfectly represented words, such as the binary dataset, are being divided in such a way that words with length up to the maximum number of partitions, in this case 10, have a representation with histograms containing 1 letter. This means that we don't have to calculate any optimal pathings since there are no other options.

LONGITUD	ENCERTS	ERRORS	ACCURACY
1->10	71891	0	100%
11	5687	1230	82%
12	3316	1130	75%
13	1508	1041	59%
14	669	614	52%
15	244	388	39%
16	78	183	30%
17	32	92	26%
18	5	32	14%
19	3	16	16%
20	0	7	0%
21	0	2	0%
22	0	3	0%
TOTAL	**83433**	**4738**	**95%**

Table 2. Table with the accuarcies of different word lengths using the base algorithm adapted to 10-level histograms. The test is using a 88.171 words binary histogram dataset.

As we can see in Table 3, by increasing the number of partitions to 10, we have increased the overall accuracy by 14% compared to the best result obtained with 5-level binary histograms.

DATASET TYPE	DATASET SIZE (WORDS)	VERSION	ACCURACY
BINARY	88171	5-LEVEL	81%
BINARY	88171	10-LEVEL	95%

Table 3. Table that compares the best result obtained in the original algorithm using binary histograms to the current best result using the same dataset.

Synthetic generation

In this results section I will show the results obtained using the different methods used for generating synthetic histogram datasets. First, we will see the results for binary-based histograms. As we can see in Table 4, I have separated the results by word length ranges. The reason for that is that the datasets used are different. While the real-image dataset contains 546 words, the one I have used for generation has 88.171 words. I chose to use different datasets for 2 main reasons. First, the dataset that I am more interested in using in further tests is the larger one, since larger datasets give more weight to the results. And second, I didn't want to overfit the tuning of the parameters to the smaller dataset, so I wanted to achieve similar accuarcies using different words.

We can see, as well, that I have marked the total accuracy as "pondered". This means that the total accuracy has been calculated by taking into account only the accuracy values of the different word ranges instead of the number of correct and incorrect words. This is, again, because of the difference between the datasets, since the amount of words in the range 5 to 10, for example, does not represent the same proportion in all datasets.

As I mentioned in the corresponding development sub-section, I have chosen to use the generated dataset with 5% error because it resembled a bit more the original one.

LENGTH	REAL IMG	GEN 0.5% ERR	GEN 5% ERR
1 -> 5	88,0%	90,6%	86,4%
5 -> 10	40,3%	40,8%	38,5%
11 or more	0,0%	10,1%	7,5%
TOTAL (pondered)	**42,8%**	**47,2%**	**44,1%**

Table 4. Table comparing the accuracies of real images to 2 binary-based datasets. All datasets consist of 5-level histograms.

As for image-based generation, the comparison between the original 5-level and the generated dataset doesn't give much information because of the difference in levels. However, we can compare the two synthetic generations by generating a 10-level binary based dataset.

As we can see in Table 5, the binary based version has more accuracy since it is being generated from perfect representations. But, despite being slightly less realistic than the image-based representation, we can generate larger datasets, which ultimately, I have considered to be of greater importance.

LENGTH	5L IMAGE	10L IMG BASED	10L BIN BASED
1	0%	0%	100%
3	94%	99%	92%
4	87%	90%	93%
5	89%	90%	92%
6	53%	73%	84%
7	35%	80%	84%
8	33%	79%	69%
9	28%	55%	84%
10	32%	68%	64%
11	0%	11%	100%
12	0%	0%	25%
13	0%	0%	25%
14	0%	0%	0%
TOTAL	**61%**	**79%**	**85%**

Table 5. Table comparing results between the original image histograms and different generation methods. Dataset: 546 words.

Algorithm improvement: word versioning

In this test I will expose the results of the only algorithm improvement that I have managed to develop.

As I explained in the development section, the hypothesis behind this improvement was that by giving the algorithm different versions of the same word and considering different histogram representations, we could correct some errors that may happen in the chosen histogram.

LENGTH	NO WV	WV NO BOOST	WV OPTIMAL BOOST
1	100,0%	100,0%	100,0%
2	95,4%	99,5%	99,5%
3	93,0%	77,6%	97,1%
4	90,3%	81,9%	96,2%
5	87,3%	84,1%	94,7%
6	79,5%	85,3%	92,0%
7	73,6%	83,2%	89,1%
8	72,9%	83,2%	87,1%
9	71,8%	81,2%	84,7%
10	69,1%	79,6%	81,6%
11	59,5%	65,6%	66,4%
12	51,5%	54,0%	53,7%
13	38,1%	38,8%	39,1%
14	31,7%	31,8%	32,3%
15	22,9%	22,9%	23,6%
16	18,0%	18,0%	18,0%
17	13,7%	13,7%	13,7%
18	10,8%	10,8%	10,8%
19	10,5%	10,5%	10,5%
20	0,0%	0,0%	0,0%
21	0,0%	0,0%	0,0%
22	0,0%	0,0%	0,0%
TOTAL	**71,0%**	**77,1%**	**81,9%**

Table 6. Table comparing the results with and without Word Versioning, as well as the usage of boosting parameters. Dataset: 88.171 words.

The results in Table 6 have been obtained by using the same 88.171 word 10-level binary-based dataset for every case. It is shown that we can improve the accuracy around 6% by applying the word versioning technique and improve this by and additional 5% by using the boosting modification mentioned in the development section. The application of a boost, to the already improving word versioning, improves the overall accuarcy a considerable amount. We have to take into consideration that the dataset used contains almost 90.000 words, which means any slight change in the accuracy percentage equals a great number of words that are being predicted correctly. To put it in real numbers, by improving from 71% to roughly 82% we are guessing correctly almost 10.000 words more than before.

Final results: overall accuracy improvement

As we have seen in previous results, the overall accuracy has been improved by applying several modifications to the original algorithm. In Table 7, we can see the comparison of the 3 results I consider to be most meaningful. Firstly, we have the result obtained from the original algorithm using a dataset of 546 words and 5-level decimal histograms. Secondly, we have the result of the first modification, the improvement from 5-level histograms to 10-level ones. This test has been ran using the larger 88.171 word binary-based 10-level dataset. I considered that despite the fact that the dataset is different to the original one, the results were much more meaningful by using a larger dataset. And finally, the results of all the improvements combined, level representation increase and word versioning with its optimization by applying cost boosts. As we can see, the overall improvement has been of 21% accuracy, ending up in a total 82% accuracy for an almost 90.000 word dataset. I believe these results to be very positive since they prove the initially proposed hypothesis.

DATASET	VERSION	ACCURACY
546	ORIGINAL IMAGES 5 LEVEL	61%
88,171	BINARY-BASED GEN 10 LEVEL	71%
88,171	**OPTIMIZED WORD VERSIONING**	**82%**

Table 7. Table comparing the most significat results. The first test uses the 546 word 5-level real-image dataset and the other tests use the 88.171 word 10-level binary-based generated dataset.

6 CONCLUSIONS

This project proposes a series of improvements to an already existing OCR decoder module. Throughout the different stages of the development, we have seen that by tweaking several parameters of the algorithm we can improve the overall accuracy. More precisely, by applying the modifications proposed in the main hypothesis I have proven that the results improve by having larger representations.

Added to the earlier proposed objectives, I have had to tackle the issues that were brought by not having access to the datasets I initially thought I would have. This has added an extra aspect to the project, which has been synthetic generation. This part has been a major part of the

work even though it was never posed in the first stages. Nonetheless, it has been a really interesting diversion to work on and I consider it to have been a success.

As for the overall results, I consider them to be very positive since they prove that the main objectives of this project have been completely fulfilled. The improvement in the results by using only a 10-level representation proves by itself that a more accurate representation of the words improves the accuracy by a great deal, which was the main hypothesis of this project. However, we have seen that we can improve these results even further by tackling the issues brought by the type of representation. With analysis and understanding the behavior of the different parts of the decoder, I have been able to improve the overall results by over 20%.

For future improvements, I belive that there are still modifications to the algorithm or entirely different approaches, that could be applied to the current methodology, that might improve even further the results. Also, a slight improvement to the current methodology could be achieved by tweaking and testing in depth the various parameters that take place into the whole algorithm.

AKNOWLEDGEMENTS

I would like to thank my tutor Ernest Valveny for the great support he has offered me throughout the whole project, by giving me constant feedback and helping me reorganize the project whenever the situation required it.

REFERENCES

[1] G. Jardí, «Implementació d'un mòdul descodificador per un sistema OCR,» *TFG Enginyeria Informàtica, Escola d'Enginyeria (EE), UAB,* 2019.

[2] J. Almazán, A. Gordo, A. Fornés y E. Valveny, «Word Spotting and Recognition with Embedded Attributes,» *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 36, pp. 2552-2566, 2014.

[3] Y. Bassil y M. Alwani, «OCR Post-Processing Error Correction Algorithm using Google Online Spelling Suggestion,» 4 2012.

[4] G. Khirbat, «OCR Post-Processing Text Correction using Simulated Annealing (OPTeCA),» de *Proceedings of the Australasian Language Technology Association Workshop 2017*, Brisbane, 2017.