Workflow for processing digitized documents at the United Nations

João Maria Prieto Dantas Vizoso
joao.vizoso@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

September 2021

Abstract

This work addresses the problem of the digitization of paper documents under the responsibility of the Archives and Records Management Service of the United Nations. Those collections are unique, reflecting the history of the organization since 1945, which motivates its digitization, to promote its easy access. So far, more than 10 million images and more than 5TB of data was created, a large part already available online, but there is still a long way ahead to either digitize what is still in paper and to have an automated workflow to process the digitized images and produce objects for convenient digital access. This thesis therefore aims to build an effective automated workflow to process those digitized images for optimal online publishing and search.
Keywords: Document, PDF, OCR, Workflow, Archive

1. Introduction

In the spring of 1945, it was held in San Francisco, USA, the United Nations Conference on International Organization. As a result of this convention, the Charter of the United Nations [1] was created, allowing the foundation of the organization known today as the United Nations. Apart from this important document, many more documents were produced during that convention, making it necessary to create a documents' service unit, which primary and mostly exclusive job was to maintain custody of those documents.

In the present date, the Archives and Records Management Section of the United Nations is the result of that service that was initiated 75 years ago, being responsible for the large number of documents produced during these years of United Nations existence [4]. With the rise of new technologies and globalization, it became almost mandatory to find a better way to preserve these documents (all of them are unique and some of the documents are very old and therefore very fragile). As a solution, ARMS started to digitize and process some of the documents [7]. Most of these documents are classified for security reasons. However, some of them are declassified and made available to the general public from time to time. When that happens, ARMS uploads those documents into their public database, which is available to everyone with Internet access. So far, more than 10 million images and more than 5 TB of information have been made available, but the process to do this still takes a considerable amount of time and resources (which can be allocated to different tasks), and there are many more documents ready to be digitized, processed and uploaded.

2. Optical Character Recognition

One of the fundamental parts of this dissertation is to convert scanned images into searchable documents and to do so, there is a very useful technology called Optical Character Recognition (OCR). As its name implies, this solution allows recognizing characters contained in a scanned document or image, turning it into text that can be manipulated by a machine. This section briefly describes the core concepts of an OCR system and its different phases.

2.1. OCR review

As mentioned above, an OCR system's main problem is the optical recognition of processed characters by a machine.

Both handwritten and printed characters can be recognized by a typical OCR system, with the performance of their recognition being directly linked with the quality of the input documents (the better the input, the better are the results) and the quality of the software being used (since each software uses its own way to recognize characters).

Some of the typical problems an OCR system can face are variations in the fonts being used, deformations on the document being used as an input, or images and graphics mixed with the text. The usage of different fonts was partially solved with the creation of two standardize fonts, easing the OCR procedure: OCR-A (American version) and OCR-B (European Version). The characteristics of these two types of fonts include the same thickness and width and the distinct shape of every character [2]. While the usage of these fonts, which are typically present in credit cards and bar codes, can improve the accuracy of the optical recognition, it still does not solve the recognition of handwritten characters and other types of fonts that were used in older documents.

Although important steps were made in recent years, there is still no perfect system, so there is always room for improvement. For example, handwriting, historical fonts, and the non-Latin alphabet are the focus of a great variety of studies being done that aim to improve the recognition of these types of characters. In addition, some topics like deep learning and voting systems can also play an important role in implementing new and better OCR solutions [9].

2.2. OCR reference architecture

Since its scanning, an image suffers different types of processing during an OCR workflow until it reaches the final result. Depending on several factors, such as the input/output type or the software being used, an OCR architecture may differ from system to system.

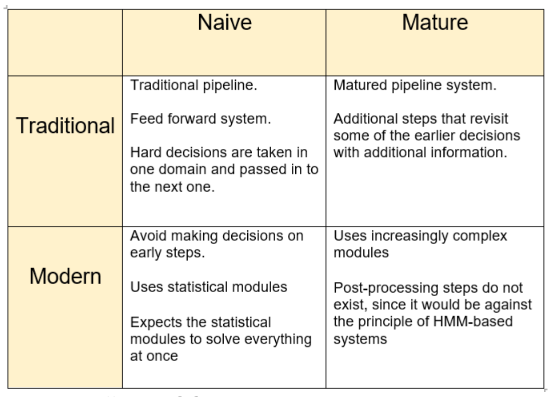
According to Ray Smith, the existing architectures of OCR systems can be divided into four different categories [8] : Traditional-naive, Traditional-mature, Modern-naive and Modern-mature. (Figure 1)

The traditional-naive systems are the ones composed by a traditional pipeline that starts with a series of steps that make hard decisions on one domains and passes the results to the next part of the pipeline.

The traditional-mature are an improvement of the previous ones, where the pipelined systems have additional steps that revisit some of the earlier decisions that were taken, with additional information obtained in other parts of the pipeline. Some examples of this are the adaptive character classification or the use of document dictionaries.

In other hand, the modern-naive OCR systems, try to avoid making decisions at an early stage and push all the hard problems into a monolithic statistical module, such as a Hidden Markov model. The system then expects this module to solve everything at once.

Finally, the modern-mature systems are characterized by increasingly complex models that consider all the printed information structure, since using post-processing modules to revisit earlier decisions would be against the main idea of using HMM-based system.



	Naive	Mature
Traditional	Traditional pipeline. Feed forward system. Hard decisions are taken in one domain and passed in to the next one.	Matured pipeline system. Additional steps that revisit some of the earlier decisions with additional information.
Modern	Avoid making decisions on early steps. Uses statistical modules Expects the statistical modules to solve everything at once	Uses increasingly complex modules Post-processing steps do not exist, since it would be against the principle of HMM-based systems

Figure 1: Different OCR system's architectures, according to Ray Smith [8]

The architecture used as a reference for this dissertation can be split into four main steps (Figure 2), and each one can be divided into several other sub-steps.

The first main step is called Pre-Processing. Considering that digitized images can present a great range of different layouts, fonts, and colors, there is a need to prepare them to obtain a better result when performing the OCR. Some of the most common methods that are used in the pre-processing of the images by the state-of-art software are binarization, in which the main goal is to convert the source image into a binary one (only black and white pixels); deskewing, by adjusting the rotation of the document, making sure it is straight or simply cropping or rescaling the images[5].

The second step is segmentation. Depending on the software used and the user requirements, different types of segmentation can be performed. Usually, there are three types of segmentation: page segmentation, line segmentation, and word segmentation. The first one aims to distinguish between areas of the image that contain text from those that do not. After that, line segmentation splits those found regions into lines of text. Finally, the lines are segmented, resulting in groups of words [5].

The third step is the recognition itself. During this step, segmented words are recognized by a trained model, resulting in a textual representation of the original image. The method used to recognize words is different between OCR engines. [5].

And finally, the fifth step is post-processing. After the words are recognized, the output is ready. However, some sub-steps can be applied during this phase, such as using dictionaries, which will prevent the output from containing non-existing words (sometimes it can be a problem since more technical or made-up words are not present in the dictionary). Considering that the output is normally given in plain text, the post-processing step can also include converting or assembling the final result into more complex formats (such as PDF, for example) [5].

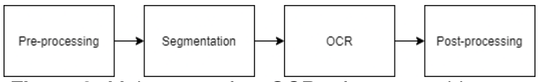


Figure 2: Main steps of an OCR reference architecture

3. Problem Analysis

Currently, the Archives and Records Management Section of the United Nations have their own workflow to digitize and process documents. The existing workflow can be divided in four different tasks, as can be seen on Figure 3.

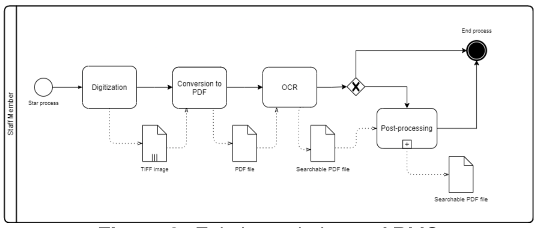


Figure 3: Existing solution at ARMS

This workflow is performed by a member of the staff. This member has the support of an IT team in case there is an issue with the software or any of the IT hardware.

It starts with the digitization of the documents. However, since this step is out of the scope of this dissertation, it will not be included in the proposed solution, and it will be assumed that the documents are already digitized. After the digitization, the documents that were scanned are saved as TIFF images.

After that, the TIFF images are converted to a PDF file with defined settings. This is done so that the OCR can be performed in the next task. Both of these tasks are possible by using the Adobe Acrobat Pro DC software. At the end of this task, the result is a searchable PDF file.

The last step of the workflow is post-processing, it is optional and depends on two conditions that can be divided into two sub-processes. The first is the final file size: if the file is too large, it is compressed or separated. Otherwise, the file is saved. The second condition is the naming convention used: if it is the desired one, the file is saved. If not, it has to be renamed before that happens.

The majority of the problems that occur during the processing of a document are between its digitization and its final state, mainly during the OCR task. This happens because the software used, despite offering that functionality, is not perfect for text recognition on documents. Although it can recognize the regions containing text (i.e., the segmentation seems reasonable), the recognized text is often just “garbage” characters. It also does not offer much flexibility on parameters available since the only settings that can be pre-defined are the language in which the text is written and the format in which the final file should be saved. In addition to that, it has some other points of failure, such as crashing during the processing of a document without providing additional information on what caused it. This issue may not be very relevant for the regular user, but this lack of information on the current state of the process makes it difficult to bypass any of the problems that may arise.

Another problem that the existing workflow has, is its low efficiency since the intermediary steps between each task (for example, place the TIFF image to be converted to PDF or prepare the document to be OCR’ed) have to be performed by a member of the staff manually.

Ideally, the automated workflow should work as a black box. The user (in this case, the staff member) should only be required to place the digitized document in the workflow, select the desired parameters and start the workflow. While the documents are being processed, the user could perform other unrelated tasks without worrying about intermediary steps. In the end, a similar file to the original one is ready, although this is a PDF file and contains the recognized text.

4. Requirements

After presenting the existing solution, the focus of this dissertation is to improve it. Nevertheless, at the same time, this one has to be considered as the minimum acceptable solution possible, so whatever the proposed solution will be, it has to provide the same or better results than the existing one.

Some of the requirements for a new solution are:

- The OCR accuracy should be improved, and new specific features may be added for the specific needs of ARMS.

- The user interface should be easy to use so that a user with basic IT knowledge can perform the tasks needed to use the workflow. A user guide explaining how to install and use the workflow should also be prepared.

- The computers in ARMS have Windows 10 installed, so the proposed solution must be able to run in this operating system. Also, it should only use free-to-use and open-source software.

- The workflow should be modular enough in case new functionalities or parameters have to be added. It also has to keep track of the state of the process so that it can start from where it stopped in case of interruption.

Building a workflow capable of processing all of the documents handled by ARMS with great accuracy, is a difficult task. These documents are very inhomogeneous, containing different sizes, different text structures, and different fonts. Some of the scanned documents are very old, and therefore barely legible, some of them even difficult to recognize by the human eye. However, it is important that the workflow is able to process the majority of the documents.
These requirements were collected during video meetings with the ARMS staff, that occurred every two weeks during the development of this dissertation. Besides gathering the requirements, these meetings would also serve to update the progress.

5. Proposed Solution

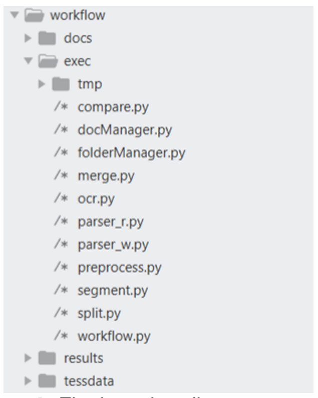
After analyzing the existing problem, it was clear that the proposed solution had to contain the following steps:

1. Split the multi-image TIFFs
2. Apply pre-processing to each image
3. Perform a first OCR run
4. Segment the images in order to find regions of text
5. Perform a second OCR run on the segmented images
6. Compare the results of both runs and chose the best ones
7. Generate a single PDF file

The first step is needed because after the digitization, the scanned images are stored in TIFF files, the majority of them containing more than one image. Since every image needs to be processed independently, they have to be split first. The focus of the second step is to improve the quality of the images so that the OCR process can produce better results. Steps 3 to 5 are an attempt to improve the accuracy of the OCR processing. The first OCR run is performed with the default settings, while the second run is performed on segmented parts of the text, using specific parametrization. The goal is to create an additional "opinion" of the software on what text should be recognized in those images. The sixth step is creating some form of comparison to decide on which opinion is the right one. Finally, the last step merges the results of the previous steps into a single searchable PDF file.

5.1. Development

The approach chosen to build this version was the creation of scripts for each part of the process. These scripts were then coordinated by another main script, that would make the connection between them. For this version, eleven different scripts were created: one for each step of the process (split.py, preprocess.py, segment.py, ocr.py, compare.py and merge.py), four auxiliary scripts to keep track of the state of the process, manage the creation/deletion of folders/files and act as parsers (docManager.py, folderManager.py, parser_r.py and parser_w.py) and one main script to manage all the other scripts (workflow.py). In addition to the scripts, this workflow also followed a predefined directory structure, as seen in Figure 4.



```
workflow
  docs
  exec
    tmp
    compare.py
    docManager.py
    folderManager.py
    merge.py
    ocr.py
    parser_r.py
    parser_w.py
    preprocess.py
    segment.py
    split.py
    workflow.py
  results
  tessdata
```

Figure 4: Final version directory structure

Trained data for specific languages being processed by the OCR, other than English should be placed in the tessdata folder, after being retrieved from the Tesseract repository.

5.2. Technologies used

The scripts were written in Python (version 3), and all the libraries and packages used would have to be compatible with this coding language. This was the chosen programming language due to its simplicity and vast library support. The developing environment was a Linux virtual machine, running Ubuntu 16, with 6GB of RAM and 20GB of disk space.
For the OCR processing, after testing some of the existing solutions, Tesseract was the chosen one. This open-source software provides good results when recognizing text, it is easy to integrate with Python, and it is well documented.

5.2.1 User interface

One of the goals when designing this workflow was to create a user interface. This interface should be easy to use and should not require any extraordinary IT knowledge to complete the tasks needed to run the workflow. However, during the design of this solution, cooperation with another master's dissertation emerged, with the intention of assisting in this project and giving their contribution in other fields of the workflow. Therefore, design and development decisions had to be made together with the others involved. One example of this collaboration is the creation of a user interface, which was no longer part of the scope of this dissertation. Therefore, the interaction with this version of the workflow is done through the command line.

5.2.2 tesserocr

In the first version of the workflow, the chosen tool to use Tesseract was the python library pytesseract. This library is just a wrapper of the tesseract-ocr command-line interface, which means that every time the function to recognize the text is called, it loads the model and processes the image. This library was not fast enough and did not allow much flexibility when choosing specific parameters.
After some research, another Python tool that dealt with the Tesseract was discovered. The name of this tool is tesserocr, and it also consists of a Python wrapper, but this one is a wrapper around the Tesseract C++ API. By interfacing directly with the API, there is more flexibility, and other advanced features can be used. The trade-off is understanding its behavior since using it is more complex than the previous solution. Some features allowed with this tool are the introduction of new trained data for specific languages (more than 100 options available), the access to information and decisions being made by the OCR engine, or the definition of the page segmentation method to be used.

5.3. Execution

To run the workflow, the following command should be used in the Command Line:

```
$ python3 workflow.py path_to_file
```

Another feature that was added to this version was the introduction of parametrization, being possible to define some parameters when running the workflow. In this particular version, there are seven different parameters that could be defined, but more can be added in the future. Those seven parameters are:

- –comp - Disables the line segmentation comparison, as explained in the “Segmentation” subsection.
- –folder - Instead of running the workflow in just a single file, all the files in the specified folder are processed.
- –force - If the workflow was interrupted, instead of continuing from where it stopped, it starts from the beginning.
- –help - Shows how to run the workflow and shows the existing parameters.
- –lang - This parameter is used to tell the workflow in which language the documents are written. More than one language can be selected. Default is English. This parameter is important for the OCR processing, and its use is recommended. Available languages have to be downloaded previously.
- –prep - Performs some additional pre-processing techniques to the images, as explained in the “Pre-processing” subsection.
- –tmp - Keeps the temporary files instead of deleting them at the end of the workflow.

When executing the previous command with or without any of these parameters, the workflow execution starts.

5.3.1 Prologue

The workflow starts its execution by creating a process. This process can correspond to the execution of a single file or the execution of several files in a specific folder, in case the –folder parameter was used.
The creation of a process is characterized by the creation of a JSON file in the tmp folder, as well as the creation of a folder inside the tmp and results folders, with the name of the original file (or the name of the original folder), which is also the name of the process by default. The JSON file contains only one entry with two different fields: the name of the process and the already processed files included in this process (which is empty at the beginning of the execution)
The purpose of this file is to keep the state of the workflow in case there is an interruption and to prevent overwriting in case the workflow is used again with the same files. Every time the workflow is started, it checks if a process with the same name already exists. If the answer is yes, it continues the process from where it stopped unless the –force parameter is used. In that case, all the progress until then is deleted, and the workflow starts from the beginning.
To keep track of the state of a specific file, another JSON file is created inside the corresponding folder of each process. There is one JSON file for each file in that process. This data file contains only one entry with the following fields: name of the file, path to the original file, and status. The

status field contains six other fields corresponding to each step of the process, namely split, preprocess, segment, ocr, compare and merge.

5.3.2 Splitting

The first step is splitting the pages in case there is a multi-page TIFF image. This splitting is performed by the split.py script. The script uses the Python Image Library (PIL) to handle the multi-image TIFF files, specifically the Image module. With the seek() and save() functions, it is possible to split the TIFF multi-paged files into separated ones containing only one TIFF image per file. These files are saved, following the naming convention “page_” + “#number of the page” in the tmp folder.

Input: multi-paged TIFF image with n pages
Output: n TIFF images , with n=number of pages

5.3.3 Pre-processing

One of the steps that was added to this version was pre-processing. In this step, performed by the preprocess.py script, additional pre-process techniques are applied to the split TIFF images. There are two options when running this script. In the default option, the thresholded images produced by the internal process of the Tesseract API are saved on the tmp folder. The other option occurs when the –prep parameter is used. When selecting this option, four additional techniques are applied, using the cv2 library,
The images resulting from this step are the ones that are going to be used on the rest of the workflow, except the merge step, where the original images are the ones used.

Input: n TIFF images
Output: n TIFF processed images

5.3.4 Segmentation

This step is performed by the segment.py script, and can be divided into two different sub-steps. The first sub-step to perform the segmentation of the images is a first OCR run. This run is done with the default settings of the Tesseract API and results on an hOCR file for each one of the images. These hOCR files are stored in the tmp folder.
In the second sub-step, with the information gathered in the hOCR files, all the text lines in the images are cropped and saved as separated images. After that, the information regarding all the recognized text lines is stored in a JSON file, called regions.JSON. There is an entry for each one of the images, each entry containing the corresponding information. The information gathered includes the id of the text line, the bounding box values (bbox property), the coordinates of the text line in the image, and the path to the corresponding hOCR file. Additional fields like the recognized text and the words confidence are also created but remain empty until the next step.

Input: TIFF image
Output: n hOCR files, with n=original TIFF images + n TIFF images, with n=number of segments found

5.3.5 OCR

The OCR step, which is actually the second time the Tesseract API is called to do text recognition, is performed by the ocr.py script. This time the recognition is done on the text segments collected in the previous step. By calling the Tesseract API, with the ”Page Segmentation Mode” parameter set to PSM.SINGLE_LINE, it is expected that the recognition results will be more accurate because the system is being told that the images being processed contain only one line of text. In addition to the recognized text, the corresponding word confidences are collected as well. In the end, the regions.JSON is updated on all the text and word_conf fields.

Input: n TIFF images, with n=number of segments + regions.JSON
Output: updated regions.JSON

5.3.6 Comparison

Now that there are two different OCR runs, there is a necessity to compare which one contains the best results. To do so, there is a step called Comparison, performed by the compare.py script.
In this step, the text recognized in the first OCR run, made on the Segmentation step and stored in the hOCR files, and the text recognized in the previous run and stored in the regions.JSON file is compared.
This script compares all the text lines found in both runs and checks if the word confidences are different between the corresponding bounding boxes. If the word confidences is different, the highest one is chosen. This leads to a change on the hOCR file, where the recognized text and their corresponding word confidences are updated.
To parse and update the hOCR file, two additional auxiliary scripts were created. The first one was the parser_r.py, that uses the bs4 library, usually used for scrapping information out of web pages. This script allowed parsing the content contained in the hOCR files. The second script was the parser_w.py that used the xml library, specifically

the ElementTree module, used for creating XML data, to write into the hOCR files.
In the case where the –comp parameter was used, this step did not occur, and only the recognized text of the first OCR run would be considered. This parameter could be used in cases where time efficiency is more important than accuracy.

Input: n hOCR files + regions.JSON
Output: n updated hOCR files

5.3.7 Merging

Finally, the last step was merging all the data back into a single PDF. This step was performed by the merge.py script. This script is an adaptation of an open-source script that is part of the hocr-tools repository [3].
After the hOCR files being ready, their content is parsed again to gather the recognized text and its location on the image. Using the open-source version of the Report Lab tools, which includes a Python library, a PDF file with two layers is generated: the first layer is the original TIFF image, while the second layer is the recognized text in its correct position, written in an invisible font. This is done for each hOCR file and its corresponding TIFF image, creating the individual pages of the PDF file. In the end, a PDF file with all the pages is generated, resulting in the desired outcome of the workflow.

Input: n hOCR files + n original TIFF images
Output: PDF file

5.4. Deployment

As it was mentioned before, the environment of development of this solution was Linux. However, the workflow was designed to work in all platforms, specifically on Windows10, as stated in the requirements. When the deployment was done to this operating system to test its feasibility, it was found that one of the components of the workflow, although compatible with Windows machines, was too complex for someone with basic IT knowledge to install. This component that was part of the Tesseract software would not allow the workflow to be natively installed on ARMS computers.
To solve this setback, it was necessary to add another task to the development of the proposed solution: create a virtual environment to run the workflow. Taking this into account, and after considering between two solutions, the creation and configuration of a virtual machine or the creation of a Docker container, the latter was chosen.
After some research about this technology, a Dockerfile containing all the steps necessary to install and configure all the components of the workflow was produced, enabling the deployment of the workflow on Windows computers.
The Docker image, besides all the required software and packages that needed to be installed, also included the creation of a Docker volume, which is a file system mounted on the Docker container in order to preserve the data generated by the running container. In other words, it works as a shared folder between the virtual environment and the host computer. This feature is useful so that the users will not have to worry about transferring the images to be processed into the virtual environment, since they only must be placed in a specific folder and then be accessed by both the host and the virtual environment.

6. Assessments

The assessment of the developed workflow can be split into two different phases. First, after the development ended, the workflow was deployed and installed in the ARMS computers to test its feasibility in a real-world environment. Second, at the same time, another phase to evaluate the workflow was performed in the same environment as it was developed. This evaluation was performed both in terms of efficiency and effectiveness.

6.1. Real-world environment

The assessment on a real-world environment is fundamental since it is this assessment that will tell if the workflow can be used by ARMS.

6.1.1 Installation and setup

To perform this evaluation, two sessions were scheduled to install the workflow. One session with a staff member responsible for digitizing and processing the digitized images (next referred to as User1), and another session with a staff member responsible for other tasks, not directly related to the processing of the digitized images (referred to as User2). Prior to these sessions, the code was made available in a GitHub repository, together with a user guide prepared specifically for this purpose.
In addition to the source code, the only thing that was required to install previously was the Docker desktop software and the Linux kernel update package.
The sessions to install the workflow were performed using the shared screen feature of the Microsoft Teams software. To assess how easy it was to install it, the users were required to try to perform the installation by themselves, following the steps on the user guide. However, if there were questions, they could be answered to ensure that the workflow would work.

The first session with User1 took about thirty minutes. Issues related to the installation and setup of the Docker desktop software in ARMS computers were the reason why it took so long. These computers have a strict policy on the type of software that can be installed, for security reasons, so administrator privileges were needed.

Another issue raised during that installation session was that the first version of the user guide required the user to introduce some commands on the Windows command line. Since the user did not have enough IT knowledge to perform those actions, two bash scripts were prepared to ease its work. One bash script to build the Docker image from the Dockerfile, that only has to be used once, and another bash script to run the Docker, that has to run every time the workflow is used.

For the second session, the user guide was changed to introduce the recently created bash scripts. After these changes and with the previous experience obtained in the previous session, it was easier to solve the issues that could arise during the installation and setup of the workflow. For that reason, the session took less time, being completed in about fifteen minutes.

The use of Docker containers to deploy the workflow ended up being a good idea since it turns the process a lot easier, taking less time to do it. In addition, instead of having to install all the libraries and dependencies, only the Docker software has to be installed.

6.1.2 Testing

The testing of the workflow in a real-world environment was performed for two weeks. Unfortunately, due to delays in the elaboration of this thesis, it was not possible to perform extensive testing with an appropriate results collection.

The workflow was tested with a few scanned documents by User1, and the feedback was given by e-mail and through one of the biweekly meetings.

The user reported two different bugs on the execution of the workflow and one processing error. Both bugs were related to problems in the development of the workflow and were quickly fixed. The processing error happened during the processing of one of the samples, and it was caused by one faulty TIFF image that was causing the workflow to crash.

Positive feedback was also given, and accordingly, to the user who performed the tests, the OCR results are better than those obtained by the previously used software, Adobe Acrobat Pro DC.

In addition to that, the user also stated that running the workflow without the –prep parameter produced very bad results when compared with the results obtained with that parameter. However, the option to use the workflow without this parameter was kept for when the scanned documents are already legible and do not need that extra step.

6.2. Development environment

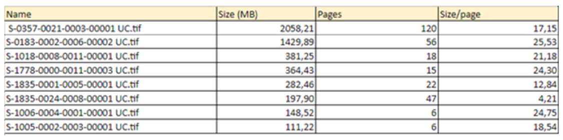
Besides the functional assessment made on real-world environment, a more extensive testing had to be performed to measure the performance of the proposed solution. For this purpose, a batch of files, containing TIFF images, were prepared, and made available by the ARMS team, to be tested during and after the development of the workflow.

The performance was measured in two different ways. It was measured in terms of efficiency, where the chosen metric was the time each step takes to process. It was also measured in terms of effectiveness, where the OCR accuracy is assessed based on what the expected result is. The environment used to obtain the results was a machine running Windows10, with an Intel(R) Core (TM) i7-10510U CPU @ 1.80GHz 2.30 GHz and 16GB RAM. The workflow was running inside a Docker container, running Ubuntu18.

The results on the assessment of the efficiency and effectiveness of the developed workflow are presented in the next two subsections.

6.2.1 Efficiency

As stated above, the chosen metric to measure the efficiency of the proposed solution was the time the workflow took to perform each task. For this evaluation, nine different files were chosen from the available batch. Although it is difficult to obtain a set of documents that represent the majority of the documents usually processed by ARMS, these documents were chosen based on their size and number of pages, with the objective of obtaining a greater representation of what the documents processed by the workflow may be. The size of each file (in megabytes) and their number of pages can be seen in Figure 5. As it is possible to observe, the documents vary from 111,22MB and 6 pages to 2058,21MB and 120 pages.

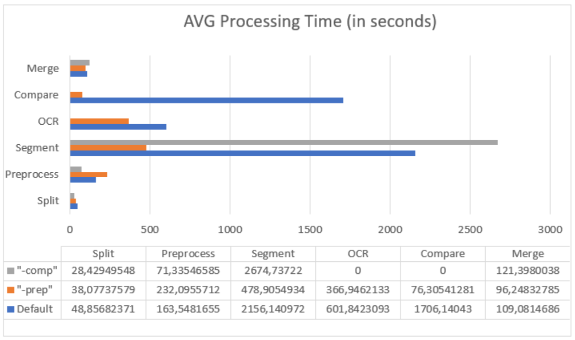


Name	Size (MB)	Pages	Size/page
S-0357-0021-0003-00001 UC.tif	2058,21	120	17,15
S-0183-0002-0006-00002 UC.tif	1429,89	56	25,53
S-1018-0008-0011-00001 UC.tif	381,25	18	21,18
S-1778-0000-0011-00003 UC.tif	364,43	15	24,30
S-1835-0001-0005-00001 UC.tif	282,46	22	12,84
S-1835-0024-0008-00001 UC.tif	197,90	47	4,21
S-1006-0004-0001-00001 UC.tif	148,52	6	24,75
S-1005-0002-0003-00001 UC.tif	111,22	6	18,54

Figure 5: Files used in the assessment

To measure the time each task took to finish, the source code of the workflow was changed, to retrieve that information. The obtained values were then saved into an Excel file.

The obtained values were collected after the workflow was used with three different parametrizations: with the default parameters, with the –prep parameter and with the –comp parameter. The use of –prep parameter was important to see if improved images resulted in an improvement of the execution time. On the other hand, the use of the parameter –comp only serves to compare the final execution time, since two steps (second OCR pass and comparison) are skipped. Figure 6 shows the average execution time of each task for the three different parametrizations and the total times for each parameter can be checked in Figure 7.



	Split	Preprocess	Segment	OCR	Compare	Merge
■ "-comp"	28,42949548	71,33546585	2674,73722	0	0	121,3980038
■ "-prep"	38,07737579	232,0955712	478,9054934	366,9462133	76,30541281	96,24832785
■ Default	48,85682371	163,5481655	2156,140972	601,8423093	1706,14043	109,0814686

Figure 6: Average processing time using three different parametrizations

Used parameter	Total time (in seconds)
--comp	2895,900
--prep	1288,578
default	4785,610

Figure 7: Total processing time using three different parametrizations

By looking at the results obtained it is possible to reach the following conclusions:

- Overall, the workflow usually performs faster if the –prep parameter is used. It performs the slowest using the default parameters.
- The only task where the –prep parameter takes longer than the others, is the Preprocessing.
- The segmentation task takes much more time if no additional pre-processing is provided.

6.2.2 Effectiveness

The accuracy of the OCR was the chosen metric to measure the effectiveness of the workflow. However, since the majority of the documents at ARMS do not present well-structured text, it is difficult to calculate specific metrics like the Character Error Rate (CER) and the Word Error Rate (WER) using the original images. In addition to that, measuring the OCR accuracy implicates the manual transcription of the images to obtain the ground truth (i.e., the expected result), which is a laborious and time-consuming task.

To perform this assessment, three images were cropped so that the OCR system processed only well-structured text. Next, those images were processed by the workflow developed in this dissertation and by the Adobe Acrobat Pro DC software. After that, the results of each run were compared with the expected result after manually transcribing the images.

In addition to the visual comparison, the comparison method was the calculation of the Character Error Rate and the Word Error Rate. These metrics use the Levenshtein distance [6] to calculate differences between two different strings. The CER calculates the differences on a character level, while the WER calculates it on a word level (which is more relevant in this project's scope). The closer these values are to zero, the fewer errors there were when recognizing the text. The calculation of these values was done through a Python library called fastwer.

The chosen images were selected accordingly to their legibility. The first image contained black text on white background, providing good legibility. The second image was less legible since it had black text on beige background. Finally, the third image was even less legible due to its light blueish text on beige background.

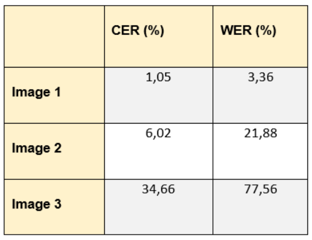
To compare the results, the workflow was used with two different parameters: the default parameters and the –prep parameter. This way, it was possible to infer how helpful the latter could be.

The obtained results when using the workflow with the default parameters, can be seen in Figure 8.

	CER (%)	WER (%)
Image 1	1,05	3,36
Image 2	68,72	76,04
Image 3	16,17	27,42

Figure 8: Obtained results using the default parameters

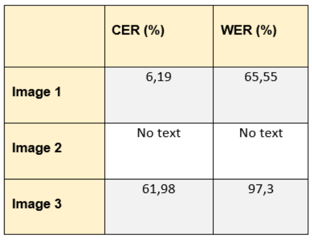
The results when using the workflow with the –prep parameter, can be observed in Figure 9.



	CER (%)	WER (%)
Image 1	1,05	3,36
Image 2	6,02	21,88
Image 3	34,66	77,56

Figure 9: Obtained results using the –prep parameter

Finally, the results obtained by the Adobe Acrobat Pro DC are presented in Figure 10.



	CER (%)	WER (%)
Image 1	6,19	65,55
Image 2	No text	No text
Image 3	61,98	97,3

Figure 10: Obtained results using the Adobe Acrobat Pro DC

When analyzing the results, it is possible to conclude that in the set of images used, the proposed solution of this dissertation provided better results than the Adobe Acrobat Pro DC software, when recognizing the text contained in the images. Regarding the use of the –prep parameter, the results have shown that it can be useful in a certain type of images. However, it does not always provide better results than the default parameter, that is why it is necessary to keep both options available.

7. Conclusions

As mentioned earlier, it was necessary to develop a new workflow for processing documents scanned for ARMS since the current workflow lacked automation, and sometimes satisfactory results. It was with this in mind that this dissertation was composed.

After presenting the problem, it was necessary to start outlining what would become the final solution. For this, it was essential to deepen the knowledge about the relevant file formats, namely PDF and TIFF. Furthermore, after this study, it was also essential to know the state of the art of OCR technology since this is one of the fundamental parts of the workflow. After gathering all this knowledge, an analysis of the existing solution and the stakeholder requirements was made to design a solution.

In an initial phase, this development resulted in a simple workflow with the goal of serving as a proof-of-concept. The goal of this design choice was to have a complete (i.e., workflow that functioned from start to finish) to present to the ARMS team.

Since this workflow was too simplistic and had the potential to add other functionalities, the initial workflow started to be incremented until it reached a final version. In this final version, new functionalities such as pre-processing the images or using two different segmentation types were added.

Upon completion of this final version, it was deployed to the computers of the ARMS team in order to test the feasibility of using this workflow. At the same time, performance assessments were made, to measure the efficiency and effectiveness of the workflow. The results of these evaluations were satisfactory, although there is room for improvement.

In conclusion, this dissertation was able to deliver what it set out to do, namely a complete workflow capable of processing a set of digitized images, resulting in a PDF file with a text layer, ready to be uploaded to the ARMS public database.

References

[1] United nations charter and statute of the international court of justice, 1945.

[2] N. Anderson, G. Muhlberger, and A. Antonacopoulos. Optical character recognition impact best practice guide impact project. 2013.

[3] T. M. Breuel. hocr-tools. https://github.com/ocropus/hocr-tools, 2013.

[4] R. Claus. The united nations archives. *The American Archivist*, 10(2):129–132, 1947.

[5] N. Islam, Z. Islam, and N. Noor. A survey on optical character recognition system. *ITB Journal of Information and Communication Technology*, 12 2016.

[6] A. S. Lhoussain, G. Hicham, and Y. Abdellah. Adaptating the levenshtein distance to contextual spelling correction. *International Journal of Computer Science and Applications*, 12(1):127–133, 2015.

[7] T. Newton. Record-keeping requirements for digitization, Apr 2009.

[8] R. Smith. History of the tesseract ocr engine: what worked and what didn't. *Proceedings of SPIE - The International Society for Optical Engineering*, pages 02–, 02 2013.

[9] A. Somani. Council post: The future of ocr is deep learning, Sep 2019.